В.Л. Тарасов

Лекции по программированию на C++

Лекция 5

Функции

Термином *функции* в языке C++ обозначают подпрограммы. В виде функций оформляются вычисления, преобразования, другие действия, которые затем могут быть выполнены неоднократно с различными аргументами.

5.1. Определение функции

Определение любой функции имеет вид:

```
тип-результата имя-функции(объявления аргументов)
{
   объявления и инструкции
}
```

Отдельные части определения могут отсутствовать, например, следующая функция ничего не делает:

```
void dummy() {}
```

Тип результата функции `void` означает, что функция ничего не возвращает.

Вычисления в функции завершаются при достижении последней закрывающей фигурной скобки `}` или при выполнении инструкции `return`. Когда функция завершает работу, управление передается в вызывающую программу той инструкции, которая следует за вызовом функции.

Если функция завершается инструкцией

```
return выр;
```

то вычисляется выражение `выр` и его значение передается в вызывающую программу в точку вызова функции. Значение, возвращаемое функцией, может быть использовано вызывающей программой или проигнорировано:

```
cout << sin(M_PI / 6);   // Значение, возвр. функцией sin используется
```

```
sin(M_PI / 2);          // Значение, возвр. функцией sin игнорируется
```

Функции могут возвращать значения *любых* типов. Например, следующая функция возвращает логическое значение `true` (истина), если ее аргумент `a` больше `b`, иначе значением функции будет `false` (ложь).

```
bool greater(int a, int b)
{
   return a > b;
}
```

5.2. Формальные параметры и фактические аргументы

Аргументы, используемые при определении функции, называются *формальными параметрами*. В следующей программе приведен пример, функции, вычисляющей сумму своих аргументов.

Программа 5.1. Сумма аргументов

В состав программы входит функция `int sum(int a, int b)`, которая вычисляет сумму аргументов `a` и `b` и возвращает эту сумму как результат своей работы. Код программы размещается в одном файле: сначала функция `sum()`, затем – функция `main()`, из которой вызывается `sum()`.

```
// Файл SumArgs.cpp

int sum(int a, int b)    // Возвращает сумму аргументов
{
   return a + b;
}

#include <iostream>
using namespace std;
int main()
{
   cout << "3 + 2 = " << sum(3, 2) << endl;   // Фактические аргументы –
                                              // константы
   int x = 3, y = 2;                          // Переменные целого типа
   cout << "x + y = " << sum(x, y) << endl;   // Фактические аргументы –
                                              // переменные
   return 0;
}
```

Здесь `a` и `b` – формальные параметры функции `sum()`.

При вызове функции вместо формальных параметров подставляются *фактические аргументы*. Первый вызов функции имеет вид `sum(3, 2)`. Здесь фактическими аргументами являются константы `3` и `2`.

во втором вызове (sum(x, y)) фактическими аргументами являются переменные x и y.

Программа выводит:

```
3 + 2 = 5
x + y = 5
```

В языках Си и C++ аргументы передаются в функции *по значению*. Это реализуется созданием внутри функции *локальных копий* фактических аргументов, которые и используются в вычислениях. Отсюда следует, что фактические аргументы не могут быть непосредственно изменены внутри функции.

Программа 5.2. Передача аргументов по значению

Входящая в состав программы функция add(int c) пытается изменить свой аргумент. Но это ей не удается, так как изменяется только *копия* аргумента.

```
// Файл ChangeArgs.cpp

void add(int c)          // Попытка изменить аргумент
{ c++; }

#include <iostream>
using namespace std;

int main()
{
   int x = 1;
   add(x);
   cout << "x = " << x << endl;
   add(2);
   return 0;
}
```

Здесь делается попытка увеличить переменную x на 1 и попытка изменить значение константы 2. Программа выведет:

```
x = 1
```

Таким образом, значение аргумента x не изменилось, так же, как ничего не произошло с константой 2. Первый вызов функции имеет вид add(x). Здесь формальный параметр c инициализируется *значением* переменной x. Во втором вызове (add(2)) формальный параметр инициализируется константой 2. Внутри функции add() переменная c изменяется, что никак не сказывается ни на x, ни на константе 2.

Память под формальные параметры функции выделяется при вызове функции и освобождается при завершении ее работы.

Если необходимо изменить фактический аргумент внутри функции, в нее должен передаваться *адрес* аргумента или *ссылка* на аргумент. Как это сделать, будет описано позже.

Программа 5.3. Степени целых чисел

В программе определена функция power() для возведения целого base в целую степень degree. В main() функция power() вызывается для вычисления степеней чисел 2 и -3.

```
// Файл PowerInt.cpp

// Функция power() возводит base в степень degree

int power(int base, int degree)
{
   int i p = 1;                        // p - переменная для степени
   for(i = 0; i < degree; i++)         // degree раз
      p = p * base;                    // умножаем   на base
   return p;          // Возвращение вычисленного значения p из функции
}

#include <iostream>
#include <locale>
#include <cstdlib>
using namespace std;

int main()                          // Использование функции power
{
    setlocale(LC_ALL, "Russian");
    for(int i = 1; i < 10; i++)
      cout << "2 в степ. "  << i << " =\t" << power(2, i)
    << "\t-3 в степ. " << i << " =\t" << power(-3, i) << endl;
   system("pause");
   return 0;
}
```

Программа выдает следующее:

```
2 в степ. 1 =    2   -3 в степ. 1 =   -3
2 в степ. 2 =    4   -3 в степ. 2 =   9
2 в степ. 3 =    8   -3 в степ. 3 =   -27
2 в степ. 4 =    16  -3 в степ. 4 =   81
2 в степ. 5 =    32  -3 в степ. 5 =   -243
2 в степ. 6 =    64  -3 в степ. 6 =   729
2 в степ. 7 =    128 -3 в степ. 7 =   -2187
2 в степ. 8 =    256 -3 в степ. 8 =   6561
2 в степ. 9 =    512 -3 в степ. 9 =   -19683
```

В данной программе сначала идет определение функции power() с двумя аргументами целого типа. Для возведения base в степень degree создается переменная p с начальным значением 1, которая умножается

на base в цикле degree раз. Вычисленный результат возвращается из функции инструкцией:

```
return p;
```

Внутри функции power() определены локальные переменные i и p. Они создаются (под них выделяется память) только в момент вызова функции, а при завершении работы функции память, занимавшаяся локальными переменными, освобождается. Это значит, что то место памяти, которое занимали i и p, может быть использовано для других переменных.

Функция main() направляет значение, возвращаемое функцией power(), в выходной поток.

Сама функция main() вызывается на выполнение операционной системой, а возвращаемое ею значение может быть каким-либо образом использовано в среде ОС, например, для анализа успешности завершения программы.

Обратите внимание на использование символа табуляции \t. Вывод символа табуляции переводит курсор в следующую позицию табуляции, которые во всех строках экрана находятся в одинаковых местах. Благодаря этому результаты вывода выравниваются по левому краю.

5.3. Автоматические и статические переменные

Переменные, определенные в теле функции, являются *локальными* внутри этой функции и, поэтому, недоступны для других функций. Существуют два вида локальных переменных: *автоматические* и *статические*.

Автоматические переменные создаются только в момент обращения к функции и исчезают при завершении ее работы. Память под автоматические переменные выделяется, когда функция начинает работать, и освобождается, когда функция завершается, поэтому автоматические переменные *не сохраняют* своих значений между вызовами функции и должны устанавливаться заново при каждом новом обращении к функции.

Если локальные переменные объявить *статическими* с использованием ключевого слова static, то под них будет выделяться постоянная память при начале работы программы, поэтому их значения будут сохраняться между вызовами функции. В программе 5.4 функции fauto() и fstat() увеличивают на 1 свою локальную переменную k и возвращают ее значение.

Программа 5.4. Автоматические и статические переменные

```
// Файл AutoStat.cpp

int fauto ()
{                   // Автоматическая локальная переменная k
    int k = 0;      // инициализируется при каждом вызове функции
    return ++k;     // Изменение локальной автоматической переменной
}

int fstat ()
{                   // Статическая локальная переменная k инициализируется
    static int k = 0;    // только при первом вызове функции
    return ++k;          // Изменение локальной статической переменной
}

#include <iostream>
using namespace std;

void main()
{
    for(int i = 0; i < 10; i++)       // 10 раз вызываем fauto()
        cout << fauto() << " ";
    cout << endl;
    for(int i = 0; i < 10; i++)       // 10 раз вызываем fstat()
        cout << fstat() << " ";
    cout << endl;
}
```

Программа выводит:

```
1 1 1 1 1 1 1 1 1 1
1 2 3 4 5 6 7 8 9 10
```

Память под статические переменные выделяется в самом начале работы программы. Тогда же производится их инициализация, то есть статическая переменная инициализируется *только один раз* при первом вызове функции. Так как статическим переменным выделяется постоянная память, они сохраняют свои значения между вызовами функции. Статическая переменная, объявленная внутри функции, является локальной, то есть она доступна только внутри своей функции.

Все переменные внутри функции являются по умолчанию автоматическими, но можно явно указать на это, используя ключевое слово `auto`:

```
auto int k = 0;                // Автоматическая переменная
```

Параметры функций создаются заново при каждом вызове функции и инициализируются значениями фактических аргументов, поэтому они также являются локальными автоматическими переменными. Парамет-

ры функции можно произвольно изменять внутри функции, причем это никак не влияет на значения фактических аргументов. Например, функцию power() можно переписать в виде:

```
int power(int base, int degree)       // Возведение base в степень degree
{
   int p = 1;
   while(degree-- > 0)           // Цикл выполняется degree раз
      p *= base;
   return p;
}
```

Здесь в заголовке цикла while текущее значение degree сравнивается с нулем и затем уменьшается на 1. Однако изменение параметра degree внутри функции никак не повлияет на фактический аргумент. Инструкция

```
cout << power(3, 2);
```

выведет 9. Здесь начальное значение локальной переменной degree равно 2.

Использование параметров функции как локальных переменных позволяет обойтись меньшим числом переменных. Так во втором варианте функции power() не создается переменная i.

5.4. Прототипы функций

Прототипом называется предварительное объявление функции. Прототип имеет вид:

```
тип-результата имя-функции(объявления аргументов);
```

то есть состоит из заголовка функции и точки с запятой. Например, для функции возведения в степень прототип имеет вид:

```
int power(int base, int degree);          // Прототип функции power()
```

Имена аргументов в объявлении (прототипе) функции можно не указывать, так как важен лишь их тип:

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

После объявления функцию можно вызывать из других функций, расположенных ниже по тексту программы, причем, вызовы должны быть согласованы с прототипом. Компилятор проверяет соответствие типов фактических и формальных параметров, и, если они не согласованы, выдает ошибку. Если обращение к функции соответствует ее прототипу, будет сгенерирован правильный код вызова функции, так как

объявление сообщает компилятору все необходимое: имя функции, количество и типы аргументов, тип возвращаемого значения.

Прототипы полезны при разработке больших программ, так как позволяют быстро набросать скелет программы, отложив реализацию деталей.

Компилятору достаточно объявления функции, но редактор связей должен иметь код функции, поэтому объявленная функция должна быть где-то определена.

Программа 5.5. Вычисление степени с контролем

В программе функция для вычисления целой степени целого числа сначала объявляется, а определяется в конце программы. Добавлена проверка, чтобы показатель степени не был отрицательным.

```
// Файл PowerIntContr.cpp

// Прототип (предварительное объявление)
// функции power() для вычисления целой степени целого числа

int power(int, int);

#include <iostream>
#include <locale>
#include <cstdlib>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    cout << "2 в степ. 10 = " << power(2, 10) << endl;
    cout << "2 в степ. -1 = ";
    cout << power(2, -1) << endl;
    system("pause");
    return 0;
}

// Определение функции power()
// base - обснование степени, degree - показатель степени

int power(int base, int degree)
{
    if(degree < 0){
        cerr << "\nОтрицательный показатель недопустим\n";
        system("pause");
        exit(1);
    }
    int p = 1;
    while(degree-- > 0)      // base возводится
        p *= base;           // в степень degree
    return p;                // Возвращение вычисленной степени
}
```

Программа выводит:

```
2 в степ. 10 = 1024
2 в степ. -1 =
Отрицательный показатель недопустим
```

5.5. Рекурсия

В языке C++ функции могут быть *рекурсивными*, то есть вызывать сами себя.

Возможность использования рекурсии можно проиллюстрировать на примере вычисления факториала неотрицательного целого числа:

$$n! = \begin{cases} 1, \text{ если } n = 0, \\ 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n, \text{ если } n > 0. \end{cases}$$

Эта формула показывает, как, зная значение факториала для меньшего числа $n-1$, вычислить факториал для следующего числа n, что позволяет сводить задачу к вычислению факториала все более меньшего числа. Также задан предельный случай $n = 0$.

Программа 5.6. Вычисление факториала

```
// Файл Factorial.cpp

// factorial: возвращает факториал аргумента n

int factorial(int n)
{
    int f;                                  // Значение факториала
    if (n == 0)
        f = 1;
    else
        f = factorial(n - 1) * n;// Рекурсивный вызов функции factorial()
    return f;
}

#include <iostream>
#include <locale>
#include <cstdlib>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    int k;
    do{
        cout << "Введите целое число >= 0: ";
        cin >> k;
    }while(k < 0);
    int fct = factorial(k);
```

```
    cout << "factorial(" << k << ") = " << fct << endl;
    system("pause");
    return 0;
}
```

Ниже приводится пример работы программы.

```
Введите целое число >= 0: 5
factorial(5) = 120
```

При каждом вызове рекурсивной функции создается новый набор локальных переменных, в том числе и формальных параметров. Сам же код рекурсивной функции существует в единственном экземпляре. Таким образом, рекурсивная функция, не закончив обработку одного набора данных, переключается на обработку другого, но затем возвращается и завершает прерванную обработку.

Полезно выполнить данную программу в пошаговом режиме, наблюдая за работой рекурсивной функции, за значениями формального параметра `n` и локальной переменной `f`, которые создаются заново при каждом новом вызове функции. Значения этих переменных при углублении в рекурсию и выходе из нее приведены в табл.13. Знаком вопроса (?) обозначено неопределенное значение переменной.

Таблица 13. Работа рекурсивной функции

Номер вызова	Углубление в рекурсию		Возврат из рекурсии	
	`n`	`f`	`n`	`f`
1	5	?	5	24 · 5 = 120
2	4	?	4	6 · 4 = 24
3	3	?	3	2 · 3 = 6
4	2	?	2	1 · 2 = 2
5	1	?	1	1 · 1 = 1
6	0	1	0	1

Заметим, что использование в функции `factorial()` переменной `f` излишне. Она включена для того, чтобы сделать код более понятным. Без ее использования функцию `factorial()` можно записать в виде:

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return factorial(n - 1) * n; // Рекурсивный вызов factorial()
}
```

5.6. Перегруженные имена функций

Если функции делают примерно одинаковую работу над объектами разных типов, удобно давать им одинаковые имена. Использование одного имени для операции, выполняемой над различными типами, называется *перегрузкой*.

Пусть в программе требуется вычислять абсолютную величину переменных типа `int` и типа `float`. Для этого можно написать две функции с одинаковыми именами, что и сделано в следующей программе.

Программа 5.7. Перегрузка функций

```
// Файл OverloadFunctions.cpp

// module: возвращает абсолютную величину целого n
int module(int n)
{
   if(n < 0)
      return -n;
   return n;
}

// module: возвращает абсолютную величину плавающего x
float module(float x)
{
   if(x < 0)
      return -x;
   return x;
}

// double module(int n)           // Ошибка, отличие от int module(int n)
// {return n >= 0 ? n : -n;}      // только в типе возвращаемого значения

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
// Работаем с int
   cout << "module(-3) = " << module(-3) << endl;

// Работаем с double
// В следующей строке неоднозначность выбора перегруженной функции
//  cout << "module(-3.14) = " << module(-3.14) << endl;

// Явное преоразование типа для вызова перегруженной функции

   cout << "module(-3.14) = " << module(float(-3.14)) << endl;
   system("pause");
   return 0;
}
```

Программа выводит:

```
module(-3) = 3
module(-3.14) = 3.14
```

Компилятор различает функции с одинаковыми именами по количеству и типам аргументов. При выборе подходящей функции из нескольких перегруженных компилятор ищет ту, которая имеет наилучшее соответствие типов формальных и фактических параметров.

Для аргумента `-3`, имеющего тип `int`, есть подходящая функция `int module(int)`.

Константы с плавающей точкой имеют тип `double`, поэтому для числа `- 3.14` можно вызвать либо `module(int)`, либо `module(float)` с преобразованием аргумента типа `double` либо к `int`, либо к `float`. В этой ситуации возникает ошибка: неоднозначный вызов перегруженной функции. Неоднозначность устраняется путем явного преобразования типа: `module(float(-3.14))`.

5.7. Аргументы функций по умолчанию

У функций общего назначения часто больше аргументов, чем требуется в простых случаях. В программе 5.8 рассмотрена функция печати целого `val`. В качестве аргумента функции передается основание системы счисления `base`, в которой следует печатать целое, но предполагается, что в большинстве случаев целые будут печататься в виде десятичных чисел, поэтому значением по умолчанию для `base` указано 10.

Программа 5.8. Аргументы по умолчанию

Функция печати целого `print()` рекурсивная. Если основание системы счисления `base <= 10`, используются обычные цифры `0, 1, …, 9`. Если `base > 10`, то в качестве цифр используются заглавные латинские буквы `A, B, C, D, E, F, G, H,…` со значениями: `10, 11, 12, 13, 14, 15, 16, 17,…`

```
// Файл ArgDefaultPrnNumb.cpp

#include <iostream>
#include <cstdlib>
using namespace std;

// prn_numb: печать val в системе счисления с основанием base

void prn_numb(int val, int base = 10)     // 10 – значение для base
{                                         // по умолчанию
   if(val < 0){
      cout.put('-');                      // Вывод знака «минус»
      prn_numb(-val, base);               // Рекурсивный вызов print()
```

```
   }
   if(val / base > 0)                    // Если число многозначное,
      prn_numb(val / base, base);        // печатать старшие цифры
   int r = val % base;                   // Остаток от деления – значение
                                         // последней цифры
   if(r < 10)                            // Используем обычные цифры
      cout.put('0' + r);                 // Вывод обычной цифры
   else                                  // Используем буквы
      cout.put('A' + r - 10);
}

void main()
{
   prn_numb(31);             cout.put(' ');       // По умолчанию base = 10
   prn_numb(31, 10);         cout.put(' ');
   prn_numb(31, 16);         cout.put(' ');
   prn_numb(31, 2);          cout.put(' ');
   prn_numb(31, 8);          cout << endl;
   system("pause");
}
```

Программа выводит:

```
31 31 1F 11111 37
```

Символы цифр `'0'`, `'1'`, …, `'9'` занимают в кодовой таблице непрерывный участок и расположены в порядке возрастания их значений. Код цифры `'0'` равен 48, у цифры `'1'` код 49 и т.д., поэтому выражение `'0' + r` дает правильный код последней десятичной цифры числа `val`. Например, если `val = 321`, то при любом основании `base` остаток от деления

`r = val % base = 1` и `'0' + r = 48 + 1 = 49`,

что равно коду цифры `'1'`, и эту цифру изобразит на экране функция `put('0' + r)` в соответствии со значением своего аргумента.

Для формирования цифр, значения которых больше 9, использовано выражение:

`'A' + r – 10`.

Для `val = 31` и `base = 16` значение остатка `r = 31 % 16 = 15`. Поэтому

`'A' + r – 10 = 'A' + 15 – 10 = 'A' + 5 = 65 + 5 = 70 = 'F'`.

Аргументы по умолчанию можно задавать только в конце списка аргументов, например,

```
int f(int, int = 0, char = 0);           // Правильно
int g(int = 0, int = 0, char);           // Ошибка
```

Аргументы по умолчанию указывают один раз в объявлении функции. Если определение функции идет после объявления значения аргументов по умолчанию не повторяются:

```
int f(int i, int j, char c)       // Определение функции. Значения по
{…}                               // умолчанию для j и c заданы в объявлении
```

5.8. Ссылки

Ссылка (reference) является альтернативным именем объекта. При объявлении ссылки на переменную некоторого типа после имени типа ставится знак &, например:

```
void f()
{
   int i = 13;       // i – переменная целого типа
   int & r = i;      // r и i ссылаются на одно и то же целое
   int x = r;        // x = 13
   r = 29;           // i = 29
   r = x ;           // i = 13
}
```

Ссылку всегда надо инициализировать, чтобы она ссылалась на какой-нибудь объект, например,

```
int k   = 1;
int& r1 = k;     // Правильно, r1 инициализирована
int& r2;         // Ошибка, отсутствует инициализация
extern int& r3;  // Правильно, r3 инициализируется в другом месте
```

Ссылку можно передавать в функции в качестве аргумента и возвращать из функций в качестве результата, при этом аргумент, передаваемый в функцию по ссылке, можно изменить внутри функции.

Программа 5.9. Использование ссылок

В состав программы входит функция `swap()`, меняющая значения своих аргументов, передаваемых в нее по ссылке, и функция `max()`, возвращающая ссылку на максимальный из двух своих аргументов.

```
// References.cpp: использование ссылок

void swap(int& a, int& b)         // Обмен значений a и b
{
    int tmp = a;
    a = b;
    b = tmp;
}

int& max(int& a, int& b)      // Возвращает ссылку на максимальное из a и b
{
```

```
   return a > b ? a : b;
}

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    int x, y;
    cout << "Введите два числа: ";
    cin >> x >> y;
    cout << "x = " << x << ", y = " << y << endl;
    swap(x, y);
    cout << "После обмена: \n";
    cout << "x = " << x << ", y = " << y << endl;
    max(x, y) = 0;
    cout << "Максимальное обнулили: \n";
    cout << "x = " << x << ", y = " << y << endl;
    system("pause");
    return 0;
}
```

Пример работы программы:

```
Введите два числа: 12 36
x = 12, y = 36
После обмена:
x = 36, y = 12
Максимальное обнулили:
x = 0, y = 12
```

При вызове функции `swap()` ее параметры-ссылки `a` и `b` инициализируются именами переменных `x` и `y`, поэтому `a` и `b` становятся другими *именами* для переменных `x` и `y`, что и позволяет изменить `x` и `y` внутри функции.

Благодаря тому, что `max()` возвращает *ссылку* на свой максимальный аргумент, `max()` может стоять в левой части оператора присваивания, благодаря чему максимальному аргументу присваивается новое значение.