9. Функции

Кроме главной (main) и библиотечных (системных) функций в программах могут использоваться дополнительные функции, написанные самим пользователем и предназначенные для решения вспомогательных подзадач. Код программы при этом становится более структурированным и читабельным.

Как программный объект, функция — это именованная последовательность операторов, выполняющих какое-либо логически законченное действие. Функция может принимать параметры (аргументы) и возвращать значение определённого типа. Функция прописывается только один раз, а вызывать её выполнение можно многократно, из разных точек программы.

Любая функция должны быть **объявлена** и **описана**. Объявлений может быть несколько, а описание только одно. Объявление функции должно находиться в тексте программы раньше её первого вызова (принцип опережающего объявления). Если до первого вызова находится полное описание функции, то специального её объявления не требуется.

**Внимание!** В языке C/C++ нельзя определять одну функцию внутри другой.

Функция состоит из **заголовка** и **тела**. **Объявить функцию** — это значит задать её заголовок (прототип, сигнатуру), завершающийся точкой с запятой. В заголовке указывается тип возвращаемого значения, имя функции и затем в круглых скобках через запятую список передаваемых аргументов (формальных параметров) с указанием их типов (типы, как обычно, предваряют имена). В случае, когда аргументы отсутствуют, скобки оставляются пустыми. При объявлении функции имена параметров можно не указывать, достаточно назвать их типы. **Описание функции** содержит, кроме заголовка (с обязательными именами параметров), тело функции, представляющее собой последовательность операторов (в фигурных скобках), которые должна произвести функция при её вызове.

**Возврат из функции** в точку вызова осуществляется оператором

return *выражение*;

При этом вычисляется значение *выражения* и передаётся в то место программы, откуда функция была вызвана. Тип *выражения* должен соответствовать типу, указанному в заголовке функции, или быть приводимым к нему. Оператор возврата return может присутствовать в функции несколько раз. Но функция может возвратить в точку вызова только одно значение.

*Замечание.* Тип возвращаемого значения может быть любым, кроме массива и функции. Но может быть указателем на массив или функцию.

Функции могут не возвращать никакого значения, а просто производить некоторые действия. В этом случае в заголовке указывается пустой тип возвращаемого значения void, а оператор return либо отсутствует, либо присутствует в одиночном виде, без *выражения*:

return;

**Вызов функции** в простейшем случае осуществляется путём указания её имени, за которым в круглых скобках перечисляются через запятую имена передаваемых аргументов. В случае, когда функция реально возвращает значение, запись её вызова и формирует это значение, которое может, в свою очередь, входить в состав других выражений (в частном случае — располагаться в правой части оператора присваивания). Если тип возвращаемого значения есть void, то функция вызывается как отдельный оператор, с точкой с запятой в конце.

При работе с функциями следует отличать **формальные и фактические параметры**. Формальные параметры указываются при описании функции, в её заголовке. Через них прописываются все действия, которые должна производить функция. При вызове функции указываются фактические параметры. Они подставляются компилятором на место соответствующих формальных параметров (с соблюдением очерёдности и с контролем соответствия типов), и с ними при вызове производятся те же действия, которые были прописаны для формальных параметров. Фактический параметр может быть константой, одиночной переменной или выражением. На имена фактических параметров ограничения по соответствию не накладываются.

**Пример 1.** Объявление, описание и использование простейшей функции, вычисляющей сумму двух вещественных аргументов.

#include <iostream>

using namespace std;

float sum(float,float); //объявление функции

int main() {

float a=5, z, t;

z=sum(a,7); //вызов функции (z=a+7)

cout<<”задайте t\n”;

cin>>t;

cout<<”z+t=”<<sum(z,t)<<endl; //вызов ф-ии

return 0; }

float sum(float x, float y) { //описание

return x+y; } //функции

Вариант этих же действий, но без специального объявления функции:

#include <iostream>

using namespace std;

float sum(float x, float y) { //описание

return x+y; } //функции

int main() {

float a=5, z, t;

z=sum(a,7); //вызов функции (z=a+7)

cout<<”задайте t\n”;

cin>>t;

cout<<”z+t=”<<sum(z,t)<<endl; //вызов ф-ии

return 0; }

В заголовке функции могут быть указаны **значения параметров по умолчанию**. Они пишутся через знак равенства после имени соответствующего параметра. В списке параметров они должны стоять последними. При вызове функции с неполным комплектом фактических параметров недостающим параметрам автоматически присваиваются значения по умолчанию. Пусть, к примеру, функция sum из Примера 1 описана в следующем виде:

float sum(float x=3, float y=4) {

return x+y; }

Тогда вызов sum(10,20) возвратит значение 30, вызов sum(10) возвратит 14, вызов sum() возвратит 7.

При работе со многими функциями всегда следует учитывать такое понятие, как **область видимости**. Параметры функции, а также все объекты, описанные внутри неё являются локальными и не видны за пределами функции. Этот факт допускает возможность существования в разных функциях одноимённых величин, но с разным смыслом. В частности, программа из Примера 1 может быть записана в таком варианте:

#include <iostream>

using namespace std;

float sum(float,float);

int main() {

float x, y=5, z;

z=sum(y,7);

cout<<”задайте x\n”;

cin>>x;

cout<<”z+x=”<<sum(z,x)<<endl;

return 0; }

float sum(float x, float y) {

return x+y; }

Здесь в функции main t переименовано на x, a — на y. Не следует смешивать x и y из main с x и y из функции sum.

Заметим, что в любой программе на C/C++ кроме функций имеется ещё **глобальный уровень** — вне всех функций. На этом уровне, в частности, в программе из Примера 1 стоит операция

using namespace std;

Величины, описанные на глобальном уровне, видны во всех функциях. Исключение составляет ситуация, когда внутри функции имеется одноимённая локальная величина, тогда видна именно эта локальная величина. Тем не менее, обращение к глобальной величине в этом случае возможно, но тогда перед её именем требуется ставить знак «двойное двоеточие». Вот коротенький пример без номера, с глобальной и локальной g:

#include <iostream>

using namespace std;

int g=200;

int main() {

int g=9;

cout<<g+::g<<endl; //напечатается 209

}

Однако при работе с функциями глобальных величин без особой необходимости рекомендуется избегать. Следует придерживаться общего принципа при написании программ: **функции должны быть максимально независимыми**.

***Передача параметров функции по значению, по адресу, по ссылке.***

При вызове функции фактические параметры **передаются** в неё **по значению**. Это означает, что в стек заносятся копии значений фактических параметров, и операторы функции работают с этими копиями. Доступа к самим ячейкам памяти фактических параметров у функции нет, следовательно, если с параметрами происходят какие-то изменения в процессе работы функции, то на исходных значениях параметров (в точке вызова) это никак не отражается. Однако если типом параметра является указатель (т. е. адрес ячейки памяти), то функция работает с этим адресом, и все изменения величины, размещённой по этому адресу, естественно, переходят в точку вызова. Такая передача параметров называется **передачей по адресу**. Коротенький пример:

#include <iostream>

using namespace std;

void f(int x, int \*y);

int main() {

int a=20, b=30;

f(a,&b);

cout<<”a=”<<a<<” b=”<<b<<endl;

// напечатается a=20 b=31

return 0; }

void f(int x, int \*y) {

x++; (\*y)++; }

Первый параметр передаётся по значению, и его изменение внутри функции f не повлияло на исходное значение. Второй параметр передаётся по адресу, поэтому его изменение отразилось на исходном значении. Однако его использование выглядит более сложным, поскольку требуется операция разыменовывания и взятия адреса. Этого можно избежать, если использовать технологию **передачи параметров по ссылке**.

Формально в языке C/C++ **ссылкой** называется альтернативное имя уже существующей величины. Ссылка объявляется с помощью знака & и при объявлении должна явно инициализироваться указанием величины, на которую она ссылается:

*тип* &*имя\_ссылки=имя\_величины*;

Здесь *тип* — это тип величины, на которую указывает ссылка. Значением ссылки становится адрес этой величины, однако при обращении к величине через ссылку, этот адрес неявно разыменовывается. Благодаря этому свойству, ссылки очень удобно использовать в качестве параметров функций, для которых требуется фиксировать все их обновления. При передаче параметра по ссылке в функцию передаётся его адрес, но при использовании этого параметра разадресации не требуется. Вот вариант предыдущего примера, с передачей второго параметра по ссылке:

#include <iostream>

using namespace std;

void f(int x, int &y);

int main() {

int a=20, b=30;

f(a,b);

cout<<”a=”<<a<<” b=”<<b<<endl;

// напечатается a=20 b=31

return 0; }

void f(int x, int \*y) {

x++; y++; }

Дополнительно, этот пример демонстрирует использование **функции типа void**, которая результаты своей работы не возвращает, а записывает в **выходные параметры** (которые, естественно, передаваться должны по адресу или по ссылке). Подобные функции используются, в частности, в тех случаях, когда результат работы функции представляет не одну, а несколько величин.

**Пример 2.** Функция типа void, вычисляющее для величин *x* и *y* их среднее арифметическое 0,5∙(*x*+*y*) и среднее геометрическое .

#include <iostream>

#include <math.h>

using namespace std;

void sr(float x, float y,

float &sr\_ar, float &sr\_geom) {

sr\_ar=0.5\*(x+y);

sr\_geom=sqrt(x\*y); }

int main() {

float a=9, b=25, SA, SG;

sr(a,b,SA,SG);

cout<<SA<<” ”<<SG<<endl;

return 0; }

***Передача массивов в качестве параметров.***

Если параметром функции является массив, то он должен передаваться по адресу, т. е. передаётся указатель на его начало (в качестве которого удобнее всего использовать наименование массива). Однако при этом теряется информация о его количестве элементов. Это означает, что размер массива следует передавать через отдельный параметр. Для двумерных массивов в качестве отдельных параметров следует передавать оба размера.

**Пример 3.** Функция, вычисляющая сумму элементов вещественного массива *x* размера *n*.

#include <iostream>

using namespace std;

void SUM(int n, float \*x) {

float s=0;

for (int i=0;i<n;i++) s+=x[i];

return s; }

int main() {

int n;

cout<<”задайте n\n”;

cin>>n;

float \*a=new float[n];

cout<<”задайте ”<<n<<” эл-тов массива\n”;

for (int i=0;i<n;i++) cin>>a[i];

float S;

S=sum(n,a);

cout<<”сумма=”<<S<<endl;

return 0; }

***Возвращение указателя в качестве результата.***

Указатель — это полноценный тип, поэтому вполне допустимо возвращать указатель как результат работы функции. Это удобно делать, когда этим результатом является комбинированный объект, например, массив.

**Пример 4.** Функция, формирующая массив *x* из *n* элементов по правилу:  и возвращающая указатель на этот массив.

Поскольку массив формируется внутри функции, выделение для него памяти должно происходить тоже внутри неё.

#include <iostream>

using namespace std;

float\* MASS(int n) {

float \*x=new float[n];

for (int k=0;k<n;k++) x[k]=1.0/(k\*k+1);

return x; }

int main() {

int n;

cout<<”задайте n\n”;

cin>>n;

float \*x=MASS(n);;

for (int i=0;i<n;i++) cout<<x[i]<<endl;

return 0; }

***Указатели на функции. Передача имён функций в качестве параметров.***

**Указатель на функцию** содержит адрес, по которому в памяти располагается исполняемый программный код функции. По умолчанию именно на этот адрес передаётся управление из точки вызова функции (обратная переадресация на точку вызова производится оператором return). Тип указателя на функцию определяется типом аргументов и типом возвращаемого значения. Например, объявление

float (\*fun)(int,double);

определяет указатель с именем fun на функцию, возвращающую значение типа float и имеющую два аргумента: первый типа int, второй типа float. Указатели на функции используются для косвенного вызова функции (не через её имя, а через адрес её программного кода), а также для передачи имени функции в другую функцию в качестве параметра. Вызов функции через указатель записывается так:

(\**имя\_указателя*)(*список\_фактических\_параметров*);

Для иллюстрации вспомним функцию sum (из Примера 1), вычисляющую сумму двух вещественных аргументов:

...

float z;

float sum(float x, float y) {

return x+y; }

float (\*psum)(float,float); //указатель на

... //функцию

psum=&sum; //указателю присваивается

... //адрес функции

z=(\*psum)(10,20); //функция вызывается

... //через адрес

В двух последних приведённых операторах допустима также запись:

psum=sum; и z=psum(10,20);

При желании можно с помощью оператора typedef определить тип «указатель на функцию»:

...

float z;

float sum(float x, float y) {

return x+y; }

typedef float (\*PS)(float,float);// PS - тип

... //«указатель на функцию»

PS psum=&sum; //определяется указатель на

... //функцию с инициализацией

z=(\*psum)(10,20);

...

Если функция является параметром другой функции, то передаётся она в ту функцию через указатель — тем же способом, как и параметры других типов. Однако чтобы избежать громоздких записей, целесообразно здесь использовать оператор typedef.

**Пример 5.** Функция *S*, определённая для двух вещественных массивов *x* и *y* размера *n* и вычисляющее среднее арифметическое величин *N*(*xi*, *yi*), где .

Обращаем внимание на то, что в программе n и N, а также s и S — это разные объекты (язык регистрозависимый).

#include <iostream>

using namespace std;

#include <math.h>

double N(double t, double u) {

return sqrt(t\*t+u\*u); }

typedef double (\*PFUN)(double,double);

double S(int n, double \*x, double \*y, PFUN PN)

{double s=0;

for(int i=0;i<n;i++) s+=PN(x[i],y[i]);

return s/n; }

int main() {

int n;

cout<<”задайте n\n”;

cin>>n;

double \*x=new double [n];

double \*y=new double [n];

cout<<”задайте ”<<n<<” эл-тов масс. x\n”;

for (int i=0;i<n;i++) cin>>x[i];

cout<<”задайте ”<<n<<” эл-тов масс. y\n”;

for (int i=0;i<n;i++) cin>>y[i];

PFUN PN=&N;

double sr\_ar=S(n,x,y,PN);

cout<<”sr\_ar=”<<sr\_ar<<endl;

return 0; }

**Задачи**

1. Даны три точки на плоскости. Определить, какая из них ближе к началу координат. Вычисление расстояния от точки до начала координат оформить как функцию.
2. Определить функцию, вычисляющую площадь треугольника по трём сторонам (формуле Герона, см. задание 4 к теме 4), с предварительной поверкой условия существования треугольника. С помощью этой функции вычислить площадь фигуры

1

*a*

*b*

*c*

*d*

2

2,5

(*a*, *b*, *c*, *d* – даны).

1. Определить функцию, вычисляющую площадь треугольника по трём сторонам и написать программу для определения площади треугольника, заданного координа­тами вершин на плоскости.
2. Определить две функции max и min, вычисляющие, соответственно, максимум и минимум из двух величин. С их помощью при заданных *a*, *b*, *c* вычислить величину



1. Даны *x*, *y*, *n*. Вычислить величину



где . Вычисление *f*(*x*, *n*) оформить как функцию.

1. Даны: *n*, массивы *x* и *y* по *n* элементов. Определить функцию *S*, вычисляющую их скалярное произведение . С помощью этой функции вычислить величину



Все три суммы должны быть вычислены через функцию *S*.

1. Написать функцию определения центра и радиуса окружности, проходящей через три заданные точки на плоскости (если это невозможно – выдать соответствующее сообщение).
2. Написать функцию типа void, вычисляющую наибольший и наименьший элемент массива *x* из *n* элементов. Применить её для конкретного массива.
3. Определить функцию, вычисляющую сумму *k*‑й строки матрицы *a*(*m*×*n*). С ее помощью вычислить сумму строки с номером 3 заданной матрицы размера 5×7.
4. Логической переменной *t* присвоить значение true, если уравнения *x*2+6,2*x*+*a*2=0 и *x*2+*ax*+*b*–1=0 имеют вещественные корни, и при этом оба корня первого уравнения лежат между корнями второго. И присвоить *t* значение false во всех остальных случаях. Вычисление дискриминанта «абстрактного» квадратного уравнения *ax*2+*bx*+*c*=0 оформить как функцию, нахождение вещественных корней — как другую функцию.
5. Написать функцию, формирующую массив размера *n*, состоящий из идущих подряд простых чисел: 2, 3, 5, 7, 11… и возвращающую указатель на этот массив.

Написать функцию, формирующую массив *z* размера *n* по правилу: *zi*=*f*(*i*), где 