Глава 4
Функции

Использование функций при создании программ существенно упрощает структуру программного кода и позволяет решать ряд принципиальных задач. Само понятие функции в C++, как и в большинстве других языков программирования, достаточно точно соответствует математическому определению функции, хотя некоторые особенности, безусловно, существуют. Частично они связаны с тем, что это функции программные, некоторая специфика также связана с языком C++.

Под функцией подразумевают именованный программный код, который может многократно вызываться в программе. Функции реализуются отдельными программными блоками, могут иметь аргументы и возвращать значения в качестве результата (а могут и не возвращать, результатом функции может быть практически все, кроме массива).

Далее речь пойдет о том, как функции объявляются и используются.

Объявление и использование функций

Прибегните к совету Конька-Горбунка возьмите книжку и прочитайте – там все очень доступно написано

Из к/ф «Чародеи»

Формат объявления функции в C++ имеет такую структуру: указывается тип значения, которое возвращает функция, название функции, в круглых скобках список аргументов (с указанием типа аргументов). Программный код функции указывается в блоке из фигурных скобок:

```
тип_результата имя_функции(тип аргумент1, тип аргумент2, ...){
   код функции
   }
```

В листинге 4.1 приведен пример программы с объявлением функции `msum()`, которая возвращает в качестве результата значение суммы натуральных чисел. Количество слагаемых в сумме определяется целочисленным аргументом функции.

Листинг 4.1. Код программы с функцией для вычисления суммы натуральных чисел

```
#include<iostream>
using namespace std;
int msum(int n){
   int s=0;
   for(int i=1;i<=n;i++) s+=i;
   return s;}
int main(){
   int n;
   cout<<"Enter n = ";
   cin>>n;
   cout<<"Sum is "<<msum(n)<<"\n";
   return 0;
   }
```

Основу кода функции составляет оператор цикла `for(int i=1;i<=n;i++) s+=i` Индексная переменная `i` пробегает с единичным шагом дискретности значения от `1` до `n` (аргумент функции) На каждом итерационном шаге переменная `s`, инициализированная с нулевым начальным значением, увеличивается на `i` (команда `s+=i`) Значение именно этой переменной `s` возвращается в виде результата функции (команда `return s`) В основном методе `main()` программы созданная функция `msum()` вызывается в команде `cout<<"Sum is "<<msum(n)<<"\n"` для вычисления суммы натуральных чисел Предварительно значение целочисленной переменной `n` вводится пользователем с клавиатуры

Функция не всегда должна возвращать результат Если функция результат не возвращает, в качестве типа функции указывается `void` В этом случае функция напоминает процедуру Пример такой функции приведен в листинге 4 2

Листинг 4.2. Функция не возвращает результат

```
#include<iostream>
using namespace std;
void msum2(){
   int n,
   cout<<"Enter n = ";
   cin>>n;
   int s=0,
   for(int i=1;i<=n;i++) s+=i;
   cout<<"Sum is "<<s<<"\n";
   }
int main(){
```

```
msum2();
return 0;
}
```

Более того, функция не только не возвращает результат – у нее еще и нет аргументов (тем не менее, скобки после названия функции все равно указываются, причем как при объявлении функции, так и при ее вызове).

Функциональность программы после внесения исправлений не изменилась: на экране выводится приглашения для пользователя ввести натуральное число (верхнюю границу для суммы натуральных чисел), это число считывается, после чего вычисляется сумма натуральных чисел. Что изменилось, так это организация программного кода. Фактически главный метод программы состоит всего из одной команды (если не считать стандартную инструкцию `return 0`) по вызову функции `msum2()`. Вся функциональность программы спрятана в этой функции. Вначале в рамках вызова функции `msum2()` запрашивается значение целочисленной переменной `n` (теперь это локальная переменная функции). Затем вводится переменная `s` для записи значения суммы натуральных чисел, выполняется оператор цикла и на экран выводится значение суммы (переменная `s`). На этом выполнение функции заканчивается.

Все переменные, использованные в теле функции, являются локальными. Это означает, что они доступны только в теле функции, но никак не за ее пределами. Вообще же доступность переменных определяется простым правилом: переменные доступны в пределах того блока, где они объявлены. Напомним, что блок в C++ ограничивается парой фигурных скобок.

Несмотря на то что функция, объявленная с типом `void`, результат не возвращает, она может содержать инструкцию `return`, и не одну. В этом случае инструкция `return` является командой завершения функции. Пример такой функции приведен в листинге 4.3.

Листинг 4.3. Функция с инструкцией `return` не возвращает результат

```
void InvFunc(double z){
   if(z==0){
      cout<<"Division by zero!"<<endl;
      return;}
   double x;
   x=1/z;
   cout<<"1/z ="<<x<<endl;
   }
```

У функции `InvFunc()` один аргумент типа `double`, а в качестве типа возвращаемого результата указано `void` (функция результат не возвращает).

В результате выполнения функции, в зависимости от значения аргумента, возможны два сценария. При ненулевом аргументе на экран выводится значение, обратное к аргументу. Если аргумент у функции нулевой, выводится сообщение соответствующего содержания.

Проверка аргумента на предмет равенства нулю осуществляется в условном операторе. Блок условного оператора, выполняющийся при нулевом аргументе, состоит из двух команд: выводится сообщение (команда `cout<<"Division by zero!"<<endl`) и затем завершается выполнение функции (команда `return`).

Если аргумент ненулевой, объявляется переменная `x` типа `double`. В качестве значения ей присваивается величина, обратная к аргументу функции, после чего полученное значение выводится на экран.

В обеих командах вывода в функции использована инструкция завершения строки `endl`. Кроме того, в данном случае фактически инструкция `return` использована как альтернатива к `else`-блоку условного оператора: можно было бы ту часть кода, что выполняется при ненулевом аргументе функции, разместить в `else`-блоке условного оператора, отказавшись при этом от инструкции завершения функции `return`. Тем не менее, в более сложных программах использование такой инструкции часто бывает не только оправданным, но и существенно упрощает структуру кода.

Что касается непосредственно объявления функции, то в C++ допускается в начале программы указывать прототип функции (прототипом функции называется «шапка» с типом возвращаемого результата, именем функции и списком аргументов), а непосредственно объявление функции переносить в конец программы. Корректным является такой программный код (листинг 4.4).

Листинг 4.4. Функция объявлена в конце программы

```
#include <iostream>
using namespace std;
//Прототип функции InvFunc():
void InvFunc(double z);
//Функция InvFunc() использована в программе:
int main(){
   double s;
   cout<<" Enter number: ";
   cin>>s;
   InvFunc(s);
   return 0;
   }
//Объявление функции InvFunc():
```

```
void InvFunc(double z){
   if(z==0){
      cout<<"Division by zero!"<<endl;
      return;}
   double x;
   x=1/z;
   cout<<"1/z ="<<x<<endl;
   }
```

По большому счету, описывать функцию можно где угодно – главное, чтобы ее прототип был указан до ее первого использования.

Если описание функции и ее прототип разнесены в программе, в прототипе при перечислении аргументов их формальные названия указывать не обязательно – достаточно указать только тип аргумента. Например, вместо прототипа `void InvFunc(double z)` в листинге 6.4 можно было указать прототип `void InvFunc(double)`. Так же поступают в случае, когда у функции несколько аргументов: в списке аргументов перечисляют через запятую только их типы, без названий.

Механизмы передачи аргументов

В C++ существует два механизма передачи аргументов функциям: по значению и через ссылку. При передаче аргумента функции по значению при вызове функции для переменных, которые указаны ее аргументами, создаются копии, которые фактически и передаются функции. После завершения выполнения кода функции эти копии уничтожаются (выгружаются из памяти). При передаче аргументов функции по ссылке функция получает непосредственный доступ (через ссылку) к переменным, указанным аргументами функции. **С практической точки зрения разница между этими механизмами заключается в том, что при передаче аргументов по значению изменить передаваемые функции аргументы в теле самой функции нельзя, а при передаче аргументов по ссылке – можно. По умолчанию используется механизм передачи аргументов функции по значению**.

Проиллюстрируем сказанное на конкретном примере. Рассмотрим программный код, приведенный в листинге 4.5.

Листинг 4.5. Передача аргумента по значению

```
#include <iostream>
using namespace std;
//Аргумент передается по значению:
int incr(int m){
```

```
   m=m+1;
   return m;
   }
int main(){
   int n=5;
   cout<<"n ="<<incr(n)<<endl;
   cout<<"n ="<<n<<endl;
   return 0;
   }
```

Программный код функции `incr()` предельно прост: функция в качестве значения возвращает целочисленную величину, на единицу превышающую аргумент функции. Особенность программного кода состоит в том, что в теле функции выполняется команда `m=m+1`, которой с формальной точки зрения аргумент функции `m` увеличивается на единицу, и именно это значение возвращается в качестве результата. Поэтому с точки зрения здравой логики после вызова функции переменная, переданная ей в качестве аргумента, должна увеличиться на единицу. Однако это не так. Вывод результатов выполнения программы на экран будет выглядеть так:

```
n =6
n =5
```

Если с первой строкой проблем не возникает, то вторая выглядит несколько неожиданно. Хотя это только на первый взгляд. Остановимся подробнее на том, что происходит при выполнении программы.

В начале главного метода `main()` инициализируется со значением `5` целочисленная переменная `n`. Далее на экран выводится результат вычисления выражения `incr(n)` и затем значение переменной `n`. Поскольку функция `incr()` возвращает значение, на единицу большее аргумента, результат этого выражения равен `6`. Но почему же тогда не меняется переменная `n`? Все дело в механизме передачи аргумента функции. Поскольку аргумент передается по значению, при выполнении инструкции `incr(n)` для переменной `n` автоматически создается копия, которая и передается аргументом функции `incr()`. В соответствии с кодом функции значение переменной-копии увеличивается на единицу и полученное значение возвращается в качестве результата функции. Как только результат функцией возвращен, переменная-копия прекращает свое существование. Поэтому функция вычисляется корректно, а переменная-аргумент не меняется!

Для того чтобы аргумент передавался не по значению, а по ссылке, перед именем соответствующего аргумента необходимо указать оператор `&`. В листинге 4.6 приведен практически тот же программный код, что и в листинге 4.5, однако аргумент функции `incr()` в этом случае передается по ссылке.

Листинг 4.6. Передача аргумента по ссылке

```
#include <iostream>
using namespace std;
//Аргумент передается по ссылке:
int incr(int &m){
   m=m+1;
   return m;
   }
int main(){
   int n=5;
   cout<<"n ="<<incr(n)<<endl;
   cout<<"n ="<<n<<endl;
   return 0;
   }
```

Результат выполнения программы выглядит так:

```
n =6
n =6
```

Поскольку аргумент функции передается по ссылке, то все манипуляции в теле функции выполняются не с копией аргумента, а непосредственно с аргументом. Таким образом, вызов функции `incr(n)` не только возвращает в качестве результата увеличенное на единицу значение аргумента, но приводит к тому, что этот аргумент действительно увеличивается на единицу.

Если у функции несколько аргументов, часть из них (или все) могут передаваться по ссылке, а часть – по значению. Пример приведен в листинге 4.7.

Листинг 4.7. Разные механизмы передачи аргументов

```
#include <iostream>
using namespace std;
//Аргумент передается по ссылке и по значению:
void change(int &m, int n){
   int k;
   k=n;
   n=m;
   m=k;
   cout<<"m ="<<m<<endl;
   cout<<"n ="<<n<<endl;
   }
int main(){
   int m=3,n=5;
   change(m,n);
   cout<<"m ="<<m<<endl;
   cout<<"n ="<<n<<endl;
   return 0;
   }
```

В результате выполнения программы на экране появятся следующие строки:

```
m =5
n =3
m =5
n =5
```

У функции `change()` два целочисленных аргумента: первый передается по ссылке, а второй передается по значению. Действие функции состоит в том, что значения аргументов меняются местами: первому аргументу присваивается значение второго аргумента и наоборот. Но поскольку по ссылке передается только первый аргумент, то изменяется значение только этого аргумента – второй остается неизменным.

Обращаем внимание читателя на то, что наличие оператора & перед именем аргумента – всего лишь инструкция для определения механизма передачи аргумента. Это не влияет на способ обращения к аргументу или использования аргумента в теле функции. В этом смысле передача аргумента по ссылке отличается от передачи аргументом функции указателя на переменную. Об этом речь пойдет в следующем разделе.

Передача указателя аргументом функции

В качестве аргументов функции могут передаваться указатели. При передаче указателя аргументом функции перед именем указателя указывается оператор *. Пример передачи указателя аргументом функции приведен в листинге 4 8.

Листинг 4.8. Передача указателя аргументом функции

```
#include <iostream>
using namespace std;
//Аргументом является указатель:
int incr(int *m){
   *m=*m+1;
   return *m;
   }
int main(){
   int n=5;
   cout<<"n ="<<incr(&n)<<endl;
   cout<<"n ="<<n<<endl;
   return 0;
   }
```

Фактически представленный код является эквивалентом кода программы, представленной в листинге 4.6, однако вместо передачи аргумента по ссылке аргументом функции передается указатель. Прототип функции выглядит как `int incr(int *m)`. Теперь `m` является указателем на переменную целого типа. Чтобы получить доступ к значению этой переменной, используем инструкцию вида `*m`, что и было сделано в программе.

При вызове функции ее аргументом указывается не переменная `n`, а адрес этой переменной, который получаем с помощью инструкции `&n`.

Хочется сделать несколько замечаний относительно особенностей указателей как аргументов функции. Дело в том, что указатель, будучи переданным аргументом функции, передается, как обычные переменные, по значению. Убедиться в этом можно с помощью кода, представленного в листинге 4.9.

Листинг 4.9. Указатель передается аргументом функции по значению

```
#include <iostream>
using namespace std;
void test(int *n){
   cout<<"Address: "<<&n<<"\n";
   }
int main(){
   int n=1;
   int *p;
   p=&n;
   test(p);
   cout<<"Address: "<<&p<<"\n";
   return 0;
   }
```

Основу кода составляет функция `test()`, аргументом которой является целочисленный указатель (аргумент объявлен как `int *n`). В результате выполнения функции на экран выводится адрес аргумента функции (адрес получаем через инструкцию `&n`). Адрес аргумента – это указатель на указатель. Таким образом, при вызове функции на экране появляется сообщение с адресом переменной, которая реально обрабатывается функцией.

В главном методе программы с единичным значением инициализируется целочисленная переменная `n`. Кроме этого объявляется указатель `p` на целочисленную переменную и этой переменной в качестве значения присваивается адрес переменной `n`. Переменная-указатель `p` передается аргументом функции `test()` и, кроме этого, на экран выводится адрес переменной `p` (инструкция `&p`). Результат выполнения программы может иметь следующий вид:

```
Address: 0012FF28
Address: 0012FF78
```

Конкретные значения не важны, важно то, что они различны. Если бы в функцию передавалась не копия переменной-указателя, то в обоих случаях адреса бы совпадали. Таким образом, как и в случае с обычными переменными, указатель по умолчанию передается аргументом функции по значению. Тем не менее, он обеспечивает доступ из функции к исходной переменной, на которую ссылается, поскольку копия этого указателя ссылается на ту же самую переменную.

Наконец, чтобы передать аргумент-указатель по ссылке, а не по значению, можно в качестве прототипа функции указать `void test(int *&n)`.

Передача массива аргументом функции

Нередко в качестве аргументов функций указываются массивы. Учитывая то обстоятельство, что имя массива является ссылкой на первый его элемент, существует некоторая свобода в способе передачи массива аргументом функции. Хотя, если смотреть в корень проблемы, то все способы базируются на одном механизме. Тем не менее, рассмотрим все варианты. Первый и наиболее прямолинейный проиллюстрирован в листинге 4.10.

Листинг 4.10. Передача аргументом массива: явное указание размера

```
#include <iostream>
using namespace std;
//При объявлении массива явно указан размер:
void show(int n[5]){
   for(int i=0;i<5;i++)
   cout<<"n["<<i<<"]="<<n[i]<<endl;
   }
int main(){
   int n[5]={1,2,3,4,5};
   show(n);
   return 0;
   }
```

В программе объявляется функция `show()`, аргументом которой указан целочисленным массив из пяти элементов. Прототип функции имеет вид `void show(int n[5])`. Передаваемый массив указывается вместе с размером. В результате выполнения программы в столбик выводятся значения элементов массива:

```
n[0]=1
n[1]=2
n[2]=3
n[3]=4
n[4]=5
```

Ничего не изменится, если при объявлении функции размер в явном виде не указывать (листинг 4.11).

Листинг 4.11. Передача аргументом массива: размер не указан

```
#include <iostream>
using namespace std;
//При объявлении массива размер не указан:
void show(int n[]){
   for(int i=0;i<5;i++)
   cout<<"n["<<i<<"]="<<n[i]<<endl;
   }
int main(){
   int n[5]={1,2,3,4,5};
   show(n);
   return 0;
   }
```

Причина кроется в том, что при передаче массива аргументом и в том, и в другом случае на самом деле передается ссылка на первый элемент массива. Поэтому особого значения не имеет, указан размер массива или нет – в C++ все равно проверки на предмет выхода за пределы массива нет. Поэтому разумнее, принимая во внимание сказанное, передавать массив в виде указателя, благо таковым является имя массива. Пример соответствующего кода приведен в листинге 4.12.

Листинг 4.12. Передача аргументом массива в виде указателя

```
#include <iostream>
using namespace std;
//Аргументом указано имя массива и размер:
void show(int *n,int m){
   for(int i=0;i<m;i++)
   cout<<"n["<<i<<"]="<<n[i]<<endl;
   }
int main(){
   int n[5]={1,2,3,4,5};
   show(n,5);
   show(n,3);
   return 0;
   }
```

Прототип функции имеет вид `void show(int *n,int m)`. В соответствии с этим прототипом у функции два аргумента: целочисленный указатель (имя массива) и целое число (размер массива). В теле функции в операторе цикла верхняя граница изменения индексной переменной указана как `m`. В главном модуле функция `show()` вызывается дважды: первый раз в формате `show(n,5)` и `show(n,3)`. В результате получаем:

```
n[0]=1
n[1]=2
n[2]=3
n[3]=4
n[4]=5
n[0]=1
n[1]=2
n[2]=3
```

Здесь использовано свойство, что, во-первых, имя массива является указателем на первый его элемент, а во-вторых, указатели можно индексировать. Более того, изменяя второй аргумент, можем изменять количество обрабатываемых элементов массива (главное не выйти при этом за пределы массива).

Передача многомерных, и в частности двумерных, массивов аргументами функции осуществляется по следующему принципу: указываются все размеры массива, кроме первого. Такая явная индексация связана со способом интерпретации многомерных массивов в C++. Так, двумерный массив – это массив массивов. Поэтому, как и в случае одномерного массива, первый индекс является необязательным, поскольку функции на самом деле передается ссылка на первый элемент массива (для многомерных массивов этот первый элемент сам является массивом). Все последующие индексы нужны для того, чтобы компилятор мог корректно выделить место под элементы массива. Пример передачи двумерного массива аргументом функции приведен в листинге 4.13.

Листинг 4.13. Передача аргументом двумерного массива

```
#include <iostream>
using namespace std;
void show2(int n[][3],int size){
  int i,j;
  for(i=0;i<size;i++){
     for(j=0;j<3;j++){
            cout<<n[i][j]<<" ";}
     cout<<"\n";
     }
  }
```

```
int main(){
  int n[][3]={{1,2,3},
              {4,5,6}};
  show2(n,2);
  return 0;
  }
```

Функцией `show2()` выводится на экран двумерный массив. В результате на экране появится сообщение

```
1 2 3
4 5 6
```

Функции аргументом передается двумерный массив: инструкция `int n[][3]` в прототипе функции содержит явно указанный размер массива по второму индексу. Кроме двумерного массива, функции в качестве значения передается целочисленная переменная `size`, определяющая размер массива по первому индексу. Обращение к функции в главном методе осуществляется в формате `show2(n,2)`, где `n` – предварительно инициализированный двумерный массив.

Передача строки аргументом функции

Поскольку один из вариантов реализации текстовых строк подразумевает их представление в виде символьного массива, нетрудно догадаться, что текстовые строки могут передаваться аргументами функциям практически так же, как и прочие массивы. Хотя имеются и свои особенности. В основном они касаются способов обработки таких символьных массивов. Пример приведен в листинге 4.14.

Листинг 4.14. Передача аргументом символьного массива

```
#include <iostream>
using namespace std;
int length(char *str){
   for(int s=0;*str;s++,str++);
   return s;
   }
int main(){
   char str[20]="This is a string";
   cout<<"Length is "<<length(str)<<endl;
   return 0;
   }
```

Функцией `length()` для массива, указанного аргументом функции, подсчитывается количество символов. Для этого используется оператор цикла, в блоке инициализации которого с нулевым начальным значением инициализируется переменная `s`. Переменная необходима для подсчета количества символов. В качестве проверяемого условия указана инструкция `*str` (где `str` есть аргумент функции – указатель на данные типа `char`). В данном случае использовано то обстоятельство, что фактическая текстовая строка, представленная в виде символьного массива, заканчивается нуль-символом. Поэтому достижение конца строки эквивалентно тому, что значение соответствующего элемента массива равняется нулю. В блоке инкремента две команды: одной на единицу увеличивается значение переменной-счетчика `s`, а второй командой на единицу увеличивается значение указателя `str`. Напомним, что адресная арифметика имеет свои правила: увеличение указателя на единицу означает переход к следующей ячейке памяти. Поскольку элементы массива размещаются в памяти подряд один за другим, изменение указателя на единицу означает переход к следующему элементу массива. В качестве значения функции возвращается значение переменой `s`. Также обращаем внимание читателя на то обстоятельство, что размер массива аргументом функции не передается. В данном случае в этом нет необходимости в силу двух причин. Во-первых, сам по себе размер массива не очень актуален, поскольку речь идет о символьном массиве, в котором фактические данные занимают не все ячейки. Во-вторых, поскольку существует четкий признак окончания строки (имеется в виду нуль-символ, которым строка заканчивается), необходимости явно указывать размер строки нет.

Учитывая все вышеозначенное, легко предугадать результат выполнения программы: на экране появится сообщение `Length is 16`. Причем вычисляется именно длина строки-значения массива: в строке `"This is a string"` (с учетом пробелов) `16` символов, а массив `str` состоит из `20` элементов.

Аргументы функции `main()`

У главного метода программы `main()` могут быть аргументы: число параметров командной строки и массив с текстовыми значениями этих параметров. Обычно параметры называют `argc` (размер массива) и `argv` (символьный двумерный массив), однако это не обязательно. В листинге 4.15 приведен пример программы, в которой в столбик выводятся на экран все переданные в командной строке параметры функции. При этом первым и всегда присутствующим параметром является название запускаемой программы (полный путь к файлу).

Листинг 4.15. Аргументы главного метода программы

```
include <iostream>
using namespace std;
int main(int size,char *str[]){
   int i;
   for(i=0;i<size;i++)
      cout<<i+1<<"-th argument is: "<<str[i]<<endl;
   return 0;
   }
```

Первый целочисленный `size` аргумент особых вопросов не вызывает. Второй аргумент, объявленный как `char *str[]` – массив, элементами которого являются текстовые строки, реализованные в виде символьных массивов. Более подробно о работе с такими массивами речь будет идти в главе 5. В данном случае важно лишь то, что ссылка `str[i]` означает `i`-ю строку, т.е. текстовое значение `i`-го параметра командной строки. В приведенном программном коде индексная переменная `i` пробегает значения в соответствии с размером массива (от `0` до `size-1` включительно). Для каждого значения этого индекса на экран выводится соответствующий параметр командной строки, для чего используется ссылка `str[i]`. Например, если после компиляции файл программы называется `mytest.exe`, размещен в директории `C:\Program Files\MyProgs` и запускается на выполнение командой `mytest.exe Alexei Vasilev 2008`, в результате получим:

```
1-th argument is: C:\Program Files\MyProgs\mytest.exe
2-th argument is: Alexei
3-th argument is: Vasilev
4-th argument is: 2008
```

Если параметры в командную строку передавать не планируется, аргументы для метода `main()` можно не указывать (а можно указывать).

Аргументы по умолчанию

Курс у нас один – правильный

В.С. Черномырдин

Для аргументов функций можно указывать значения по умолчанию. **Если аргумент имеет значение по умолчанию, то в случае, если при вызове функции этот аргумент явно не указан, используется его значение по умолчанию.**

Чтобы задать аргументу значение по умолчанию, в списке аргументов функции после имени этого аргумента через знак равенства указывается соот-

ветствующее значение, т.е. синтаксис определения значения по умолчанию в прототипе функции следующий:

```
тип имя_функции(тип аргумент=значение){код функции}
```

Если у функции несколько аргументов, то значения по умолчанию можно задавать для любого их количества, в том числе и для всех При этом аргументы, которые имеют значения по умолчанию, должны следовать в списке аргументов функции последними.

В случае, когда прототип функции указывается до ее определения, значения по умолчанию указываются только в прототипе функции. В листинге 4.16 приведен пример использования функций с аргументами, имеющими значения по умолчанию

Листинг 4.16. Аргументы со значениями по умолчанию

```
#include <iostream>
using namespace std;
//Аргумент функции имеет значение по умолчанию:
void showX(int x=0){
   cout<<"x = "<<x<<endl;
   }
//Два аргумента функции в прототипе имеют значение по
//умолчанию,
//сама функция описана в конце программы:
void showXYZ(int x,int y=1,int z=2);
int main(){
   showX(3);
   showX();
   showXYZ(4,5,6);
   showXYZ(7,8);
   showXYZ(9);
   return 0;
   }
//При описании функции значения по умолчанию не указываются:
void showXYZ(int x,int y,int z){
   cout<<"x = "<<x<<" ";
   cout<<"y = "<<y<<" ";
   cout<<"z = "<<z<<endl;
   }
```

В программе объявлены две функции: у функции `showX()` один аргумент с целочисленным аргументом, имеющим значение по умолчанию, а у функции `showXYZ()` из трех целочисленных аргументов значения по умолчанию указаны для двух. Действие функций состоит в том, что выводятся значения аргументов. В главном методе приведены вызовы этих функций

с передачей разного числа аргументов: первая функция showX() вызывается с одним аргументом и без аргумента, а вторая функция showXYZ() вызывается с числом аргументов от одного до трех. Результат выполнения программы будет иметь вид

```
x=3
x=0
x=4 y=5 z=6
x=7 y=8 z=2
x=9 y=1 z=2
```

В прототипе функции, как отмечалось, формальные значения для аргументов могут не указываться (но они указываются в описании функции). Если аргументы имеют значения по умолчанию, знак равенства и значение для соответствующего аргумента в прототипе указывают сразу после идентификатора типа аргумента. Например, в приведенном листинге 6.16 вместо прототипа `void showXYZ(int x,int y=1,int z=2)` можно было использовать прототип `void showXYZ(int x,int=1,int=2)`. Функциональность программы от этого не изменится.

Возвращение функцией указателя

Функция в качестве значения может возвращать указатель. Главное, что для этого нужно сделать, – предусмотреть соответствующие инструкции в программном коде функции. В прототипе функции, возвращающей в качестве значения указатель, перед именем функции указывают оператор *. Пример функции, возвращающей указатель, приведен в листинге 4.17.

Листинг 4.17. Функция возвращает в качестве значения указатель

```
#include <iostream>
using namespace std;
int *mpoint(int &n,int &m){
   if(n>m) return &n;
   else return &m;
   }
int main(){
   int n=3,m=5;
   int *p;
   p=mpoint(n,m);
   (*p)++;
   cout<<"n ="<<n<<endl;
   cout<<"m ="<<m<<endl;
   return 0;
   }
```

В программе объявляется функция `mpoint()`, возвращающая в качестве результата указатель на максимальный из двух ее аргументов. Сразу отметим, что оба аргумента передаются по ссылке – если бы они передавались по значению, не было бы смысла возвращать указатель на один из аргументов, поскольку в этом случае указатель ссылался бы на временную переменную-копию аргумента.

При возвращении в качестве значения указателя не следует забывать, что возвращается адрес большего из аргументов, а не сам аргумент. Адрес переменной (аргумента) получаем, указав перед именем соответствующей переменной оператор `&`, что и было сделано в командах `return &n` и `return &m`.

В главном методе программы объявляются две целочисленные переменные `n` и `m` со значениями 3 и 5 соответственно. Именно они передаются аргументами функции `mpoint()`. Результат записывается в переменную-указатель `p`. В результате переменная `p` в качестве значения получает адрес переменной `m`. Поэтому после выполнения команды `(*p)++` значение переменной `m` увеличивается на единицу. Результатом выполнения программы станет пара строк

```
n =3
m =6
```

В конечном счете указатель – это всего лишь переменная, значением которой является адрес памяти. Нет ничего удивительного в том, что адрес возвращается функцией.

Возвращение функцией ссылки

В C++ функции могут возвращать в качестве результата ссылку на значение. Для того чтобы функция возвращала ссылку на значение, перед именем функции в ее прототипе (и при описании) необходимо использовать оператор `&`. Пример функции, результатом которой является ссылка, приведен в листинге 4.18.

Листинг 4.18. Функция возвращает в качестве значения ссылку

```
#include <iostream>
using namespace std;
int &mpoint(int &n,int &m){
   if(n>m) return n;
   else return m;
   }
int main(){
   int n=3,m=5;
```

```
int p;
mpoint(n,m)=2;
p=mpoint(n,m);
cout<<"n ="<<n<<endl;
cout<<"m ="<<m<<endl;
cout<<"p ="<<p<<endl;
return 0;
}
```

Приведенный программный код представляет собой модификацию предыдущего примера из листинга 4.17, только в данном случае функцией `mpoint()` возвращается не указатель, а ссылка на больший из своих аргументов. При объявлении функции перед названием указан оператор `&` (признак того, что функцией возвращается ссылка). В качестве значения, как отмечалось, возвращается больший из аргументов `n` и `m` (инструкции `return n` и `return m` в условном операторе). Однако в силу того, что функция объявлена как ссылка, в действительности возвращается не значение соответствующей переменной, а ссылка на нее! Драматичность этого утверждения легко подтвердить с помощью кода, представленного в главном методе программы.

В методе `main()`, как и в предыдущем примере, инициализируются две целочисленные переменные `n=3` и `m=5`, а также объявляется целочисленная переменная `p`. Далее следует на первый взгляд довольно странная команда `mpoint(n,m)=2`, после чего командой `p=mpoint(n,m)` присваивается значение переменной `p`. Но самое интересное – это результат, который получаем при выполнении этой программы:

```
n =3
m =2
p =3
```

По крайней мере, странно выглядят значения переменных `m` и `p`. Разумеется, это не совсем так, а точнее, совсем не так. Начнем с команды `mpoint(n,m)=2`. Чтобы понять смысл этой команды, вспомним, что результатом инструкции `mpoint(n,m)` является ссылка на больший из аргументов – при текущих значениях аргументов `n` и `m` это есть ссылка на переменную `m`. Поэтому команда `mpoint(n,m)=2` эквивалентна присваиванию переменной `m` значения `2`. Далее, командой `p=mpoint(n,m)` переменной `p` присваивается значение максимального из аргументов `n` и `m` – теперь это переменная `n` (поскольку на момент вызова означенной команды значение `n` равно `3`, а значение `m` равно `2`). Таким образом, переменная `n` остается со значением `3`, такое же значение имеет переменная `p`, а значение переменной `m` равно `2`. Отметим также, что аргументы функции `mpoint()` передаются по ссылке по тем же причинам, что и в предыдущем примере.

Указатели на функции

Чем фундаментальнее закономерность, тем проще ее можно сформулировать.

П. Капица

Это может показаться на первый взгляд странным, но указатель может ссылаться на функцию. Дело в том, что каждая функция хранится в памяти, соответствующая область памяти имеет адрес, и этот адрес можно записать в переменную-указатель на функцию. Вызов функции осуществляется через адрес, по которому она записана. Этот адрес также называют точкой входа в функцию.

Главное правило, которое следует запомнить, состоит в том, что **имя функции (без круглых скобок и аргументов) является указателем на функцию.** Значение этого указателя есть адрес, по которому записана функция.

Указатель на функцию объявляется следующим образом. Сначала указывается тип результата, который возвращается соответствующей функцией, затем заключенные в круглые скобки оператор * и имя указателя, а после этих круглых скобок еще одни круглые скобки с перечислением типов аргументов функции. Например, если функцией в качестве значения возвращается значение типа `int` и у нее два аргумента типа `double` и `char`, то указатель `p` на эту функцию объявляется как `int (*p)(double,char)`.

В качестве значения такому указателю присваивается имя функции, на которую должен ссылаться указатель. Пример использования указателя на функцию приведен в листинге 4.19.

Листинг 4.19. Указатели на функции

```
#include <iostream>
using namespace std;
//Функция возведения в квадрат:
double sqr(double x){
   return x*x;}
//Функция возведения в куб:
double cube(double x){
     return x*x*x;}
//Функция со вторым аргументом-указателем на функцию:
void myfunc(double x,double (*f)(double)){
   cout<<f(x)<<endl;}
int main(){
   double z;
   //Указатель на функцию:
   double (*p)(double);
   cout<<"z = ";
```

```
cin>>z;
//Указателю присваивается значение:
p=cube;
//Использование указателя и имени функции:
myfunc(z,sqr);
myfunc(z,p);
cout<<p(z)<<endl;
//Адрес функции:
cout<<sqr<<endl;
cout<<cube<<endl;
cout<<p<<endl;
return 0;
}
```

В программе объявляются три функции. У функции `sqr()` аргумент типа `double`, результатом является квадрат аргумента – число типа `double`. Функцией `cube()` в качестве значения возвращается значение типа `double` – куб аргумента, который также имеет тип `double`. У функции `myfunc()` два аргумента. Первый аргумент имеет тип `double`, а второй аргумент является указателем на функцию. Он объявлен как `double (*f)(double)`. В данном случае `f` – формальное название аргумента. Оператор `*` означает, что это указатель. Ключевое слово `double` в круглых скобках – тип аргумента функции, а `double` перед именем указателя – тип результата функции. Таким образом, функции `myfunc()` первым аргументом передается число, а вторым – имя функции.

В результате вызова функции отображается значение `f(z)`, то есть значение функции, имя которой указано вторым аргументом, от аргумента – первого параметра функции `myfunc()`.

В главном методе программы командой `double (*p)(double)` объявляется указатель на функцию. Значение указателю присваивается командой `p=cube`. После этого указатель `p` ссылается на функцию `cube()`. Далее этот указатель и имена функций используются в командах `myfunc(z,sqr)` (квадрат числа `z`), `myfunc(z,p)` (куб числа `z`) и `p(z)` (куб числа `z`). В конце программы разными способами отображаются адреса функций с использованием указателя `p` и имен функций. Результат выполнения программы может иметь вид (жирным шрифтом выделены вводимые пользователем данные):

```
z = 5
25
125
125
00401195
0040128F
0040128F
```

Еще раз обращаем внимание читателя, что имя функции является адресом области памяти, в которую записана функция. В этом отношении ситуация напоминает ситуацию с массивами – имя массива является указателем на первый элемент.

Рекурсия

Мой соперник не будет избран, если дела не пойдут хуже. А дела не пойдут хуже, если его не выберут.

Дж. Буш (старший)

Под рекурсией подразумевают вызов в теле функции этой же самой функции. Рекурсивный вызов может быть как прямым (функция вызывается в теле этой же функции), так и непрямым (в функции вызываются другие функции, в теле которых, в свою очередь, вызывается исходная функция). Обычно к такому приему прибегают, когда программируемая последовательность действий может быть сформулирована в терминах рекурсивной зависимости как последовательность значений, каждое из которых определяется на основе предыдущего по одной и той же схеме или принципу. В качестве примера использования рекурсии рассмотрим программу по вычислению факториала числа. Напомним, что факториал числа определяется как произведение всех натуральных чисел от единицы до этого числа включительно: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$. В силу этого определения можем представить факториал числа n в виде произведения факториала числа $(n-1)$ на число n, а именно: $n! = n \cdot (n-1)!$. Этим соотношением воспользуемся для создания специальной функции в C++ (листинг 4.20).

Листинг 4.20. Функция для вычисления факториала числа с использованием рекурсии

```
#include <iostream>
using namespace std;
int factorial(int n){
   if(n==1) return 1;
   else return n*factorial(n-1);
   }
int main(){
   int n;
   cout<<"Enter n = ";
   cin>>n;
   cout<<"n! = "<<factorial(n)<<endl;
   return 0;
   }
```

Целочисленная функция `factorial()` достаточно простая: у нее всего один целочисленный аргумент `n`, а тело кода состоит из условного оператора. Проверяемым является условие `n==1`, при выполнении которого возвращается значение `1` (следствие того замечательного факта, что факториал единицы равен единице, то есть $1! = 1$). Если аргумент у функции не единичный, значение вычисляется в виде выражения `n*factorial(n-1)`. В этом выражении вызывается определяемая функция, но с уменьшенным на единицу аргументом.

При вычислении значения функции возможны два варианта: аргумент единичный и аргумент больше единицы (другие варианты не рассматриваем в принципе). Если аргумент единичный – все просто. В качестве значения возвращается число `1`. В противном случае для вычисления значения функции вызывается эта же функция, но только аргумент у нее на единицу меньше. Если после уменьшения на единицу аргумент все равно не единичный, снова вызывается функция, и ее аргумент еще на единицу меньше и т.д. до тех пор, пока аргумент у вызываемой функции не станет единичным. Для единичного аргумента значение вычисляется «в лоб», после чего вся эта цепочка последовательных вызовов раскручивается в обратном порядке.

Отметим, что, несмотря на эффектность программного кода с рекурсивным вызовом, эффективностью он отличается редко, поскольку требует использования существенных системных ресурсов.

Перегрузка функций

Нам никто не мешает перевыполнить наши законы.

В.С. Черномырдин

Функции в C++ обладают замечательным свойством – их можно перегружать. **Под перегрузкой подразумевают создание и использование функций с разными прототипами, но одинаковыми названиями.** Поскольку название функции является частью ее прототипа, становится понятно, что отличаются прототипы не названием, а числом и типом аргументов и типом возвращаемого результата. Причем вполне достаточно хотя бы одного отличия.

Перегрузка функций – инструмент мощный и весьма полезный. Поэтому любой уважающий себя программист должен владеть им в совершенстве. Обычно перегрузку используют в тех случаях, когда приходится выполнять однотипные действия с разными объектами или разными типами данных.

При перегрузке функции, фактически, создается несколько функций с одинаковыми названиями, которые, однако, можно различить по остальным их атрибутам. При вызове презагруженной функции в программе выбор нужного варианта функции осуществляется исходя из использованного синтаксиса вызова функции. В листинге 4.21 приведен пример перегрузки функции.

Листинг 4.21. Перегрузка функции

```
#include <iostream>
using namespace std;
//Первый вариант функции:
void showArgs(double x){
   cout<<"Double-number "<<x<<endl;
   }
//Второй вариант функции:
void showArgs(double x,double y){
   cout<<"Double-numbers "<<x<<" and "<<y<<endl;
   }
//Третий вариант функции:
void showArgs(char s){
   cout<<"Symbol "<<s<<endl;
   }
//Четвертый вариант функции:
int showArgs(int n){
   return n;
   }
int main(){
   int n=3;
   double x=2.5,y=5.1;
   char s='w';
   //Первый вариант функции:
   showArgs(x);
   //Второй вариант функции:
   showArgs(x,y);
   //Третий вариант функции:
   showArgs(s);
   //Четвертый вариант функции:
   cout<<"Int-number "<<showArgs(n)<<endl;
   return 0;
   }
```

В программе описано четыре варианта функции `showArgs()`, назначение которой состоит в выводе на экран информации о ее аргументах. Однако в зависимости от количества и типа аргументов выполняются немного разные действия. В частности, предусмотрены такие варианты передачи аргументов:

1. Один аргумент типа `double`
2. Два аргумента типа `double`
3. Один аргумент типа `char`
4. Один аргумент типа `int`

В первых трех случаях результат функцией не возвращается (тип функции `void`), а на экран выводится сообщение о типе и значении аргумента (или аргументов). В четвертом случае функцией в качестве значения возвращается аргумент.

В главном методе программы функция `showArgs()` вызывается в разном контексте: вызов `showArgs(x)` соответствует варианту функции для одного аргумента типа `double`, вызов `showArgs(x,y)` соответствует варианту функции для двух аргументов типа `double`, вызов `showArgs(s)` соответствует варианту функции для одного аргумента типа `char`, вызов `cout<<"Int-number "<<showArgs(n)<<endl` соответствует варианту функции с одним аргументом типа `int`. Результат выполнения программы будет следующим:

```
Double-number 2.5
Double-numbers 2.5 and 5.1
Symbol w
Int-number 3
```

При работе с переопределенными функциями не следует забывать об автоматическом приведении типов. Эти два механизма в симбиозе могут давать очень интересные, а иногда и странные результаты. Рассмотрим пример. Так, в листинге 4.22 приведен исходный код программы, не содержащий перегруженных функций.

Листинг 4.22. Автоматическое приведение типов

```
#include <iostream>
using namespace std;
void ndiv(double x, double y){
   cout<<"x/y= "<<x/y<<endl;
   }
int main(){
   double x=10,y=3;
   int n=10,m=3;
   ndiv(x,y);
   ndiv(n,m);
   return 0;
   }
```

Результат выполнения такой программы будет следующим:

```
x/y= 3.33333
x/y= 3.33333
```

Первая строка, которая является результатом выполнения команды `ndiv(x,y)`, думается, вопросов не вызывает. Результатом вызова функции `ndiv()` является частное двух чисел – аргументов функции. Хотя в команде `ndiv(n,m)` аргументами функции указаны целые числа, благодаря автоматическому привсдению типов целочисленные значения расширяются до типа `double`, после чего вычисляется нужный результат.

Однако стоит изменить программный код, перегрузив функцию `ndiv()`, как показано в листинге 4.23, и результат изменится кардинально.

Листинг 4.23. Перегрузка и автоматическое приведение типов

```
#include <iostream>
using namespace std;
void ndiv(double x, double y){
   cout<<"x/y= "<<x/y<<endl;
   }
void ndiv(int n, int m){
   cout<<"n/m= "<<n/m<<endl;
   }
int main(){
   double x=10,y=3;
   int n=10,m=3;
   ndiv(x,y);
   ndiv(n,m);
   return 0;
   }
```

В частности, на экране в результате выполнения программы появится сообщение:

```
x/y= 3.33333
n/m= 3
```

Дело в том, что измененный программный код содержит перегруженную функцию `ndiv()`, для которой предусмотрен вызов как с двумя аргументами типа `double`, так и с двумя аргументами типа `int`. В последнем случае, хотя результат функции формально вычисляется так же, как и в исходном варианте, для целых чисел оператор `/` означает целочисленное деление с отбрасыванием остатка. Поэтому в результате получаем число `3`, а не `3.33333`, как для вызова функции с `double`-аргументами. Более того, нередко случаются ситуации, когда с учетом автоматического приведения ти-

пов невозможно однозначно определить, какой из вариантов перегруженной функции необходимо вызывать. Например, если для перегруженной функции предусмотрены два варианта передачи аргумента для типа данных `float` и для типа данных `double`, то передача функции в качестве аргумента `int`-переменной приведет к ошибке компиляции, поскольку непонятно, к какому формату данных (`float` или `double`) нужно преобразовывать тип `int`. Такие ситуации относятся к разряду логических ошибок, и их нужно внимательно отслеживать.

Как и обычные функции, перегруженные функции могут иметь аргументы со значениями, используемыми по умолчанию. Причем для каждого варианта перегружаемой функции эти значения по умолчанию могут быть разными. Единственное, за чем необходимо постоянно следить, – чтобы наличие значений по умолчанию у аргументов не приводило к неоднозначным ситуациям при вызове перегруженной функции. Ситуацию иллюстрирует пример в листинге 4.24.

Листинг 4.24. Перегрузка и значения по умолчанию

```
#include <iostream>
using namespace std;
void hello(){
   cout<<"Hello, my friend!\n";
   }
void hello(char str[],char name[]="Alex"){
   cout<<str<<", "<<name<<"!"<<endl;
   }
int main(){
   hello();
   hello("Hello","Peter");
   hello("Hi");
   return 0;
   }
```

У перегруженной функции `hello()` два варианта: без аргументов и с двумя аргументами-массивами типа `char`. Причем в последнем случае второй аргумент имеет значение по умолчанию.

Если функция вызывается без аргументов, в результате ее выполнения на экран выводится сообщение `Hello, my friend!`. При вызове функции с двумя аргументами (каждый аргумент – текстовая строка, реализованная в виде массива символов) на экран последовательно выводятся текстовые значения аргументов функции. Поскольку второй аргумент имеет значение по умолчанию `Alex`, то формально функцию можно вызывать и с одним текстовым аргументом. В главном методе функция `hello()` последова-

тельно вызывается без аргументов (команда `hello()`), с двумя аргументами (команда `hello("Hello","Peter")`) и с одним аргументом (команда `hello("Hi")`). Результат выполнения программы выглядит следующим образом:

```
Hello, my friend!
Hello, Peter!
Hi, Alex!
```

Но если попытаться для второго варианта функции (с двумя аргументами) указать значение по умолчанию и для первого аргумента, возникнет ошибка. Причина в том, что наличие у каждого из аргументов функции `hello()` значения по умолчанию приводит к неоднозначной ситуации: если функция при вызове указана без аргументов, невозможно определить, вызывается ли это вариант функции без аргументов, или нужно использовать вариант функции с двумя аргументами со значениями по умолчанию. Как уже отмечалось, на подобные ситуации следует постоянно обращать внимание при создании программных кодов в процессе перегрузки функций.

Примеры решения задач

В этом разделе рассматриваются примеры решения задач, в которых основу алгоритма составляют описываемые пользователем функции. Некоторые задачи повторяют те, что рассматривались в предыдущих главах, однако теперь соответствующие задачи решаются путем создания функции (или функций) пользователя.

Функция для вычисления гиперболического синуса

Напишем программу, в которой создается функция для вычисления гиперболического синуса по формуле $sh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + ... = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$.

В программе определено два варианта функции – т.е. функция перегружена. В первом варианте аргументом функции является переменная x, а граница ряда определяется как константа. Во втором случае количество слагаемых ряда передается вторым аргументом функции. Соответствующий программный код приведен в листинге 4.25.

Листинг 4.25. Синус гиперболический

```
#include <iostream>
#include <cmath>
using namespace std;
//Верхний предел суммы по умолчанию:
const int N=100;
//Функция с одним аргументом:
double sh(double z){
   int n;
   double s=z,q=z;
   for(n=1;n<=N;n++){
      q*=z*z/(2*n)/(2*n+1);
      s+=q;}
   return s;}
//Функция с двумя аргументами:
double sh(double z,int m){
   int n;
   double s=z,q=z;
   for(n=1;n<=m;n++){
      q*=z*z/(2*n)/(2*n+1);
      s+=q;}
   return s;}
int main(){
   //Аргумент для функции:
   double x;
   //Индексная переменная и предел для суммы:
   int i,m=9;
   //Ввод аргумента:
   cout<<"Enter x = ";
   cin>>x;
   //Значения ряда для разного числа слагаемых:
   for(i=1;i<=m;i++){
      cout<<i<<" : sh("<<x<<") = "<<sh(x,i)<<endl;}
   cout<<"--------------------\n";
   //Верхняя граница индексной переменной ряда равна N:
   cout<<N<<": sh("<<x<<") = "<<sh(x)<<endl;
   //Вызов встроенной функции:
   cout<<"Test value: "<<sinh(x)<<endl;
   return 0;}
```

В главном методе программы с помощью оператора цикла вызывается функция с явным указанием верхней границы ряда (число слагаемых в ряде на единицу больше) Далее вызывается вариант функции с одним

аргументом и, кроме этого, значение гиперболического синуса вычисляется с помощью встроенной функции. Результат может выглядеть так, как показано ниже:

```
Enter x = 5
1 : sh(5) = 25.8333
2 : sh(5) = 51.875
3 : sh(5) = 67.376
4 : sh(5) = 72.7583
5 : sh(5) = 73.9815
6 : sh(5) = 74.1776
7 : sh(5) = 74.2009
8 : sh(5) = 74.203
9 : sh(5) = 74.2032
--------------------
100: sh(5) = 74.2032
Test value = 74.2032
```

Чем меньше аргумент гиперболического синуса, тем меньше слагаемых нужно брать при вычислении соответствующего ряда. В приведенном примере при аргументе $x = 5$ уже при значении верхней границы индекса суммирования $m = 9$ вычисляется такое же значение, как и с помощью встроенной функции.

Отметим, что в данном случае вместо перегрузки функции можно было воспользоваться механизмом определения значения второго аргумента по умолчанию.

Вычисление произведения

Практически так же можно определить функцию для вычисления произведений. В данном случае воспользуемся рекурсией. Пример программного кода с использованием рекурсивной функции для вычисления произведения $\prod_{n=0}^{\infty}\left(1+x^{2^n}\right)=\frac{1}{1-x}$ (аргумент $\mid x \mid < 1$) приведен в листинге 4.26.

Листинг 4.26. Вычисление произведения

```
#include <iostream>
#include <cmath>
using namespace std;
//Функция (с рекурсией) для вычисления произведения:
double MyProd(double x,int N){
  if(N>0) return MyProd(x,N-1)*(1+pow(x,pow(2,N)));
  else return 1+x;}
```

```
int main(){
   //Аргументы функции:
   double x;
   int N;
   //Ввод аргумента:
   cout<<"Enter x = ";
   cin>>x;
   cout<<"Enter N = ";
   cin>>N;
   cout<<"Product value: "<<MyProd(x,N)<<endl;
   //Проверка результата:
   cout<<"Test value: "<<1/(1-x)<<endl;
   return 0;}
```

Следует учесть, что показатель степени в множителях произведения растет с ростом индексной переменной n как 2^n, поэтому большими значениями верхней границы произведения увлекаться не стоит. Типичный пример выполнения программы имеет вид

```
Enter x = 0.2
Enter N = 10
Product value: 1.25
Test value: 1.25
```

Видим, что даже небольшое количество слагаемых дает формально точный результат (с учетом точности округления).

Метод половинного деления

Уравнения вида $f(x) = 0$ могут решаться методом половинного деления. Корень ищется на интервале $x \in (a,b)$, таком что на границах интервала функция принимала значения разных знаков (что означает необходимость выполнения условия $f(a)f(b) < 0$). После того, как указан интервал поиска решения, проверяется значение функции $f(c)$ в центральной точке $c = \dfrac{a+b}{2}$. В эту точку переносится та граница интервала, знак функции на которой совпадает со знаком функции в центральной точке. Таким образом, интервал поиска уменьшается в два раза. Продолжая процесс необходимое количество раз, добиваемся нужной точности вычисления корня уравнения.

В листинге 4.27 приведен пример кода, в котором методом половинного деления реализовано решение уравнения $x^2 - 9x + 14 = 0$ (корни $x = 2$ и $x = 7$). При этом не только зависимость $f(x) = x^2 - 9x + 14$ реализована в виде функции, но и сама процедура поиска корня.

Листинг 4.27. Метод половинного деления

```
#include <iostream>
using namespace std;
//Функция для полинома:
double F(double x){
   return x*x-9*x+14;}
//Функция поиска корня:
double FindRoot(double (*f)(double),double a,double b,double eps){
   double c;
   while((b-a)/2>eps){
      c=(a+b)/2;
      if((f(a)*f(c))>0) a=c;
      else b=c;
      }
   return c;}
int main(){
   //Интервал, погрешность и корень:
   double a,b,eps,x;
   cout<<"interval: ";
   cin>>a;
   cin>>b;
   //Проверка корректности интервала:
   if(F(a)*F(b)>0){
      cout<<"Wrong interval!\n";
      return 0;} cout<<"error: ";
   cin>>eps;
   //Поиск решения:
   x=FindRoot(F,a,b,eps);
   cout<<"x = "<<x<<endl;
   return 0;}
```

При описании функции `FindRoot()` при передаче аргументов используется ссылка на функцию. Это позволяет использовать одну и ту же функцию для решения различных уравнений – достаточно только указать первым аргументом имя функции, определяющей решаемое уравнение. Второй и третий аргументы функции – левая и правая границы интервала поиска решения. Последний, четвертый аргумент – погрешность, с которой необходимо вычислить корень.

Пример выполнения программы при вычислении корня $x = 7$ может иметь вид:

```
interval: 3 10
error: 0.0001
x = 6.99998
```

Если интервал поиска решения введен некорректно, получим следующий результат

```
interval: 3 6
Wrong interval'
```

Близок по своей идеологии к методу половинного деления метод хорд

Метод хорд

От рассмотренного выше случая в методе хорд по-иному выбирается точка, в которой проверяется значение функции В частности, через граничные точки графика функции $f(x)$ (решается уравнение $f(x) = 0$) проводится хорда В качестве контрольной точки выбирается точка пересечения этой хорды с координатной осью Во всем остальном алгоритм такой же, как и в методе половинного деления При поиске корня на интервале $x \in (a,b)$ контрольная точка внутри этого интервала выбирается как $c = \dfrac{f(b)a - f(a)b}{f(b) - f(a)}$

В листинге 4 28 реализован программный код, с помощью которого решается все то же уравнение $x^2 - 9x + 14 = 0$, но уже методом хорд.

Листинг 4.28. Метод хорд

```
#include <iostream>
#include<cstdlib>
using namespace std;
//Функция для полинома:
double F(double x){
   return x*x-9*x+14;}
//Функция поиска корня:
double FindRoot(double (*f)(double),double a,double b,double eps){
   double c;
   while(abs(f(b)-f(a))>eps){
      c=(f(b)*a-f(a)*b)/(f(b)-f(a));
      if((f(a)*f(c))>0) a=c;
      else b=c;
      }
   return c;}
int main(){
   //Интервал, погрешность и корень:
   double a,b,eps,x;
   cout<<"interval: ";
   cin>>a;
   cin>>b;
   //Проверка корректности интервала:
   if(F(a)*F(b)>0){
```

```
    cout<<"Wrong interval!\n";
    return 0;} cout<<"error: ";
  cin>>eps;
  //Поиск решения:
  x=FindRoot(F,a,b,eps);
  cout<<"x = "<<x<<endl;
  return 0;}
```

В результате выполнения программы получаем

```
interval: 3 10
error: 0.0001
x = 7
```

Обращаем внимание, что в данном случае погрешность `eps` определяет не точность поиска корня, а точность выполнения соотношения $f(x) = 0$ (точнее, ширину интервала разности значений функции на интервале поиска). С практической точки зрения метод хорд обеспечивает более быструю сходимость по сравнению с методом половинного деления.

Метод Ньютона и перегрузка функции вычисления корня

Метод Ньютона уже рассматривался ранее при решении уравнений полиномиального типа. Здесь остановимся на реализации этого метода в том случае, если решаемое уравнение представлено в программе в виде функции. Особенность подхода состоит в том, что производная, значение которой необходимо знать для вычисления нового приближения для корня, рассчитывается в численном виде.

Напомним, что в методе Ньютона указывается начальное приближение для корня уравнения $f(x) = 0$, после чего итерационным образом этот корень начинает уточняться в соответствии с соотношением $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$.

Пример программы, в которой использован метод Ньютона для решения уравнения $x^2 - 9x + 14 = 0$, приведен в листинге 4.29.

Листинг 4.29. Метод Ньютона

```
#include <iostream>
using namespace std;
const int N=20;
//Функция для полинома:
double F(double x){
  return x*x-9*x+14;}
```

```
//Функция поиска корня:
double FindRoot(double (*f)(double),double x0,int n){
   double x=x0,df,h=0.00001;
   df=(f(x+h)-f(x))/h;
   for(int i=1;i<=n;i++)
   x=x-f(x)/df;
   return x;}
int main(){
   //Начальное приближение и корень:
   double x0,x;
   cout<<"initial x0 = ";
   cin>>x0;
   //Поиск решения:
   x=FindRoot(F,x0,N);
   cout<<"x = "<<x<<endl;
   return 0;}
```

Ниже представлен пример выполнения программы:

```
initial x0 = 12
x = 7.00048
```

Обращаем внимание читателя на способ вычисления производной для функции уравнения командой `df=(f(x+h)-f(x))/h`. Здесь фактически использована разностная схема для определения производной функции $f(x)$ в виде $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$, где шаг изменения аргумента `h` является малой величиной.

Отметим также, что на практике при программировании методов решения уравнений и прочих подобных задач используется перегрузка функций, что позволяет использовать, в зависимости от ситуации, различные алгоритмы вычисления корня. Пример такой перезагрузки можно найти в листинге 4.30.

Листинг 4.30. Перегрузка функции вычисления корня

```
#include <iostream>
using namespace std;
const int N=20;
//Функция для полинома:
double F(double x){
   return x*x-9*x+14;}
//Функция поиска корня (метод Ньютона):
double FindRoot(double (*f)(double),double x0){
   double x=x0,df,h=0.00001;
```

```
   df=(f(x+h)-f(x))/h;
   for(int i=1;i<=N;i++)
      x=x-f(x)/df;
   return x;}
//Функция поиска корня (метод хорд):
double FindRoot(double (*f)(double),double a,double b){
   double c;
   for(int i=1;i<=N;i++){
      c=(f(b)*a-f(a)*b)/(f(b)-f(a));
      if((f(a)*f(c))>0) a=c;
      else b=c;
      }
   return c;}
int main(){
   //Начальное приближение, интервал и корни уравнения:
   double x0,a,b,x1,x2;
   cout<<"initial x0 = ";
   cin>>x0;
   cout<<"interval: ";
   cin>>a;
   cin>>b;
   //Поиск решения:
   x1=FindRoot(F,x0);
   x2=FindRoot(F,a,b);
   cout<<"Newton method: x = "<<x1<<endl;
   cout<<"Chord method: x = "<<x2<<endl;
   return 0;}
```

По сравнению с предыдущими случаями, алгоритмы реализации метода хорд и метода Ньютона немного изменены: соответствующие итерационные процедуры выполняются определенное количество раз (определяется целочисленной константой `N`), а перегруженная функция `FindRoot()` утратила свой последний аргумент – теперь она может принимать два или три аргумента. Пример выполнения программы приведен ниже:

```
initial x0 = 12
interval: 3 12
Newton method: x = 7.00048
Chord method: x = 6.99998
```

Если у функции `FindRoot()` при вызове два аргумента, используется метод Ньютона. Если функции передано три аргумента – используется метод хорд.

Вывод строки

Рекурсия может использоваться в достаточно неожиданных ситуациях. В листинге 4.31 приведена программа с функцией для вывода в инверсном порядке строки, переданной аргументом этой функции. В функции с помощью рекурсии реализован следующий алгоритм. Если текущий элемент соответствующего символьного массива не является нуль-символом и не совпадает с символом `i`, вызывается функция для обработки следующего символа массива. Если элемент массива является символом `i`, работа функции заканчивается. Если элемент массива является нуль-символом окончания строки, завершается работа программы с сообщением `I didn't find any 'i'!`. Таким образом, если строка-аргумент функции содержит хотя бы одну литеру `i`, строка распечатывается в обратном порядке от этого символа (первого, если их несколько) до начала строки. Если символов `i` в строке нет, выводится сообщение `I didn't find any 'i'!`.

Листинг 4.31. Вывод строки в инверсном порядке

```
#include <iostream>
using namespace std;
void ShowStr(char *str){
     if(*str!='i'&&*str) ShowStr(str+1);
     else if(*str=='i') return;
           else {cout<<"I didn't find any 'ı'!\n";
             exit(0);}
     cout<<*str;
     }
int main(){
     char s[80];
     cout<<"Enter text here: ";
     gets(s);
     ShowStr(s);
     cout<<endl;
     return 0;}
```

Результат выполнения программы может иметь следующий вид:

```
Enter text here: Hello, my dear friend!
rf raed ym ,olleH
```

Здесь ввод пользователя выделен жирным шрифтом. Если символа `i` во введенной пользователем строке нет, результат будет следующим:

```
Enter text here: Hello, World!
I didn't find any 'ı'!
```

Отметим, что несмотря на некоторую эффектность приведенного выше кода, того же результата можно было добиться и более простыми средствами, без применения рекурсии.

Вычисление статистических характеристик

Рассмотрим пример создания специальных функций для вычисления статистических характеристик на основе данных массива, переданного аргументом функции. В листинге 4.32 приведен пример программы, в которой создано несколько перегруженных функций: для вычисления среднего значения, для вычисления дисперсии и для отображения указанных статистических характеристик. В последнем случае в соответствующей функции вызываются функции вычисления означенных показателей. Функции вычисления статистических характеристик перегружены так, что позволяют, кроме стандартных вычислений, проводить соответствующие вычисления по подмножествам.

Листинг 4.32. Статистические характеристики

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
//Среднее по всему массиву:
double avr(double *x,int n){
   int i;
   double s=0;
   for(i=0;i<n;i++) s+=x[i];
   s/=n;
   return s;
   }
//Среднее по части массива:
double avr(double *x,int n1,int n2){
   int i,n=n2-n1+1;
   double s=0;
   for(i=n1-1;i<n2;i++) s+=x[i];
   s/=n;
   return s;
   }
//Дисперсия по массиву:
double dev(double *x,int n){
   double x0,s=0;
   int i;
   x0=avr(x,n);
   for(i=0;i<n;i++) s+=x[i]*x[i];
   s/=n;
   return s-x0*x0;
   }
//Дисперсия по части массива:
double dev(double *x,int n1,int n2){
   double x0,s=0;
   int i,n=n2-n1+1;
   x0=avr(x,n1,n2);
```

```
   for(i=n1-1;i<n2;i++) s+=x[i]*x[i];
   s/=n;
   return s-x0*x0;
   }
//Отображение статистики для массива:
void ShowStat(double *x,int n){
   //Среднее:
   cout<<"Average is "<<avr(x,n)<<endl;
   //Дисперсия:
   cout<<"Dispersion is "<<dev(x,n)<<endl;
   //Стандартное отклонение:
   cout<<"Deviation is "<<sqrt(dev(x,n))<<endl;
   }
//Отображение статистики для части массива:
void ShowStat(double *x,int n1,int n2){
   //Среднее:
   cout<<"Average is "<<avr(x,n1,n2)<<endl;
   //Дисперсия:
   cout<<"Dispersion is "<<dev(x,n1,n2)<<endl;
   //Стандартное отклонение:
   cout<<"Deviation is "<<sqrt(dev(x,n1,n2))<<endl;
   }
//Функция для заполнения массива:
void fill(double *x,int n){
   for(int i=0;i<n;i++)
   x[i]=rand()%10;
   }
//Функция для отображения значений массива:
void show(double *x,int n){
   for(int i=0;i<n;i++)
   cout<<x[i]<<" ";
   cout<<endl;
   }
int main(){
   //Размер массива:
   const int N=10;
   //Массив значений:
   double z[N];
   //Заполнение массива:
   fill(z,N);
   cout<<"Base data:\n";
   //Отображение данных массива:
   show(z,N);
   cout<<"Base statistics:\n";
   //Статистика по массиву:
   ShowStat(z,N);
   cout<<"Small statistics:\n";
   //Статистика по части массива:
   ShowStat(z,3,N-2);
   return 0;}
```

У функции вычисления среднего значения `avr()` два варианта. При вычислении среднего значения по всему массиву данных аргументами функции передаются имя массива и его размер. Если для вычисления среднего значения используется только часть данных массива, вызывается вариант функции `avr()` с тремя аргументами: имя массива, порядковый номер (не индекс!) первого и последнего элементов подмассива, на основе которого вычисляется среднее. Аналогичная ситуация имеет место для функции вычисления дисперсии `dev()` и для функции отображения статистических показателей `ShowStat()`. Причем при определении функции `dev()` используется вызов соответствующего варианта функции `avr()`. В теле функции `ShowStat()` вызывается как функция `avr()`, так и функция `dev()`.

Кроме этих функций, в программе описана функция `fill()` для заполнения массива случайными числами в диапазоне от 0 до 9 включительно и функция `show()` для отображения значений массива. В результате выполнения программы получим следующее:

```
Base data:
1 7 4 0 9 4 8 8 2 4
Base statistics:
Average is 4.7
Dispersion is 9.01
Deviation is 3.00167
Small statistics:
Average is 5.5
Dispersion is 9.91667
Deviation is 3.14907
```

Для читателей, интересующихся вопросами статистики, отметим, что в данном случае вычислялась несмещенная дисперсия, или дисперсия по генеральной совокупности. Стандартное отклонение, которое также рассчитывается в программе, вычисляется как корень квадратный из дисперсии.

Транспонирование матрицы

В листинге 4.33 представлен программный код, с помощью которого выполняется транспонирование матриц. Для этой цели создана специальная функция. Функция является перегруженной, у нее два варианта: с одним аргументом и с двумя аргументами. Дело в том, что функция в качестве значения массив возвращать не может. Поэтому необходимо предусмотреть какой-то механизм возвращения результата (результат – транспонированная матрица).

Сами собой напрашиваются два подхода: передавать матрицу-результат аргументом функции либо вносить изменения непосредственно в исходную

матрицу. В соответствии с этим определяются и варианты функции транспонирования матриц. Исходная матрица (двумерный массив) передается первым аргументом функции `trans()`. Функция значения не возвращает. Если больше аргументов нет, изменения вносятся в эту матрицу. Если у функции есть еще один аргумент (двумерный массив такого же размера, как и первый), результат транспонирования записывается во вторую матрицу.

Листинг 4.33. Транспонирование матрицы

```
#include <iostream>
#include <cstdlib>
using namespace std;
//Размер матриц:
const N=3;
//Транспонирование матрицы (результат-второй аргумент):
void trans(double A[N][N],double B[N][N]){
   int i,j;
   for(i=0;i<N;i++)
   for(j=0;j<N;j++)
   B[i][j]=A[j][i];
   }
//Транспонирование матрицы (результат записывается в аргумент):
void trans(double A[N][N]){
   int i,j;
   double s;
   for(i=0;i<N;i++)
   for(j=i+1;j<N;j++){
      s=A[i][j];
      A[i][j]=A[j][i];
      A[j][i]=s;}
   }
//Заполнение матрицы случайными числами:
void fill(double A[N][N]){
   int i,j;
   for(i=0;i<N;i++)
   for(j=0;j<N;j++)
   A[i][j]=rand()%10;
   }
//Вывод матрицы на экран:
void show(double A[N][N]){
   int i,j;
   for(i=0;i<N;i++){
      for(j=0;j<N;j++)
         cout<<A[i][j]<<" ";
      cout<<endl;}
   }
int main(){
```

```
//Двумерные массивы:
double A[N][N],B[N][N];
cout<<"Initial matrix:\n";
//Заполнение массива:
fill(A);
//Отображение массива:
show(A);
cout<<"After transform:\n";
//Транспонирование:
trans(A,B);
//Результат:
show(B);
cout<<"Initial matrix:\n";
//Заполнение массива:
fill(A);
//Результат:
show(A);
cout<<"After transform:\n";
trans(A);
show(A);
return 0;
}
```

Кроме непосредственно функции для транспонирования матриц, в программе описана функция `fill()` для заполнения двумерного массива целыми случайными числами в диапазоне от 0 до 9, а также функция `show()` для выведения матрицы на экран.

В главном методе программы проверяется работа обоих вариантов перегруженной функции `trans()`. Результат выполнения программы имеет следующий вид:

```
Initial matrix:
1 7 4
0 9 4
8 8 2
After transform:
1 0 8
7 9 8
4 4 2
Initial matrix:
4 5 5
1 7 1
1 5 2
After transform:
4 1 1
5 7 5
5 1 2
```

Отметим, что при передаче двумерных массивов в качестве аргументов функций (при описании функций) размер по первому индексу можно было бы и не указывать, для второго индекса размер должен быть указан обязательно.

Резюме

Создавайте лишь немного законов, но следите за тем, чтобы они соблюдались.

Д. Локк

1. Использование функций существенно улучшает читабельность и функциональность программы. Функция – именованный блок программного кода, который может вызываться через имя не только в главном методе, но и в других функциях.
2. Функции могут передаваться аргументы (могут и не передаваться), функцией может возвращаться результат (а может и не возвращаться).
3. Результатом функции может быть любой базовый тип, указатель, ссылка, класс – практически все, кроме массива.
4. Главный метод программы – функция `main()` — также может иметь аргументы. Аргументами функции `main()` являются: количество параметров командной строки (включая имя программы) и символьный массив с названиями этих параметров.
5. Существует два механизма передачи аргументов функции: по значению и через ссылку. При передаче аргументов по значению на самом деле создается копия переменной, указанной аргументом функции. При передаче аргумента через ссылку функция имеет доступ непосредственно к переменным, указанным аргументами. По умолчанию используется механизм передачи аргументов по значению.
6. Для аргументов функции можно указывать значения по умолчанию. Если соответствующий аргумент функции явно не указан, используется значение по умолчанию. Аргументы со значением по умолчанию указываются последними в списке аргументов функции.
7. Под рекурсией подразумевают ситуацию, когда программный код функции реализуется через вызов этой же функции (как правило, с измененным аргументом). Использование рекурсии нередко упрощает структуру кода, но обычно приводит к неэффективному использованию системных ресурсов.

8. В C++ существует механизм перегрузки функций. В этом случае фактически создается несколько функций с одинаковыми названиями, но разными прототипами. Разные варианты перегруженной функции имеют разное количество или разный тип параметров, реже – разный тип результата. Механизм перегрузки используется при программировании однотипных действий (как правило, с разными данными) и позволяет для реализации общего алгоритма вызывать функцию с одним и тем же именем. Перегрузка функции является механизмом реализации концепции полиморфизма.

Контрольные вопросы

Американский народ сказал свое слово. Но чтобы понять, что он сказал, потребуется некоторое время.

В. Клинтон

1. Что такое функция? Как она объявляется?
2. Какой результат может возвращать функция?
3. Как функции передаются аргументы? Что такое передача аргумента по значению и по ссылке? Чем эти способы отличаются и как реализуются?
4. Каким образом функции в качестве значения передаются указатели?
5. Каким образом функции в качестве аргумента передаются массивы?
6. Какие аргументы у главного метода программы – функции `main()`?
7. Что такое аргументы по умолчанию и как они определяются?
8. Что такое рекурсия, и в каких случаях она используется?
9. Что такое перегрузка функции? В каких случаях применяется перегрузка?