Министерство образования Республики Беларусь

ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания к лабораторной работе № 2.2  
 по курсу «Основы алгоритмизации   
и программирования»**

«Функции»

Преподаватель: Войтехович   
Агния Витольдовна

Составитель: Войтехович   
Агния Витольдовна

Полоцк, 2017

# **ЦЕЛЬ РАБОТЫ**

Научиться создавать пользовательские функции: описание прототипа, заголовка и тела функции. Изучить методы передачи параметров в функции и возврат значений функциями

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

# 1 Общий вид функции

**Функции** — это строительные элементы языка С++ и то место, в котором выполняется вся работа программы.

В общем виде функция выглядит следующим образом:

возвр-тип имя-функции(список параметров)

{

тело функции

}

**Возвр-тип** определяет тип данного, возвращаемого функцией. Данное, возвращаемое функцией, называется также результатом. Соответственно, возвращаемый тип часто называется также типом результата. Функция может возвращать любой тип данных, за исключением массивов.

**Список параметров** — это список, элементы которого отделяются друг от друга запятыми. Каждый такой элемент состоит из имени переменной и ее типа данных. При вызове функции параметры принимают значения аргументов. Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать в явном виде, поместив для этого внутри скобок ключевое слово ***void***.

В объявлениях (декларациях) переменных можно объявить (декларировать) несколько переменных одного и того же типа, используя для этого список одних только имен, элементы которого отделены друг от друга запятыми. А все параметры функций, наоборот, должны объявляться отдельно, причем для каждого из них надо указывать и тип, и имя. То есть в общем виде список объявлений параметров должен выглядеть следующим образом:

f(тип имя\_переменной1, тип имя\_переменной2,  
..., тип имя\_переменнойN)

Вот, например, два объявления параметров функций, первое из которых правильное, а второе — нет:

f(int i, int k, int j) /\* правильное \*/

f(int i, k, float j) /\* неправильное, у переменной k должен

быть собственный спецификатор типа \*/

# 2 Область действия функции

В языке правила работы с областями действия — это правила, которые определяют, известен ли фрагменту кода другой фрагмент кода или данных, или имеет ли он доступ к этому другому фрагменту. Рассмотрим одну специальную область действия — ту, которая определяется функцией.

Каждая функция представляет собой конечный блок кода. Таким образом, она определяет область действия этого блока. Это значит, что код функции является закрытым и недоступным ни для какого выражения из любой другой функции, если только не выполняется вызов содержащей его функции. Код, который составляет тело функции, скрыт от остальной части программы, и если он не использует глобальных переменных, то не может воздействовать на другие части программы или, наоборот, подвергаться воздействию с их стороны. Иначе говоря, код и данные, определенные внутри одной функции, без глобальных переменных не могут воздействовать на код и данные внутри другой функции, так как у любых двух разных функций разные области действия.

Переменные, определенные внутри функции, являются локальными. Локальная переменная создается в начале выполнения функции, а при выходе из этой функции она уничтожается. Таким образом, локальная переменная не может сохранять свое значение в промежутках между вызовами функции. Единственное исключение из этого правила — переменные, объявленные со спецификатором класса памяти static. Таким переменным память выделяется так же, как и глобальным переменным, которые используются для хранения значений, но область действия таких переменных ограничена содержащими их функциями.

Формальные параметры функции также находятся в ее области действия. Это значит, что параметр доступен внутри всей функции. Параметр создается в начале выполнения функции, и уничтожается при выходе из нее.

Все функции имеют файл в качестве области действия (file scope). Таким образом, функцию нельзя определять внутри другой функции.

# 3 Аргументы функции

Если функция должна принимать аргументы, то в ее объявлении следует декларировать параметры, которые примут значения этих аргументов. Как видно из объявления следующей функции, объявления параметров стоят после имени функции.

/\* Возвращает 1, если символ c входит в строку s;

и 0 в противном случае. \*/

int is\_in(char \*s, char c)

{

while(\*s)

if(\*s==c) return 1;

else s++;

return 0; }

Функция **is\_in()** имеет два параметра: **s** и **c**. Если символ **c** входит в строку **s**, то эта функция возвращает **1**, в противном случае она возвращает **0**.

Хотя параметры выполняют специальную задачу, — принимают значения аргументов, передаваемых функции, — они все равно ведут себя так, как и другие локальные переменные. Формальным параметрам функции, например, можно присваивать какие-либо значения или использовать эти параметры в каких-либо выражениях.

## 3.1 Вызовы по значению и по ссылке

В языках программирования имеется два способа передачи значений подпрограмме. Первый из них — **вызов по значению**. При его применении в формальный параметр подпрограммы копируется значение аргумента. В таком случае изменения параметра на аргумент не влияют.

Вторым способом передачи аргументов подпрограмме является **вызов по ссылке**. При его применении в параметр копируется адрес аргумента. Это значит, что, в отличие от вызова по значению, изменения значения параметра приводят к точно таким же изменениям значения аргумента.

За небольшим количеством исключений, в языке С++ для передачи аргументов используется вызов по значению. Обычно это означает, что код, находящийся внутри функции, не может изменять значений аргументов, которые использовались при вызове функции.

Проанализируйте следующую программу:

#include <iostream>

using namespace std;

int sqr(int x);

int main(void)

{

int t=10;

cout<< sqr(t) << " -> " << t << endl;

return 0;

}

int sqr(int x)

{

x = x\*x;

return(x);

}

В этом примере в параметр **х** копируется **10** — значение аргумента для **sqr()**. Когда выполняется присваивание **х=х\*х**, модифицируется только локальная переменная **х**. А значение переменной **t**, использованной в качестве аргумента при вызове **sqr()**, по-прежнему остается равным **10**. Поэтому выведено будет следующее: **100** -> **10**.

Помните, что именно копия значения аргумента передается в функцию. А то, что происходит внутри функции, не влияет на значение переменной, которая была использована при вызове в качестве аргумента.

## 3.2 Вызов по ссылке

Хотя в С++ для передачи параметров применяется вызов по значению, можно создать вызов и по ссылке, передавая не сам аргумент, а указатель на него. Так как функции передается адрес аргумента, то ее внутренний код в состоянии изменить значение этого аргумента, находящегося, между прочим, за пределами самой функции.

Указатель передается функции так, как и любой другой аргумент. Конечно, в таком случае параметр следует декларировать как один из типов указателей. Это можно увидеть на примере функции **swap()**, которая меняет местами значения двух целых переменных, на которые указывают аргументы этой функции:

void swap(int \*x, int \*y)

{

int temp;

temp = \*x; /\* сохранить значение по адресу x \*/

\*x = \*y; /\* поместить y в x \*/

\*y = temp; /\* поместить x в y \*/

}

Функция **swap()** может выполнять обмен значениями двух переменных, на которые указывают х и y, потому что передаются их адреса, а не значения. Внутри функции, используя стандартные операции с указателями, можно получить доступ к содержимому переменных и провести обмен их значений.

Помните, что **swap()** (или любую другую функцию, в которой используются параметры в виде указателей) необходимо вызывать вместе с адресами аргументов. Следующая программа показывает, как надо правильно вызывать **swap()**:

#include <iostream>

using namespace std;

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

int main(void)

{

int i, j;

i = 10;

j = 20;

cout << "i и j перед обменом значениями: " << i << " " << j << endl;

swap(&i, &j); /\* передать адреса переменных i и j \*/

cout << "i и j после обмена значениями: " << i << " " << j << endl;

return 0;

}

void swap(int \*x, int \*y)

{

int temp;

temp = \*x; /\* сохранить значение по адресу x \*/

\*x = \*y; /\* поместить y в x \*/

\*y = temp; /\* поместить x в y \*/

}

И вот что вывела эта программа:

i и j перед обменом значениями: 10 20

i и j после обмена значениями: 20 10

В программе переменной **i** присваивается значение **10**, а переменной **j** — значение **20**. Затем вызывается функция **swap()** с адресами этих переменных. (Для получения адреса каждой из переменных используется унарный оператор **&**.) Поэтому в **swap()** передаются адреса переменных **i** и **j**, а не их значения.

## 3.3 Вызов функций с помощью массивов

В этом разделе рассказывается о передаче массивов функциям в качестве аргументов. Этот вопрос рассматривается потому, что эта операция является исключением по отношению к обычной передаче параметров, выполняемой путем вызова по значению, ведь при вызове по значению пришлось бы копировать весь массив.

Когда в качестве аргумента функции используется массив, то функции передается его адрес. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению. В случае передачи массива функции ее внутренний код работает с реальным содержимым этого массива и вполне может изменить это содержимое. Проанализируйте, например, функцию **print\_upper()**, которая печатает свой строковый аргумент на верхнем регистре:

#include <iostream>

#include <cstring>

using namespace std;

void print\_upper(char \*string);

int main(void)

{

char s[80];

cout << "Введите строку символов: ";

gets(s);

print\_upper(s);

cout << "\ns теперь на верхнем регистре: " << s << endl;

return 0;

}

/\* Печатать строку на верхнем регистре. \*/

void print\_upper(char \*string)

{

register int t;

for(t=0; string[t]; ++t) {

string[t] = toupper(string[t]);

putchar(string[t]);

}

}

Вот что будет выведено в случае фразы "This is a test." (это тест):

Введите строку символов: This is a test.

THIS IS A TEST.

s теперь в верхнем регистре: THIS IS A TEST.

Правда, эта программа не работает с символами кириллицы.

После вызова **print\_upper()** содержимое массива **s** в **main()** переводится в символы верхнего регистра. Если вам это не нужно, программу можно написать следующим образом:

#include <iostream>

#include <cstring>

using namespace std;

void print\_upper(char \*string);

int main(void)

{

char s[80];

cout << "Введите строку символов: ";

gets(s);

print\_upper(s);

cout << "\ns не изменялась: " << s << endl;

return 0;

}

/\* Печатать строку на верхнем регистре. \*/

void print\_upper(char \*string)

{

register int t;

for(t=0; string[t]; ++t)

putchar(toupper(string[t]));

}

Вот какой на этот раз получится фраза "This is a test.":

Введите строку символов: This is a test.

THIS IS A TEST.

s не изменилась: This is a test.

На этот раз содержимое массива не изменилось, потому что внутри **print\_upper()** не изменялись его значения.

Классическим примером передачи массивов в функции является стандартная библиотечная функция **gets()**. Хотя **gets()**, которая находится в вашей стандартной библиотеке, и более сложная, чем предлагаемая вам версия **xgets()**, но с помощью функции **xgets()** вы сможете получить представление о том, как работает **gets().**

/\* Упрощенная версия стандартной библиотечной функции gets(). \*/

char \*xgets(char \*s)

{

char ch, \*p;

int t;

p = s; /\* xgets() возвращает указатель s \*/

for(t=0; t<80; ++t){

ch = getchar();

switch(ch) {

case '\n':

s[t] = '\0'; /\* завершает строку \*/

return p;

case '\b':

if(t>0) t--;

break;

default:

s[t] = ch;

}

}

s[79] = '\0';

return p;

}

Функцию **xgets()** следует вызывать с указателем **char \***. Им, конечно же, может быть имя символьного массива, которое по определению является указателем **char \***. В самом начале программы **xgets()** выполняется цикл **for** от **0** до **80**. Это не даст вводить с клавиатуры строки, содержащие более **80** символов. При попытке ввода большего количества символов происходит возврат из функции. (В настоящей функции **gets()** такого ограничения нет.) Так как в языке С++ нет встроенной проверки границ, программист должен сам позаботиться, чтобы в любом массиве, используемом при вызове **xgets()**, помещалось не менее **80** символов. Когда символы вводятся с клавиатуры, они сразу записываются в строку. Если пользователь нажимает клавишу ***<Backspase>***, то счетчик **t** уменьшается на 1, а из массива удаляется последний символ, введенный перед нажатием этой клавиши. Когда пользователь нажмет ***<ENTER>***, в конец строки запишется нуль, т.е. признак конца строки. Так как массив, использованный для вызова **xgets()**, модифицируется, то при возврате из функции в нем будут находиться введенные пользователем символы.

# 4 Оператор return

Оператор **return** используется для выхода из функции. Отнесение его к категории операторов перехода обусловлено тем, что он заставляет программу перейти в точку вызова функции. Оператор **return** может иметь ассоциированное с ним значение, тогда при выполнении данного оператора это значение возвращается в качестве значения функции. В функциях типа **void** используется оператор **return** без значения.

Общая форма оператора return следующая:

return выражение;

Выражение присутствует только в том случае, если функция возвращает значение. Это значение выражения становится возвращаемым значением функции.

Внутри функции может присутствовать произвольное количество операторов **return**. Выход из функции происходит тогда, когда встречается один из них. Закрывающаяся фигурная скобка **}** также вызывает выход из функции. Выход программы на нее эквивалентен оператору **return** без значения. В этом случае функция, тип которой отличен от **void**, возвращает неопределенное значение.

## 4.1 Возврат из функции

Функция может завершать выполнение и осуществлять возврат в вызывающую программу двумя способами. Первый способ используется тогда, когда после выполнения последнего оператора в функции встречается закрывающая фигурная скобка (}).Например, функция **pr\_reverse()** в приведенной ниже программе просто выводит на экран в обратном порядке строку «***Мне нравится С++»***, а затем возвращает управление вызывающей программе.

#include <string.h>

#include <stdio.h>

void pr\_reverse(char \*s);

int main(void)

{

pr\_reverse("Мне нравится C++");

return 0;

}

void pr\_reverse(char \*s)

{

register int t;

for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);

}

Как только строка выведена на экран, функции **pr\_reverse()** "делать больше нечего", поэтому она возвращает управление туда, откуда она была вызвана.

Но на практике не так уж много функций используют именно такой способ завершения выполнения. В большинстве функций для завершения выполнения используется оператор **return** — или потому, что необходимо вернуть значение, или чтобы сделать код функции проще и эффективнее.

В функции может быть несколько операторов **return**. Например, в следующей программе функция **find\_substr()** возвращает начальную позицию подстроки в строке или же возвращает **-1**, если подстрока, наоборот, не найдена. В этой функции для упрощения кодирования используются два оператора **return**.

#include <iostream>

using namespace std;

int find\_substr(char \*s1, char \*s2);

int main(void)

{

if(find\_substr("C - это забавно", "is") != -1)

cout << "Подстрока найдена."<< endl;

return 0;

}

/\* Вернуть позицию первого, вхождения s2 в s1. \*/

int find\_substr(char \*s1, char \*s2)

{

register int t;

char \*p, \*p2;

for(t=0; s1[t]; t++) {

p = &s1[t];

p2 = s2;

while(\*p2 && \*p2==\*p) {

p++;

p2++;

}

if(!\*p2) return t; /\* 1-й оператор return \*/

}

return -1; /\* 2-й оператор return \*/

}

## 4.2 Возврат значений

Все функции, кроме тех, которые относятся к типу **void**, возвращают значение. Это значение указывается выражением в операторе **return**. Если для какой-либо функции указано, что она возвращает значение, то внутри этой функции у любого оператора return должно быть свое выражение. Однако если функция, тип которой отличен от void, выполняется до самого конца (то есть до закрывающей ее фигурной скобки), то возвращается произвольное (непредсказуемое с точки зрения разработчика программы!) значение. Хотя здесь нет синтаксической ошибки, это является серьезным упущением и таких ситуаций необходимо избегать.

Если функция не объявлена как имеющая тип **void**, она может использоваться как операнд в выражении. Поэтому каждое из следующих выражений является правильным:

x = power(y);

if(max(x,y) > 100) printf("больше");

for(ch=getchar(); isdigit(ch); ) ... ;

Общепринятое правило гласит, что вызов функции не может находиться в левой части оператора присваивания. Выражение

swap(x,y) = 100; /\* неправильное выражение \*/

является неправильным. Если компилятор в какой-либо программе найдет такое выражение, то пометит его как ошибочное и программу компилировать не будет.

В программе можно использовать функции трех видов. **Первый вид** — простые вычисления. Эти функции предназначены для выполнения операций над своими аргументами и возвращают полученное в результате этих операций значение. Вычислительная функция является функцией "в чистом виде". В качестве примеров можно назвать стандартные библиотечные функции **sqrt()** и **sin()**, которые вычисляют квадратный корень и синус своего аргумента соответственно.

**Второй вид** включает в себя функции, которые обрабатывают информацию и возвращают значение, которое показывает, успешно ли была выполнена эта обработка. Примером является библиотечная функция **fclose()**, которая закрывает файл. Если операция закрытия была завершена успешно, функция возвращает 0, а в случае ошибки она возвращает EOF.

У функций последнего, **третьего вида** нет явно возвращаемых значений. В сущности, такие функции являются чисто процедурными и никаких значений выдавать не должны. Примером является **exit()**, которая прекращает выполнение программы. Все функции, которые не возвращают значение, должны объявляться как возвращающие значение типа **void**. Объявляя функцию как возвращающую значение типа **void**, вы запрещаете ее применение в выражениях, предотвращая таким образом случайное использование этой функции не по назначению.

Иногда функции, которые, казалось бы, фактически не выдают содержательный результат, все же возвращают какое-то значение. Например, **printf()** возвращает количество выведенных символов. Если возвращаемое значение не входит ни в один из операторов присваивания, то это значение будет просто отброшено. Отметим также, что такое отбрасывание значения встречается очень часто. Проанализируйте следующую программу, в которой используется функция **mul()**:

include <iostream>

using namespace std;

int mul(int a, int b);

int main(void)

{

int x, y, z;

x = 10; y = 20;

z = mul(x, y); /\* 1 \*/

printf("%d", mul(x,y)); /\* 2 \*/

mul(x, y); /\* 3 \*/

return 0;

}

int mul(int a, int b)

{

return a\*b;

}

В строке 1 значение, возвращаемое функцией **mul()**, присваивается переменной **z**. В строке 2 возвращаемое значение не присваивается, но используется функцией **printf()**. И наконец, в строке 3 возвращаемое значение теряется, потому что не присваивается никакой из переменных и не используется как часть какого-либо выражения.

## 4.3 Возвращаемые указатели

Указатели не являются ни целыми, ни целыми без знака. Они являются адресами в памяти и относятся к особому типу данных. Так как размеры разных типов данных могут быть разными, то компилятор должен знать тип данных, на которые может указывать указатель. Поэтому в объявлении функции, которая возвращает указатель, тип возвращаемого указателя должен декларироваться явно. Например, нельзя объявлять возвращаемый тип как **int \***, если возвращается указатель типа **char \***!

Чтобы функция могла возвратить указатель, она должна быть объявлена как возвращающая указатель на нужный тип. Например, следующая функция возвращает указатель на первое вхождение символа, присвоенного переменной **с**, в строку **s**. Если этого символа в строке нет, то возвращается указатель на символ конца строки (**'0'**).

/\* Возвращает указатель на первое вхождение c в s. \*/

char \* match(char c, char \*s)

{

while(c!=\*s && \*s) s++;

return(s);

}

Вот небольшая программа, в которой используется функция match():

include <iostream>

using namespace std;

char \* match(char c, char \*s); /\* прототип \*/

int main(void)

{

char s[80], \*p, ch;

gets(s);

ch = getchar();

p = match(ch, s);

if(\*p) /\* символ найден \*/

cout<< p << endl;

else

cout << "Символа нет." << endl;

return 0;

}

Эта программа сначала считывает строку, а затем символ. Потом проводится поиск местонахождения символа в строке. При наличии символа в строке переменная **p** укажет на него, и программа выведет строку, начиная с найденного символа. Если символ в строке не найден, то **p** укажет на символ конца строки ( **'0'** ), причем **\*p** будет представлять логическое значение **ЛОЖЬ (false)**. В таком случае программа выведет сообщение «***Символа нет***».

## 4.4 Функция типа void

Одним из применений ключевого слова **void** является явное объявление функций, которые не возвращают значений. Мы уже знаем, что такие функции не могут применяться в выражениях, и указание ключевого слова **void** предотвращает их случайное использование не по назначению. Например, функция **print\_vertical()** выводит в боковой части экрана свой строчный аргумент по вертикали сверху вниз.

void print\_vertical(char \*str)

{

while(\*str)

cout << \*str++ << endl;

}

Вот пример использования функции **print\_vertical()**:

include <iostream>

using namespace std;

void print\_vertical(char \*str); /\* прототип \*/

int main(int argc, char \*argv[])

{

if(argc > 1) print\_vertical(argv[1]);

return 0;

}

void print\_vertical(char \*str)

{

while(\*str)

cout << \*str++ << endl;

}

# 5 Рекурсия

В языке С++ функция может вызывать сама себя. В этом случае такая функция называется рекурсивной. **Рекурсия** — это процесс определения чего-либо на основе самого себя, из-за чего рекурсию еще называют рекурсивным определением.

Простым примером рекурсивной функции является **factr(),** которая вычисляет факториал целого неотрицательного числа. Факториалом числа **n** (обозначается n!) называется произведение всех целых чисел, от 1 до **n** включительно (для 0, по определению, факториал равен 1.). Например, 3! — это 1×2×3, или 6. Здесь показаны **factr()** и эквивалентная ей функция, в которой используется итерация:

/\* рекурсивная функция \*/

int factr(int n) {

int answer;

if(n<1) return(1);

answer = factr(n-1)\*n; /\* рекурсивный вызов \*/

return(answer);

}

/\* нерекурсивная функция \*/

int fact(int n) {

int t, answer;

answer = 1;

for(t=1; t<=n; t++)

answer=answer \* t;

return(answer);

}

Нерекурсивное вычисление факториала, то есть вычисление с помощью **fact()**, выполняется достаточно просто. В этой функции в теле цикла, выполняющемся для t от 1 до n, вычисленное ранее произведение последовательно умножается на каждое из этих чисел. (Значение факториала для 0 получается, конечно, с помощью оператора присваивания. Значение факториала для 1 также получается умножением не на ранее полученное произведение, а на заранее подготовленное число, тоже равное 1.)

Работа же рекурсивной функции **factr()** чуть более сложная. Когда **factr()** вызывается с аргументом 0, то она сразу возвращает 1. Если же аргумент больше 0, то возвращается произведение **factr(n-1)\*n**. Чтобы вычислить значение этого выражения, **factr()** вызывается с аргументом **n-1**. Это выполняется до тех пор, пока n не станет равным 0. Когда это произойдет, вызовы функции начнут возвращать вычисленные ими значения факториалов.

При вычислении 2! первый вызов **factr()** влечет за собой второй, теперь уже рекурсивный вызов с аргументом 1, который, в свою очередь, влечет третий, тоже рекурсивный вызов с аргументом 0. Этот вызов возвращает число 1, которое затем умножается на 1, а потом на 2 (первоначальное значение n). Ответ в данном случае равен 2.

Когда функция вызывает сама себя, новый набор локальных переменных и параметров размещается в памяти в стеке, а код функции выполняется с самого своего начала, причем используются именно эти новые переменные. При рекурсивном вызове функции новая копия ее кода не создается. Новыми являются только значения, которые использует данная функция. При каждом возвращении из рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и сразу за рекурсивным вызовом возобновляется работа функции. При использовании рекурсивных функций стек работает подобно "телескопической" трубе, выдвигающейся вперед и складывающейся обратно.

Главным преимуществом рекурсивных функций является то, что с их помощью упрощается реализация некоторых алгоритмов, а программа становится понятнее.

В тексте рекурсивной функции обязательно должен быть выполнен условный оператор, например **if**, который при определенных условиях вызовет завершение функции, т.е. возврат, а не выполнит очередной рекурсивный вызов. Если такого оператора нет, то после вызова функция никогда не сможет завершить работу. Распространенной ошибкой при написании рекурсивных функций как раз и является отсутствие в них условного оператора. При создании программ не отказывайтесь от функции printf(); тогда вы сможете увидеть, что происходит на самом деле и сможете прервать выполнение, когда обнаружите ошибку.

# 6 Прототип функции

В современных, правильно написанных программах на языке С++ каждую функцию перед использованием необходимо объявлять. Обычно это делается с помощью прототипа функции. Хотя прототипы формально не требуются, но их использование очень желательно. Прототипы дают компилятору возможность тщательнее выполнять проверку типов, подобно тому, как это делается в таких языках как Pascal. Если используются прототипы, то компилятор может обнаружить любые сомнительные преобразования типов аргументов, необходимые при вызове функции, если тип ее параметров отличается от типов аргументов. При этом будут выданы предупреждения обо всех таких сомнительных преобразованиях. Компилятор также обнаружит различия в количестве аргументов, использованных при вызове функции, и в количестве параметров функции.

В общем виде прототип функции должен выглядеть таким образом:

тип имя\_функции(тип имя\_парам1, тип имя\_парам2,  
 ..., имя\_парамN);

Использование имен параметров не обязательно. Однако они дают возможность компилятору при наличии ошибки указать имена, для которых обнаружено несоответствие типов, так что не поленитесь указать эти имена — это позволит сэкономить время впоследствии.

Следующая программа показывает, насколько ценными являются прототипы функций. В ней выводится сообщение об ошибке, происходящей из-за того, что программа содержит попытку вызова **sqr\_it()** с целым аргументом, в то время как требуется указатель на целое.

/\* В этой программе используется прототип функции

чтобы обеспечить тщательную проверку типов. \*/

void sqr\_it(int \*i); /\* прототип \*/

int main(void)

{

int x;

x = 10;

sqr\_it(x); /\* несоответствие типов \*/

return 0;

}

void sqr\_it(int \*i)

{

\*i = \*i \* \*i;

}

В качестве прототипа функции может также служить ее определение, если оно находится в программе до первого вызова этой функции. Вот, например, правильная программа:

/\* Это определение будет также служить и

прототипом внутри этой программы. \*/

void f(int a, int b)

{

cout << a % b << endl;

}

int main(void)

{

f(10,3);

return 0;

}

В этом примере специальный прототип не требуется; так как функция **f()** определена еще до того, как она начинает использоваться в **main()**. Хотя определение функции и может служить ее прототипом в малых программах, но в больших такое встречается редко — особенно, когда используется несколько файлов.

Единственная функция, для которой не требуется прототип — это **main()**, так как это первая функция, вызываемая в начале работы программы.

Если функция не имеет параметров, то в ее прототипе внутри списка параметров стоит только ключевое слово **void**. Вот, например, прототип функции **f()** в том виде, в каком он должен быть в программе на языке С++:

float f(void);

Таким образом компилятор узнает, что у функции нет параметров, и любое обращение к ней, в котором имеются аргументы, будет считаться ошибкой.

Прототипы функций позволяют "отлавливать" ошибки еще до запуска программы. Кроме того, они запрещают вызов функций при несовпадении типов (т.е. с неподходящими аргументами) и тем самым помогают проверять правильность программы.

# 7 Структура программы, содержащей несколько функций

До этого момента разработка программ велась только с реализацией одной основной функции - функции **main**. При использовании функционального подхода структура программы приобретает более сложный вид, нежели без его использования. При разработке программ на языке C/С++ рекомендуется придерживаться следующей структуры программы:

* 1: подключение библиотек.
* 2: объявление глобальных пользовательских типов данных и переменных.
* 3: объявление прототипов пользовательских функций.
* 4: реализация функции **main**.
* 5: реализация (описание заголовков и тел) пользовательских функций.

В разделе 1 (подключение библиотек) осуществляется подключение тех библиотек языка С++, функции из которых будут использоваться в данной программе.

В разделе 2 (объявление глобальных пользовательских типов данных и переменных) осуществляется объявление новых пользовательских типов данных (как скалярных, так и векторных), которые должны быть доступны для объявления переменных или параметров в различных функциях программы. В языке С++ действует следующее правило: пользовательский тип данных является доступным для использования после его описания в пределах блока, в котором он описан. Под блоком здесь понимается фрагмент программного кода, заключенный в фигурные скобки. Если же тип данных описан вне какого-либо блока, то он является доступным во всех блоках, описанных после его объявления. Также в разделе 2 можно описывать глобальные переменные программы (понятие глобальных переменных будет рассматриваться далее).

В разделе 3 (объявление прототипов пользовательских функций) осуществляется описание прототипов пользовательских функций, которые будут реализованы в данной программе.

В разделе 4 (реализация функции **main**) осуществляется описание заголовка и тела функции **main** - основной функции программы, которая вызывается при запуске программы.

В разделе 5 (реализация пользовательских функций) осуществляется последовательное описание заголовков и тел пользовательских функций, которые необходимо реализовать в программе.