|  |  |
| --- | --- |
| ДИСЦИПЛИНА | Технологии индустриального программирования |
| ИНСТИТУТ | ИПТИП |
| КАФЕДРА | Индустриального программирования |
| ВИД УЧЕБНОГО МАТЕРИАЛА | Методические указания к практическим занятиям по дисциплине |
| ПРЕПОДАВАТЕЛЬ | Александров Алексей Сергеевич |
| СЕМЕСТР | 3 семестр, 2024/2025 уч. год |

# Практическое занятие №10

## Функции и процедуры

При написании кода могут возникать ситуации, когда код нужно использовать несколько раз. В таких случаях лучше всего его вывести в подпрограмму — функцию или процедуру. Это позволит значительно сократить объём кода.

**Функции**

Рассмотрим вот такую программу:

int name1 = "John";

int name2 = "James";

int name3 = "Jack";

int name4 = "Ivan";

int name5 = "Igor";

int name6 = "Boris";

std::cout << "Hello, " << name1 << "!\n";

std::cout << "Hello, " << name2 << "!\n";

std::cout << "Hello, " << name3 << "!\n";

std::cout << "Hello, " << name4 << "!\n";

std::cout << "Hello, " << name5 << "!\n";

std::cout << "Hello, " << name6 << "!\n";

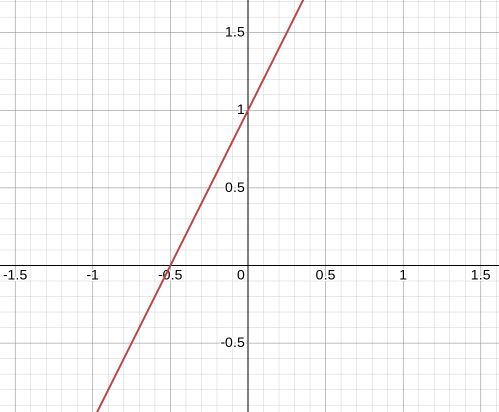
Несмотря на то, что данный код небольшой, дублирование одинакового кода приведёт к тому, что если, вместо вывода «Hello \*name\*» нужно будет вывести «Hello, user \*name\*», т.е. потребуется немного изменить вывод, то менять прийдётся все строчки кода, что приводит к выполнению лишней работы, а также ухудшает читаемость кода.

Также не стоит забывать, что дублирование кода может быть не сплошным, как в данном примере, а разбросанным по разным файлам — один и тот же код копировался и вставлялся в разные файлы, тогда возникает необходимость помнить, в каком файле, что копировалось. В таком случае нельзя исключать человеческий фактор — можно забыть внести изменения в какой-то из файлов и потратить несколько часов на поиск ошибки.

Решением этой проблемы являются подпрограммы.

По сути, подпрограмма — это выносной алгоритм, который представляет из себя некоторый фиксированный набор действий, допускающий использование переменных.

Самым простым примером подпрограммы может быть математическая функция. Предположим, что у нас есть такой график функции:



Чтобы записать математическое представление этого графика, можно записать все его точки:

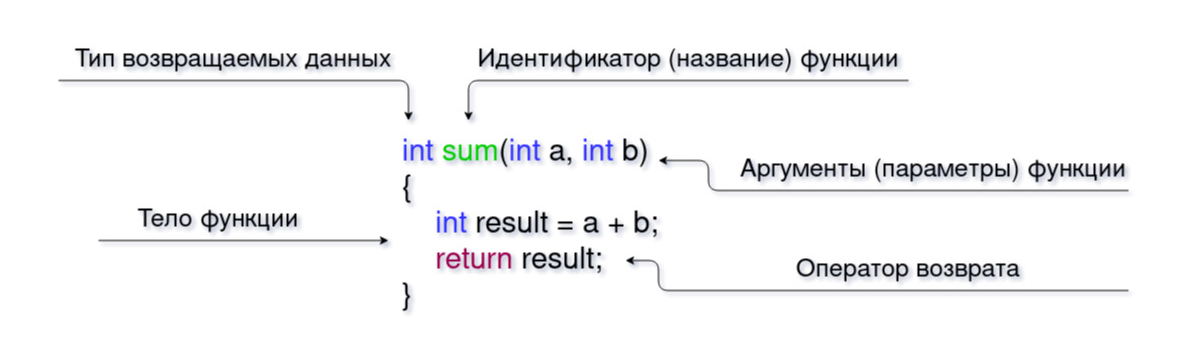
A(-0.5; 0); B(-0.25; 0.5); C(0; 1) и т.д.

Такое представление графика тяжело для восприятия, да и если график функции — прямая, то записать все её точки будет невозможно.

Поэтому вместо конкретных точек пишут функцию: y=2x+1. Такое представление понятно, а значение любой точки можно вычислить, просто подставляя в качестве *x* значение в функцию y(x), например, для точки А, значение y(-0.5) = 2\*(-0.5)+1 = 0. В данном случае X — параметр функции, -0.5 — конкретное значение Х, которое подставляется вместо Х, такое значение называется аргументом, а Y — значение функции при данном параметре.

В программировании работают такие же принципы. Если код повторяется, с изменением только небольшой части, то необходимо такой код вынести в функцию, а изменяющуюся часть «подставлять» как параметр этой функции. Результат выполнения функции, это «Y», т.е. конкретное значение функции при конкретном значении аргумента.

Синтаксис функций в C++ выглядит следующим образом:



Например, если взять функцию возведения числа в квадрат, она будет выглядеть следующим образом: y = x\*x, или в коде:

int pow(int x)

{

return x\*x;

}

int p = pow(10); // p = 100

В данном случае pow(x) это аналог f(x), т.е. это функция, x — параметр функции, 10 — аргумент функции, т.е. значение, которое *подставляется вместо x*, int p — результат выполнения функции, аналог y, а return — аналог знака = из математики. В данном случае функция выглядела бы так: y(x) = x\*x.

Функция может иметь любое количество параметров, или не иметь ни одного параметра вообще:

double getPiValue()

{

return 3.14;

}

double pi = getPiValue();

В данном случае функция не принимает ни одного параметра и возвращает число Пи.

Если в функции присутствуют параметры, то при её вызове необходимо указать их значения (аргументы):

int pow(int x)

{

return x\*x;

}

int p = pow(10); // p = 100

**Параметры по умолчанию**

Для функции можно указать параметры по умолчанию – это такие значения, которые будут подставляться на место параметров, в случае если значение аргумента не указано. Значение параметров по умолчанию указываются через знак = после параметра:

int pow(int x=2)

{

return x\*x;

}

int p = pow(); // p = 4

int p = pow(10); // p = 100

В данном примере значение параметра *х* указано по умолчанию равное 2, соответственно при вызове функции pow() без аргументов, было подставлено значение 2, но при вызове функции с указанием аргумента, функция была выполнена с указанным значением.

Параметры по умолчанию в функции должны идти только после обязательных параметров, например:

int sum(int a, int b**=10**) // правильно

{

return a+b;

}

int p = sum(5, 7); // p = 12

И другой вариант (так делать нельзя):

int sum(int a=10, int b) // неправильно

{

return a+b;

}

int p = sum(5, 7); // ошибка компиляции

Важно отметить, что название параметра не обязательно должно совпадать с названием переменной в коде. Название параметра — переменная которая существует лишь внутри функции.

Таким образом принцип работы функции может быть записан следующим образом:

Предположим, что у нас есть функция pow(x):

int pow(int x)

{

return x\*x;

}

int value = 10;

int p = pow(value); // p = 100

Как этот код выглядел бы, если функцию «развернуть», т.е. предположить, что вместо функции идёт сплошной код?

int value = 10; // число, которое мы хотим возвести в квадрат

int x = value; // создание параметра функции и присвоение значения

int result = x\*x; // формирование результата выполнения функции

int p = result; // присвоение результата выполнения функции

А что будет, если название переменной и параметра функции будет одинаковым? Например:

int pow(int x)

{

return x\*x;

}

int x = 10; // название переменной x, параметр тоже x

int p = pow(x); // p = 100

В данном случае, несмотря на одинаковые названия, они будут точно также двумя разными переменными, независимыми друг от друга:

int x = 10; // число, которое мы хотим возвести в квадрат

int x\_2 = x; // создание параметра функции и присвоение значения

int result = x\_2\*x\_2;// формирование результата выполнения функции

int p = result; // присвоение результата выполнения функции

Таким образом стоит учитывать, что функция, это, в первую очередь, выносной алгоритм, который принимает параметры, изолированно от основного кода выполняет заданный алгоритм, и возвращает лишь результат своего выполнения.

**Результатом выполнения функции всегда является только одно значение, т.е. функция может вернуть только одно число/строку/объект.**

Простыми словами — функция y(x) = k\*x+b не может при одном и том же значении x иметь разные значения y — у функции может быть много параметров, например, функция y() может зависеть не только от x, но и от k и b: y(k,x,b) = k\*x+b, но результат всегда один – значение y.

**Процедуры**

Однако, не всегда функции возвращают результат. В рассматриваемых примерах функция всегда возвращала какое-либо значение, например целое число (int). Но не всегда есть необходимость возвращать какое-либо значение, в таком случае тип возвращаемого значения функции указывается как void, что буквально обозначает пустой тип данных.

Такие функции называются процедурами, и используются обычно для действий, не требующих значимого результата, например, для вывода текста на экран – результатом является не значение, а действие, которое пользователь видит на экране.

Таким образом код из начала:

int name1 = "John";

int name2 = "James";

int name3 = "Jack";

int name4 = "Ivan";

int name5 = "Igor";

int name6 = "Boris";

std::cout << "Hello, " << name1 << "!\n";

std::cout << "Hello, " << name2 << "!\n";

std::cout << "Hello, " << name3 << "!\n";

std::cout << "Hello, " << name4 << "!\n";

std::cout << "Hello, " << name5 << "!\n";

std::cout << "Hello, " << name6 << "!\n";

Можно заменить процедурой:

void showName(string name) // процедура для вывода имени

{

std::cout << "Hello, " << name << "!\n";

}

int name1 = "John";

int name2 = "James";

int name3 = "Jack";

int name4 = "Ivan";

int name5 = "Igor";

int name6 = "Boris";

showName(name1); // вызов процедуры

showName(name2);

showName(name3);

showName(name4);

showName(name5);

showName(name6);

Таким образом, если потребуется изменить формат вывода текста на экран, то достаточно будет изменить только код внутри процедуры.

Стоит отметить, что функции и процедуры могут быть созданы в классе, тогда они будут называться методами:

class myclass

{

int sum(int a; int b) // это метод

{

return a+b;

}

}

Таким образом можно сказать, что функция — это выносной алгоритм, принимающий некоторое количество параметров и возвращающий результат выполнения алгоритма. Функции, которые ничего не возвращают, называются процедурами. Функции и процедуры в классе называются методами.

Однако все рассмотренные реализации предполагают возвращение только одного значения после выполнения функции, через оператор return.

Но что делать, если требуется, чтобы функция вернула не одно, а сразу несколько значений? Или требуется вернуть результат выполнения из процедуры?

Для этого можно использовать различные виды параметров функции — передача значений через адрес или указатель.

**Переменные**

Все переменные хранятся в памяти, в которой у каждой переменной есть свой адрес — т.е. номер ячейки в памяти, в котором эта переменная хранится, и записывается в 16-м виде, например, 0х152 или 0х248. Аналогом адреса переменной может быть индекс элемента в массиве — если массив состоит из 10 элементов, то все значения в нём будут иметь индексы от 0 до 9, при этом значения элементов массива могут быть любыми:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Значение | 15 | 8 | -3 | 6 | 8 | -17 | 21 | -5 | 11 | 26 |

Следовательно, зная индекс элемента в массиве, к этому элементу возможно обратиться и получить или записать значение, независимо от того, какое число там лежит в данный момент времени. Именно так и работают адреса – если мы знаем адрес переменной, то в неё можно произвести запись или прочитать значение, не обращаясь к ней напрямую, например, если запись нужно произвести из функции.

**Адрес переменной**

Адрес в C++ обозначают символом &, который ставится перед названием переменной:

int a = 15; // исходная переменная

int &b = a; // адрес переменной

b = 10; // a = 10, т.к. b - это адрес переменной a

В данном фрагменте кода создаётся переменная a, со значением 15. Переменная b является адресом, т.к. перед ней стоит знак &, и хранит она адрес переменной a.

Таким образом, можно создать функцию:

void pow(int &b) // параметр – адрес переменной

{

b = b\*b;

}

int a = 10;

pow(a); // вызываем функцию, передавая в неё переменную

// a = 100;

После выполнения функции pow(), значение a будет равно 100, т.е. функция pow() принимает не значение, а адрес переменной, т.е. номер ячейки в памяти, где лежит переменная, а следовательно, запись b = b\*b; нужно читать следующим образом: значение, которое лежит по адресу b равно значение, которое лежит по адресу b, умножить на значение, которое лежит по адресу b, что эквивалентно a = a\*a, т.е. b является адресом переменной a.

Или, по аналогии с массивами, — значение под индексом i равно значение под индексом i умножить на значение под индексом i.

Таким образом, несмотря на то, что в данном случае функция void pow(int &b) по сути является процедурой, и, как следствие, не может вернуть значение через оператор return, она всё таки, можно сказать, что возвращает значение, записывая его через адрес переменной (int &b) — т.е. возврат значения происходи не явно.

Таким образом, если в функцию требуется передать значение переменной только на чтение, то используются обычные параметры (передача по значению), например, void pow(int b), но если требуется передать значение в функцию, и для чтения, и для записи, то необходимо использовать передачу по адресу, например, void pow(int &b).

Это же можно представить в виде таблицы:

|  |  |  |  |
| --- | --- | --- | --- |
| Вид параметра | Чтение | Запись | Пример |
| Значение | Да | Нет | void pow(int b) |
| Адрес | Да | Да | void pow(int &b) |

Тут же может возникнуть логичный вопрос — почему бы не использовать всегда передачу по адресу, ведь читать значение можно в обоих случаях, а записывать — только по адресу — значит можно всегда использовать адрес, даже если запись не требуется? Ответ — это не запрещается, но не рекомендуется — можно случайно записать данные, что сломает алгоритм работы, поэтому, если требуется работать со значением в формате «только для чтения», то нет необходимости передавать значение «для записи».

*Примечание:* Однако, если понимать механику работы адресов, то можно оптимизировать выполнение кода по времени выполнения и использованию памяти — при передачи данных по адресу, не происходит копирование данных, особенно, если передаётся объект класса с большим количеством полей — тем самым, передавая в функцию адрес объекта, он не копируется, а значит экономится время и память.

Всё то же самое относится и к передаче указателей в функцию.