Функции в C++.

Русаков Андрей ИС1-23

Определение функции

Функция определяет действия, которые выполняет программа. Функции позволяют выделить набор инструкций и придать ему имя. А затем многократно по присвоенному имени вызывать в различных частях программы. По сути функция - это именованный блок кода. Первая строка представляет заголовок функции. Вначале указывается возвращаемый тип функции. Если функция не возвращает никакого значения, то используется тип **void**.

```
тип имя_функции(параметры)
{
    инструкции
}
```

Затем идет имя функции, которое представляет произвольный идентификатор. К именованию функции применяются те же правила, что и к именованию переменных.

После имени функции в скобках идет перечисление параметров. Функция может не иметь параметров, в этом случае указываются пустые скобки.

```
void hello()
{
    std::cout << "hello\n";
}
```

После заголовка функции в фигурных скобках идет тело функции, которое содержит выполняемые инструкции.

Для возвращения результата функция применяет оператор **return**. Если функция имеет в качестве возвращаемого типа любой тип, кроме void, то она должна обязательно с помощью оператора return возвращать какое-либо значение.

Выполнение функции

Для выполнения функции ее необходимо вызвать. Вызов функции осуществляется в форме. После имени функции указываются скобки, в которых перечисляются аргументы - значения для параметров функции.

Например, определим и выполним простейшую функцию. Здесь определена функция hello, которая вызывается в функции main два раза. В этом и заключается преимущество функций: мы можем вынести некоторые общие действия в отдельную функцию и затем вызывать многократно в различных местах программы. В итоге программа два раза выведет строку "hello".

```
имя_функции(аргументы);
```

```
#include <iostream>

void hello()
{
    std::cout << "hello\n";
}

int main()
{
    hello();
    hello();
    return 0;
}
```

Объявление функции

При использовании функций стоит учитывать, что компилятор должен знать о функции до ее вызова. Поэтому вызов функции должен происходить после ее определения, как в случае выше. В некоторых языках это не имеет значение, но в языке C++ это играет большую роль. И если, к примеру, мы сначала вызовем, а потом определим функцию, то мы получим ошибку на этапе компиляции, как в следующем случае:

```
1  #include <iostream>
2
3  int main()
4  {
5      hello();
6      hello();
7      return 0;
8  }
9
10 void hello()
11 {
12     std::cout << "hello\n";
13 }
```

Параметры функции

Через параметры в функцию можно передать различеые значения. Параметры перечисляются в скобках после имени функции имеют следующее определение.

```
тип название_параметра
```

Функция exchange принимает два параметра типа double, которые называются currate (текущий курс) и sum (сумма, которую надо обменять). При вызове функции exchange для этих параметров необходимо передать значения.

Функция display принимает параметры типов string и int.

Функция square принимает число и выводит на консоль его квадрат. Параметр функции представляет тип int, однако при ее вызове ей передается число с плавающей точкой, то есть значение типа double.

```
#include <iostream>
#include <string>

void square(int);
void display(std::string, int);

int main()
{
    display("Tom", 33);
    square(4.56);

    return 0;
}
void square(int x)
{
    std::cout << "Square of " << x << " is equal to " << x * x << std::endl;
}

void display(std::string name, int age)
{
    std::cout << "Name: " << name << "\tAge: " << age << std::endl;
```

Аргументы по умолчанию

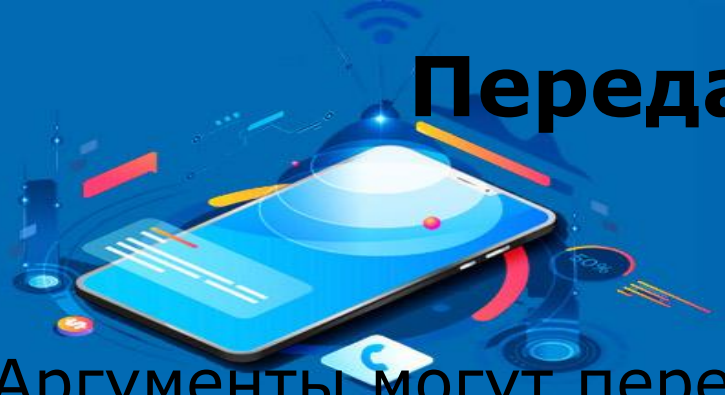
Функция может принимать аргументы по умолчанию, то есть некоторые значения, которые функция использует, если при вызове для параметров явным образом не передается значение.

При объявлении прототипа подобной функции он тоже может содержать значение по умолчанию для параметра. И в этом случае мы можем не определять в функции значение по умолчанию для параметра - оно будет браться из прототипа:

```
#include <iostream>

void multiply(int n, int m = 3)
{
    int result = n * m;
    std::cout << "n * m = " << result << std::endl;
}
int main()
{
    multiply(4, 5);
    multiply(4);
    return 0;
}
```

```
#include <iostream>

void multiply(int n, int m=3);

int main()
{
    multiply(4, 5);
    multiply(4);
    return 0;
}

void multiply(int n, int m)
{
    int result = n * m;
    std::cout << "n * m = " << result << std::endl;
}
```

Передача аргументов по значению



Аргументы могут передаваться по значению (by value) и по ссылке (by reference). При передаче аргументов по значению внешний объект, который передается в качестве аргумента в функцию, не может быть изменен в этой функции. В функцию передается само значение этого объекта.

Функция square принимает два числа и возводит их в квадрат. При выполнении мы увидим, что изменения аргументов в функции square действуют только в рамках этой функции. Вне ее значения переменных a и b остаются неизменными:

```
#include <iostream>

void square(int, int);

int main()
{
    int a = 4;
    int b = 5;
    std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;
    square(a, b);
    std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

    return 0;
}
void square(int a, int b)
{
    a = a * a;
    b = b * b;
    std::cout << "In square: a = " << a << "\tb=" << b << std::endl;
}
```

```
Before square: a = 4  b = 5
In square: a = 16          b = 25
After square: a = 4    b = 5
```

Передача параметров по ссылке

При передаче параметров по ссылке передается ссылка на объект, через которую мы можем манипулировать самим объектов, а не просто его значением. Теперь параметры a и b передаются по ссылке. Ссылочный параметр связывается непосредственно с объектом, поэтому через ссылку можно менять сам объект. И если мы скомпилируем и запустим программу, то результат будет иным

```
#include <iostream>

void square(int&, int&);

int main()
{
    int a = 4;
    int b = 5;
    std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;
    square(a, b);
    std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

    return 0;
}
void square(int &a, int &b)
{
    a = a * a;
    b = b * b;
    std::cout << "In square: a = " << a << "\tb=" << b << std::endl;
}
```

```
Before square: a = 4  b = 5
In square: a = 16         b = 25
After square: a = 16  b = 25
```

Передача параметров по ссылке

Передача по ссылке позволяет возвратить из функции сразу несколько значений. Также передача параметров по ссылке является более эффективной при передаче очень больших объектов. От передачи аргументов по ссылке следует отличать передачу ссылок в качестве аргументов. Если функция принимает аргументы по значению, то изменение параметров внутри функции также никак не скажется на внешних объектах, даже если при вызове функции в нее передаются ссылки на объекты.

```
#include <iostream>

void square(int, int);

int main()
{
    int a = 4;
    int b = 5;
    int &aRef = a;
    int &bRef = b;
    std::cout << "Before square: a = " << a << "\tb=" << b << std::endl;
    square(aRef, bRef);
    std::cout << "After square: a = " << a << "\tb=" << b << std::endl;

    return 0;
}
void square(int a, int b)
{
    a = a * a;
    b = b * b;
    std::cout << "In square: a = " << a << "\tb=" << b << std::endl;
```

```
Before square: a = 4  b = 5
In square: a = 16        b = 25
After square: a = 4    b = 5
```

Константные параметры

Параметры могут быть константными - значения таких параметров не могут меняться. Например:

```
void square(const int n)
{
    // n = n * n;  // можно считать значение параметра, но не изменять его
    std::cout << n * n << std::endl;
}
```

То же самое касается и передачи параметра по ссылке:

```
void square(const int &n)
{
    // n = n * n;  // можно считать значение параметра, но не изменять его
    std::cout << n << std::endl;
}
```

Константному параметру можно передать в качестве аргумента как константу, так и переменную:

```
#include <iostream>

void square(const int, const int);

int main()
{
    const int a = 4;
    int b = 5;
    square(a, b);   // 20
    return 0;
}
void square(const int a, const int b)
{
    //a = a * a;      так нельзя сделать
    //b = b * b;      так нельзя сделать
    std::cout << "In square: a * b = " << a * b << std::endl;
}
```

Константные ссылки

Если функция получает аргументы по ссылке, то чтобы передать в функцию константу, параметры тоже должны представлять ссылку на константу:

И если в функцию необходимо передать большие объекты, которые не должны изменяться, то определение параметров именно как константных ссылок больше всего подходит для данной задачи.

```
#include <iostream>

void square(const int&, const int&);

int main()
{
    const int a = 4;
    const int b = 5;
    square(a, b);   // 20
    return 0;
}
void square(const int &a, const int &b)
{
    // a = a * a;   так нельзя сделать
    // b = b * b;   так нельзя сделать
    std::cout << "In square: a * b = " << a * b << std::endl;
}
```

Оператор return и возвращение результата

Для возвращения результата из функции применяется оператор **return**. Этот оператор имеет две формы:

Первая форма используется, если в качестве возвращаемого типа функции применяется тип **void**.

В данном случае функция factorial вычисляет факториал переданного числа. Однако если число меньше 1, то функция выводит соответствующее сообщение, и с помощью оператора return осуществляется выход из функции.

```
return;
return выражение;
```

```
#include <iostream>

void factorial(int);

int main()
{
    factorial(-3);
    factorial(5);
    factorial(4);
    return 0;
}

void factorial(int n)
{
    if(n<1)
    {
        std::cout << "Incorrect number" << std::endl;
        return;
    }
    int result = 1;
    for(int i = 1; i <=n; i++)
    {
        result *= i;
    }
    std::cout << "Factorial of " << n << " is equal to " << result << std::endl;
}
```

Оператор return и возвращение результата

Вторая форма оператора return применяется для возвращения результата из функции. Если функция имеет в качестве возвращаемого типа любой тип, отличный от void, то такая функция обязательно должна возвратить некоторое значение с помощью оператора return. Причем значение, которое возвращается оператором return, должно соответствовать возвращаемому типу функции, либо допускать неявное преобразование в этот тип.

Так как функция factorial возвращает значение, то ее результат можно присвоить какой-нибудь переменной или константе:

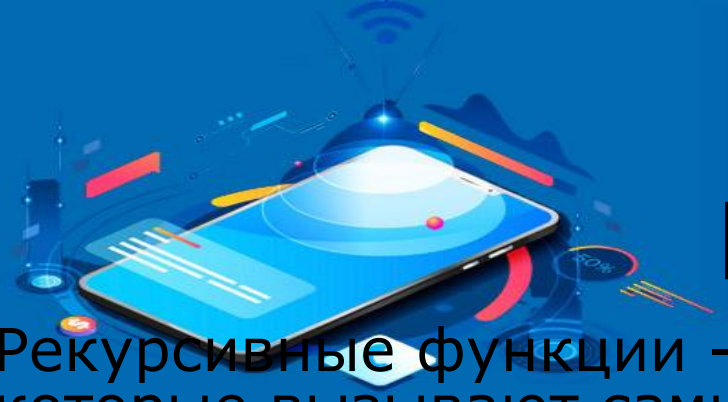
```cpp
#include <iostream>

int factorial(int);

int main()
{
    int n = 5;
    int result = factorial(n);
    std::cout << "Factorial of " << n << " is equal to " << result << std::endl;
    return 0;
}

int factorial(int n)
{
    int result = 1;
    for(int i = 1; i <=n; i++)
    {
        result *= i;
    }
    return result;
}
```

```cpp
int result = factorial(n);
```

Возвращение ссылки

Не следует возвращать ссылку на локальный объект, который создается внутри функции. Поскольку все создаваемые в функции объекты удаляеются после ее завершения, а их память очищается, то возвращаемая ссыла будет указывать на несуществующий объект, как в следующем случае:

```
int &factorial(int n)
{
    int result = 1;
    for(int i = 1; i <=n; i++)
    {
        result *= i;
    }
    return result;
}
```



Рекурсивные функции

Рекурсивные функции - это функции, которые вызывают сами себя. Например, определим вычисление факториала в виде рекурсивной функции:

В функции factorial задано условие, что если число n больше 1, то это число умножается на результат этой же функции, в которую в качестве параметра передается число n-1. То есть происходит рекурсивный спуск. И так далее, пока не дойдем того момента, когда значение параметра не будет равно 1. В этом случае функция возвратит 1. Рекурсивная функция обязательно должна иметь некоторый базовый вариант, который использует оператор **return** и к которому сходится выполнение остальных вызовов этой функции

```
#include <iostream>

int factorial(int);

int main()
{
    int n = 5;
    int result = factorial(n);
    std::cout << "Factorial of " << n << " is equal to " << result << std::en
    return 0;
}

int factorial(int n)
{
    if(n>1)
        return n * factorial(n-1);
    return 1;
}
```

Другим распространенным показательным примером рекурсивной функции служит функция, вычисляющая числа Фиббоначчи. n-й член последовательности чисел Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1. Значения f(0)=0 и f(1)=1, таким образом, определяют базовые варианты для данной функции:

Результат работы программы - вывод 10 чисел из последовательности Фиббоначчи на консоль:

```
#include <iostream>

int fibonachi(int);

int main()
{
    int n;
    for(int i = 0; i < 10; i++)
    {
        n = fibonachi(i);
        std::cout << n << "\t";
    }
    std::cout << std::endl;
    return 0;
}
int fibonachi(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
```

```
0,  1,  1,  2,  3,  5,  8,  13,  21,  34
```

Внешние объекты

Кроме функций внешние файлы могут содержать различные объекты - переменные и константы. Для подключения внешних объектов в файл кода применяется ключевое слово **extern**.

Для объявления объектов определим файл **objects.h** со следующим содержимым.

Для определения этих объектов добавим новый файл **objects.cpp**:

Используем эти объекты в файле **app.cpp**:

```
extern const int x;
extern double y;
```

```
#include "objects.h"

const int x = 5;
double y = 3.4;
```

```
#include <iostream>
#include "objects.h"

int main()
{
    std::cout << "x = " << x << std::endl;
    std::cout << "y = " << y << std::endl;

    return 0;
}
```