Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА
Институт Информационных Технологий
Кафедра Промышленной Информатики

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Тема лекции «ФУНКЦИИ в С++. ИТЕРАЦИЯ И РЕКУРСИЯ. ЗАДАЧА ПРО ШАРИКИ»

Лектор Каширская Елизавета Натановна (к.т.н., доцент, ФГБОУ ВО "МИРЭА - Российский технологический университет") e-mail: liza.kashirskaya@gmail.com

Лекция № 6

ФУНКЦИИ в С++

Сегодня мы поговорим о функциях в С++. Очень часто в программировании необходимо выполнять одни и те же действия. Например, мы хотим выводить пользователю сообщения об ошибке в разных местах программы, если он ввел неверное значение. Без функций это выглядело бы так:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string valid_pass = "qwerty123";
string user_pass;
cout << "Введите пароль: ";
getline(cin, user_pass);
if (user_pass == valid_pass) { cout << "Доступ разрешен." << endl;
} else {
cout << "Неверный пароль!" << endl;
} return 0;
}
```

А вот аналогичный пример с функцией:

```
#include <iostream>
#include <string>
using namespace std;
void check_pass (string password)
{
        string valid_pass = "qwerty123";
        if (password == valid_pass) {
                        cout << "Доступ разрешен." << endl;
} else {
        cout << "Неверный пароль!" << endl;
        }
}
int main()
{
        cout << "Введите пароль: ";
        string user_pass;
        getline (cin, user_pass);
        check_pass (user_pass);
        return 0;
}
```

void (*англ.*) - пустота

По сути, после компиляции не будет никакой разницы для процессора, как для первого кода, так и для второго. Но ведь такую проверку пароля мы можем делать в нашей программе довольно много раз. И тогда получается копипаста, и код становится нечитаемым.

ФУНКЦИИ в С++

Функции — один из самых важных компонентов языка С++.

- Любая функция имеет тип, также, как и любая переменная.
- Функция может возвращать значение, тип которого аналогичен типу самой функции.
- Если функция не возвращает никакого значения, то она должна иметь тип **void** (такие функции иногда называют процедурами).
- При объявлении функции после ее типа должно находиться имя функции и две круглые скобки — открывающая и закрывающая, внутри которых могут находиться один или несколько аргументов функции, которых также может не быть вообще.
- После списка аргументов функции ставится открывающая фигурная скобка, после которой находится само тело функции.
- В конце тела функции обязательно ставится закрывающая фигурная скобка.

Определение функции

Все функции можно разбить на две категории: те, которые не возвращают значений, и те, которые их возвращают. Функции, не возвращающие значений, называются функциями типа **void** и имеют следующую общую форму:

```
void имяФункции(списокПараметров)
{
оператор(ы)
return; // не обязательно
}
```

Обычно функция **void** используется для выполнения каких-то действий. Например, функция, которая должна напечатать слово "Cheers!» (Ура!) заданное число раз (n) может выглядеть следующим образом:

```
void cheers(int n) // возвращаемое значение отсутствует
{
for (int i = 0; i < n; i++)
std::cout << "Cheers! ";
std::cout << std::endl;
}
```

Параметр **int n** означает, что cheers() ожидает передачи значения типа **int** в качестве аргумента при вызове функции.

Функция с возвращаемым значением передает генерируемое ею значение функции, которая ее вызвала. Другими словами, если функция возвращает квадратный корень из 9.0 (sqrt (9.0)), то вызывающая ее функция получит значение 3.0. Такая функция объявляется, как имеющая тот же тип, что и у возвращаемого ею значения.

Вот общая форма:

```
имяТипа имяФункции(списокПараметров)
{ оператор (ы)
return значение; // значение приводится к типу имяТипа
}
```

Возвращаемое значение

Функции с возвращаемыми значениями требуют использования оператора **return** таким образом, чтобы вызывающей функции было возвращено значение. Само значение может быть константой, переменной либо общим выражением. Единственное требование — выражение должно сводиться по типу к имяТипа, либо может быть преобразовано в имяТипа. (Если объявленным возвращаемым типом является, скажем, double, а функция возвращает выражение int, то int приводится к double.) Затем функция возвращает конечное значение в вызывавшую ее функцию.

Язык C++ накладывает ограничения на типы возвращаемых значений: **возвращаемое значение не может быть массивом**. Все остальное допускается — целые числа, числа с плавающей точкой, указатели и даже объекты. (Интересно, что хотя функция C++ не может вернуть массив непосредственно, она все же может вернуть его в составе объекта.)

Функция завершается после выполнения оператора **return**. Если функция содержит более одного оператора **return**, например, в виде альтернатив разных выборов **if … else**, то в этом случае она **прекращает свою работу по достижении первого оператора return**. Например, в следующем коде конструкция **else** излишняя, однако она помогает понять намерение разработчика:

```
int bigger (int a, int b)
{
if (a > b )
return a; // если a > b, функция завершается здесь
else
return b; // в противном случае функция завершается здесь
}
```

Пример построения функции

```
#include <iostream>

using namespace std;

void function_name ()

{

	cout << "Hello, world" << endl;

}

int main()

{

function_name(); // Вызов функции

return 0;

}
```

Перед вами тривиальная программа, **Hello, world**, только реализованная с использованием функций.

Если мы хотим вывести «Hello, world» где-то еще, нам просто нужно вызвать соответствующую функцию. В данном случае это делается так: **function_name();.**

Вызов функции имеет вид имени функции с последующими круглыми скобками. Эти скобки могут быть пустыми, если функция не имеет аргументов. Если же аргументы в самой функции есть, их необходимо указать в круглых скобках.

Параметры и аргументы функции

Во многих случаях нам нужно будет передавать данные в вызываемую функцию, чтобы она могла с ними как-то взаимодействовать. Например, если мы хотим написать функцию умножения двух чисел, то нам нужно каким-то образом сообщить функции, какие это будут числа. В противном случае, как она узнает, что на что перемножать? Здесь нам на помощь приходят параметры и аргументы.

Также существует такое понятие, как параметры функции по умолчанию. Такие параметры можно не указывать при вызове функции, т.к. они примут значение по умолчанию, указанное после знака присваивания после данного параметра и списка всех параметров функции.

В предыдущих примерах мы использовали функции типа **void**, которые не возвращают никакого значения. Как вы знаете, оператор **return** используется для возвращения вычисляемого функцией значения.

Параметры функции

Рассмотрим пример функции, возвращающей значение, на примере проверки пароля.

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

using namespace std;

string check_pass (string password)
{
    string valid_pass = "qwerty123";
    string error_message;
    if (password == valid_pass) {
        error_message = "Доступ разрешен.";
    } else {
        error_message = "Неверный пароль!";
    }
    return error_message;
}

int main()
{
    string user_pass;
    cout << "Введите пароль: ";
    getline (cin, user_pass);
    string error_msg = check_pass (user_pass);
    cout << error_msg << endl;
    return 0;
}
```

В данном случае функция **check_pass** имеет тип **string**, следовательно, она будет возвращать только значение типа string, иными словами говоря, строку. Давайте рассмотрим алгоритм работы этой программы.

Самой первой выполняется функция **main()**, которая должна присутствовать в каждой программе. Теперь мы объявляем переменную **user_pass** типа string, затем выводим пользователю сообщение «Введите пароль», который после ввода попадает в строку user_pass. А вот дальше начинает работать наша собственная функция check_pass().

Аргументы функции — это, если сказать простым языком, переменные или константы **вызывающей** функции, которые будет использовать вызываемая функция. Это по сути **фактические параметры**.

В качестве аргумента этой функции передается строка, введенная пользователем.

При объявлении функций создается **формальный параметр**, имя которого может отличаться от параметра, передаваемого при вызове этой функции. Но типы формальных параметров и передаваемых функции аргументов (фактических параметров) должны быть одинаковы.

После того, как произошел вызов функции check_pass(), начинает работать данная функция. Если функцию нигде не вызвать, то этот код будет проигнорирован программой.

Итак, мы передали в качестве аргумента строку, которую ввел пользователь. Теперь эта строка в полном распоряжении функции. Хочу обратить ваше внимание на то, что переменные и константы, объявленные в разных функциях, независимы друг от друга, они даже могут иметь одинаковые имена.

Теперь мы проверяем, правильный ли пароль ввел пользователь или нет. Если пользователь ввел правильный пароль, присваиваем переменной error_message соответствующее значение, если нет, то сообщение об ошибке.

После этой проверки мы **возвращаем** переменную error_message. На этом работа нашей функции закончена. А теперь, в функции main(), то значение, которое возвратила наша функция, мы присваиваем переменной error_msg и выводим это значение (строку) на экран терминала.

Смотрите еще один пример:

Функции очень сильно облегчают работу программисту и намного повышают читаемость и понятность кода, в том числе и для самого разработчика.

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

using namespace std;

bool password_is_valid (string password)
{
        string valid_pass = "qwerty123";
        if (valid_pass == password)
                return true;
        else
                return false;
}

void get_pass ()
{
        string user_pass;
        cout << "Введите пароль: ";
        getline(cin, user_pass);
        if (!password_is_valid(user_pass)) {
                cout << "Неверный пароль!" << endl;
                get_pass (); // Здесь делаем рекурсию
        } else {
                cout << "Доступ разрешен." << endl;
        }
}

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

Итак, **функция** — это последовательность операторов для выполнения определенного задания.

Часто ваши программы будут прерывать выполнение одних функций ради выполнения других. Вы делаете аналогичные вещи в реальной жизни постоянно. Например, вы читаете книгу и вспомнили, что должны были сделать телефонный звонок. Вы оставляете закладку в своей книге, берете телефон и набираете номер. После того, как вы уже поговорили, вы возвращаетесь к чтению: к той странице, на которой остановились.

Программы на языке C++ работают похожим образом. Иногда, когда программа выполняет код, она может столкнуться с вызовом функции. **Вызов функции** — это выражение, которое приказывает процессору прервать выполнение текущей функции и приступить к выполнению другой функции. Процессор «оставляет закладку» в текущей точке выполнения, а затем выполняет вызываемую функцию. Когда выполнение вызываемой функции завершено, процессор возвращается к *закладке* и возобновляет выполнение прерванной функции.

Пример функции

Функция, в которой находится вызов, называется **caller**, а функция, которую вызывают — **вызываемая функция**, например:

```cpp
#include <iostream> // для std::cout и std::endl

// Объявление функции doPrint(), которую мы будем вызывать
void doPrint() {
    std::cout << "In doPrint()" << std::endl;
}

// Объявление функции main()
int main()
{
    std::cout << "Starting main()" << std::endl;
    doPrint(); // прерываем выполнение функции main() вызовом функции doPrint(). Функция main() в данном случае является caller
    std::cout << "Ending main()" << std::endl;
    return 0;
}
```

Результат выполнения программы:

```
Starting main()
In doPrint()
Ending main()
```

Пример функции

Эта программа начинает выполнение с первой строки функции main(), где выводится на экран следующая строка: Starting main(). Вторая строка функции main() вызывает функцию doPrint(). На этом этапе выполнение операторов в функции main() приостанавливается, и процессор переходит к выполнению операторов внутри функции doPrint(). Первая (и единственная) строка в doPrint() выводит текст ”In doPrint()". Когда процессор завершает выполнение doPrint(), он возвращается обратно в main() к той точке, на которой остановился. Следовательно, следующим оператором является вывод строки Ending main().

Обратите внимание, для вызова функции нужно указать её имя и список параметров в круглых скобках (). В примере на предыдущем слайде **параметры не используются**, поэтому круглые **скобки пусты**.

Правило: не забывайте указывать круглые скобки () при вызове функций.

Оператор return

Когда функция main() завершает свое выполнение, она возвращает целочисленное значение обратно в операционную систему, используя **оператор return**.

Функции, которые мы пишем, также могут возвращать значения. Для этого нужно указать **тип возвращаемого значения** (или ***«тип возврата»***). Он указывается при объявлении функции, перед её именем. Обратите внимание, тип возврата не указывает, какое именно значение будет возвращаться. Он указывает только тип этого значения.

Затем, внутри вызываемой функции, мы используем оператор **return**, чтобы указать **возвращаемое значение** — какое именно значение будет возвращаться обратно в caller.

Рассмотрим простую функцию, которая возвращает целочисленное значение:

```cpp
#include <iostream>

// int означает, что функция возвращает целочисленное значение обратно в caller
int return7()
{
    // Эта функция возвращает целочисленное значение, поэтому мы должны использовать оператор return
    return 7; // возвращаем число 7 обратно в caller
}

int main()
{
    std::cout << return7() << std::endl; // выведется 7
    std::cout << return7() + 3 << std::endl; // выведется 10

    return7(); // возвращаемое значение 7 игнорируется, так как функция main() ничего с ним не делает

    return 0;
}
```

Результат выполнения программы:
7
10

Оператор return

Разберемся детально.

- Первый вызов функции return7() возвращает 7 обратно в caller, которое затем передается в **std::cout** для вывода.
- Второй вызов функции return7() опять возвращает 7 обратно в caller. Выражение 7 + 3 имеет результат 10, который затем выводится на экран.
- Третий вызов функции return7() опять возвращает 7 обратно в caller. Однако функция main() ничего с ним не делает, поэтому ничего и не происходит (возвращаемое значение игнорируется).

Примечание. Возвращаемые значения не выводятся на экран, если их не передать объекту std::cout. В последнем вызове функции return7() значение не отправляется в std::cout, поэтому ничего и не происходит.

Но, на самом деле, такая функция не будет вызываться, компилятор её встроит в тело main. Вызов такой функции не выгоден. Нужно пойти, взять код функции, в стек положить адрес возврата и т.д.

Тип возврата void

Как вы уже знаете, функции могут и не возвращать значения. Чтобы сообщить компилятору, что функция не возвращает значение, нужно использовать **тип возврата void**. Взглянем еще раз на функцию doPrint() из примера на 21-ом слайде.

```
void doPrint() // void - это тип возврата
{
    std::cout << "In doPrint()" << std::endl;
    // Эта функция не возвращает никакого значения, поэтому оператор return здесь не нужен
}
```

Эта функция имеет тип возврата void, который означает, что функция не возвращает значения. Поскольку значение не возвращается, то и оператор return не требуется.

Вот еще один пример использования функции типа void:

```cpp
#include <iostream>

// void означает, что функция не возвращает значения
void returnNothing()
{
    std::cout << "Hi!" << std::endl;
    // Эта функция не возвращает никакого значения, поэтому оператор return здесь не нужен
}

int main()
{
    returnNothing(); // функция returnNothing() вызывается, но обратно в main() ничего не возвращает

    std::cout << returnNothing(); // ошибка, эта строчка не скомпилируется. Вам нужно будет её закомментировать
    return 0;
}
```

В первом вызове функции returnNothing() выводится **Hi!**, но ничего не возвращается обратно в caller. Точка выполнения возвращается обратно в функцию main(), где программа продолжает свое выполнение.

Второй вызов функции returnNothing() даже не скомпилируется. Функция returnNothing() имеет тип возврата void, который означает, что эта функция не возвращает значения. Однако функция main() пытается отправить это значение (которое не возвращается) в std::cout для вывода. std::cout не может обработать этот случай, так как значения на вывод не предоставлено. Следовательно, компилятор выдаст ошибку.

Возврат значений обратно в функцию main()

Теперь у вас есть понимание того, как работает функция **main().** Когда программа выполняется, операционная система делает вызов функции main() и начинается её выполнение. Операторы в **main()** выполняются последовательно. В конце функция main() возвращает целочисленное значение (обычно 0) обратно в операционную систему. Поэтому **main()** объявляется как **int main().**

Возврат значений обратно в функцию main()

Почему нужно возвращать значения обратно в операционную систему? Дело в том, что возвращаемое значение функции main() является **кодом состояния**, который сообщает операционной системе об успешном или неудачном выполнении программы. Обычно возвращаемое значение 0 (ноль) означает, что всё прошло успешно, тогда как любое другое значение означает неудачу/ошибку.

Обратите внимание, по стандартам языка C++ функция main() должна возвращать целочисленное значение. Однако, если вы не укажете return в конце функции main(), компилятор возвратит 0 автоматически, если никаких ошибок не будет. Но рекомендуется указывать return в конце функции main() и использовать тип возврата int для функции main().

Во-первых, если тип возврата функции не void, то она должна возвращать значение указанного типа (использовать оператор return). Единственно исключение — функция main(), которая возвращает 0, если не предоставлено другое значение.

Во-вторых, когда процессор встречает в функции оператор return, он немедленно выполняет возврат значения обратно в caller, и точка выполнения также переходит в caller. Любой код, который находится за return-ом в функции — игнорируется.

Еще о возвращаемых значениях

Функция может возвращать только одно значение через return обратно в caller. Это может быть либо число (например, 7), либо значение переменной, либо выражение (у которого есть результат), либо определенное значение из набора возможных значений.

Но есть способы обойти правило возврата одного значения, возвращая сразу несколько значений. Для этого нужно использовать указатели на функцию (ну, или ссылки, но я вам о них еще не говорила).

Наконец, автор функции решает, что означает её возвращаемое значение. Некоторые функции используют возвращаемые значения в качестве кодов состояния для указания результата выполнения функции (успешно ли выполнение или нет). Другие функции возвращают определенное значение из набора возможных значений. Кроме того, существуют функции, которые вообще ничего не возвращают.

Одну и ту же функцию можно вызывать несколько раз, даже в разных программах, что очень полезно.

```cpp
#include <iostream>

// Функция getValueFromUser() получает значение от пользователя, а затем возвращает его обратно в caller
int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;
    return x;
}

int main()
{
    int a = getValueFromUser(); // первый вызов функции getValueFromUser()
    int b = getValueFromUser(); // второй вызов функции getValueFromUser()

    std::cout << a << " + " << b << " = " << a + b << std::endl;

    return 0;
}
```

Результат выполнения программы:

Enter an integer: 4
Enter an integer: 9
4 + 9 = 13

Здесь **main()** прерывается 2 раза. Обратите внимание, в обоих случаях полученное пользовательское значение сохраняется в переменной **x**, а затем передается обратно в **main()** с помощью **return**, где присваивается переменной **a** или **b.**

Также **main()** не является единственной функцией, которая может вызывать другие функции. Любая функция может вызывать любую другую функцию!

```cpp
#include <iostream>

void print0()
{
    std::cout << "0" << std::endl;
}

void printK()
{
    std::cout << "K" << std::endl;
}

// Функция print0K() вызывает как print0(), так и printK()
void print0K()
{
    print0();
    printK();
}

// Объявление main()
int main()
{
    std::cout << "Starting main()" << std::endl;
    print0K();
    std::cout << "Ending main()" << std::endl;
    return 0;
}
```

Результат выполнения программы:

Starting main()
0
K
Ending main()

Вложенные функции

В языке C++ одни функции не могут быть объявлены внутри других функций (т.е. быть вложенными). Следующий код вызовет ошибку компиляции:

```cpp
#include <iostream>

int main()
{
    int boo() // эта функция находится внутри функции main(), что запрещено
    {
        std::cout << "boo!";
        return 0;
    }

    boo();
    return 0;
}
```

Правильно вот так:

```cpp
#include <iostream>

int boo() // теперь уже не в main()
{
    std::cout << "boo!";
    return 0;
}

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

Какие из следующих программ не скомпилируются (и почему), а какие скомпилируются (и какой у них результат)?

Программа № 1:

```
#include <iostream>

int return5()
{
    return 5;
}

int return8()
{
    return 8;
}

int main()
{
    std::cout << return5() + return8() << std::endl;

    return 0;
}
```

Программа № 2:

```
#include <iostream>

int return5()
{
    return 5;

    int return8()
    {
        return 8;
    }
}

int main()
{
    std::cout << return5() + return8() << std::endl;

    return 0;
}
```

Программа № 3:

```
#include <iostream>

int return5()
{
    return 5;
}

int return8()
{
    return 8;
}

int main()
{
    return5();
    return8();

    return 0;
}
```

Программа № 4:

```
#include <iostream>

void print0()
{
    std::cout << "0" << std::endl;
}

int main()
{
    std::cout << print0() << std::endl;

    return 0;
}
```

Программа № 5:

```
#include <iostream>

int getNumbers()
{
    return 6;
    return 8;
}

int main()
{
    std::cout << getNumbers() << std::endl;
    std::cout << getNumbers() << std::endl;

    return 0;
}
```

Программа № 6:

```
#include <iostream>

int return 6()
{
    return 6;
}

int main()
{
    std::cout << return 6() << std::endl;

    return 0;
}
```

Программа № 7:

```cpp
#include <iostream>

int return6()
{
    return 6;
}

int main()
{
    std::cout << return6 << std::endl;

    return 0;
}
```

Ответы

Ответ №1

Скомпилируется, результатом выполнения программы будет значение 13.

Ответ №2

Эта программа не скомпилируется. Вложенные функции запрещены.

Ответ №3

Эта программа скомпилируется, но не будет никакого вывода. Возвращаемые значения из функций не используются в main() и, таким образом, игнорируются.

Ответ №4

Эта программа не скомпилируется, так как тип возврата функции printO() — void, а мы отправляем несуществующее возвращаемое значение на вывод. Результат — ошибка компиляции.

Ответ №5

Результатом выполнения этой программы будет:

6

6

Оба раза, когда вызывается функция getNumbers(), возвращается значение 6. Компилятор, встречая первый return, сразу же выполняет возврат этого значения, и всё, что находится за первым return-ом, — игнорируется. Строка return 8; никогда не выполнится.

Ответ №6

Эта программа не скомпилируется из-за недопустимого имени функции.

Ответ №7

Эта программа скомпилируется, но функция не будет вызвана, так как в её вызове отсутствуют круглые скобки. Результат вывода зависит от компилятора.

Использование функций

Для того чтобы использовать функцию в C++, вы должны выполнить следующие шаги:

- предоставить определение функции;
- представить прототип функции;
- вызвать функцию.

Если вы планируете пользоваться библиотечной функций, то она уже определена и скомпилирована. К тому же вы можете, да и должны пользоваться стандартным библиотечным заголовочным файлом, чтобы предоставить своей программе доступ к прототипу. Все что вам остается — правильно вызвать эту функцию.

Когда вы создаете собственные функции, то должны самостоятельно обработать все три аспекта — определение, прототипирование и вызов. В листинге ниже демонстрируются все три шага на небольшом примере.

Использование функций

```
#include<iostream>

using namespace std;
void simple (); // прототип функции
int main ()
{

cout << "main () will call the simple () function: \n";
simple(); // вызов функции
cout << "main() is finished with the simple () function.\n";
// cin.get();
return 0;
}
// Определение функции
void simple ()
{
cout << "I'm but a simple function. \n";
}
```

Ниже показан вывод программы из листинга. Выполнение программы в main() останавливается, как только управление передается функции simple(). По завершении simple() выполнение программы возобновляется в функции main().

```
main () will call the simple () function:
I'm but a simple function.
main() is finished with the simple () function.
```

Прототипирование и вызов функции

Вы уже знакомы с тем, как вызываются функции, но, возможно, менее уверенно себя чувствуете в том, что касается их прототипирования, поскольку зачастую прототипы функций скрываются во включаемых (с помощью #include) файлах. В листинге ниже демонстрируется использование функций cheers () и cube (); обратите внимание на их прототипы.

```
#include <iostream>
void cheers(int); // прототип: нет значения возврата
double cube(double x); // прототип: возвращает double
int main()
{
using namespace std;
cheers(5); // вызовфункции
cout << "Give me a number: ";
double side;
cin >> side;
double volume = cube(side); // вызов функции
cout << "A " << side <<"-foot cube has a volume of ";
cout << volume << " cubic feet.\n";
cheers (cube(2)); // защита прототипа в действии
return 0;
}
void cheers(int n)
{
using namespace std;
for (int i = 0; i < n; i++)
cout << "Cheers! ";
cout << endl;
}
double cube(double x)
{
return x * x * x;
}
```

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

Программа из листинга помещает директиву using только в те функции, которые используют члены пространства имен std.

Обратите внимание, что main() вызывает функцию cheers() типа void с использованием имени функции и аргументов, за которыми следует точка с запятой: cheers (5);. Это пример оператора вызова функции. Но поскольку cube () возвращает значение, main () может применять его как часть оператора присваивания:

double volume = cube(side);

Прототип описывает интерфейс функции для компилятора. Это значит, что он сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции. Прототип функции является оператором, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой. Это, собственно, и делает программа из листинга с функцией cube ()

Однако прототип функции не требует предоставления имен переменных-параметров; достаточно списка типов. Программа из листинга строит прототип cheers (), используя только тип аргумента:

void cheers(int); // в прототипе можно опустить имена параметров

В общем случае в прототипе можно указывать или не указывать имена переменных в списке аргументов. Имена переменных в прототипе служат просто заполнителями, поэтому если даже они заданы, то не обязательно должны совпадать с именами в определении функции.

Прототипом функции в языке Си или C++ называется объявление функции, не содержащее тела функции, но указывающее имя функции, количество аргументов, типы аргументов и возвращаемый тип данных. В то время как определение функции описывает, что именно делает функция, прототип функции может восприниматься как описание её интерфейса.

В прототипе имена аргументов являются необязательными, тем не менее, необходимо указывать тип (например, указатель ли это или константный аргумент).

Кстати, описание функций можно помещать в заголовочный файл (.h/ .hpp)

В качестве примера, рассмотрим следующий прототип функции:

```
int foo(int n);
```

Этот прототип объявляет функцию с именем «foo», которая принимает один аргумент «n» целого типа и возвращает целое число. Определение функции может располагаться где угодно в программе, но объявление требуется только в случае её использования.

Прототип описывает интерфейс функции для компилятора. Это значит, что он сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции. Прототип функции является оператором, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой.

В общем случае в прототипе можно указывать или не указывать имена переменных в списке аргументов. Имена переменных в прототипе служат просто заполнителями, поэтому если даже они заданы, то не обязательно должны совпадать с именами в определении функции.

РЕКУРСИЯ

Реку́рсия — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя.

Рекурсивное изображение экрана:

Большая часть шуток о рекурсии касается бесконечной рекурсии, в которой нет условия выхода, например, известно высказывание: *«чтобы понять рекурсию, нужно сначала понять рекурсию»*.

Весьма популярна шутка о рекурсии, напоминающая словарную статью:

Рекурсия

см. рекурсия

РЕКУРСИЯ

Несколько рассказов Станислава Лема посвящены (возможным) казусам при бесконечной рекурсии.

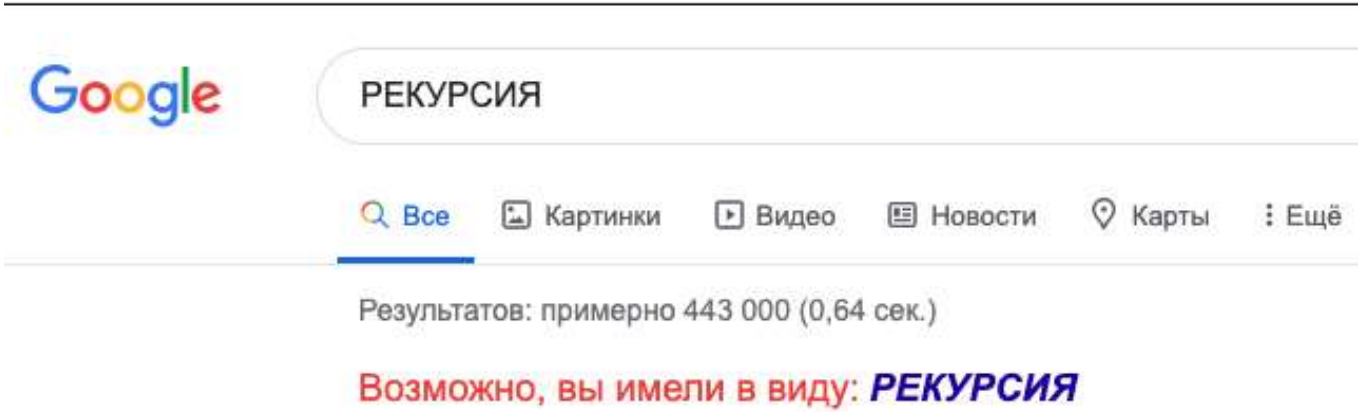
Рассказ из «Кибериады» о разумной машине, которая обладала достаточным умом и ленью, чтобы для решения поставленной задачи построить себе подобную и поручить решение ей. Итогом стала бесконечная рекурсия, когда каждая новая машина строила себе подобную и передавала задание ей.

Рассказ про Ийона Тихого «Путешествие четырнадцатое» из «Звёздных дневников Ийона Тихого», в котором герой последовательно переходит от статьи о сепульках к статье о сепуляции, оттуда к статье о сепулькариях, в которой снова стоит отсылка к статье «сепульки»:

Нашёл следующие краткие сведения:
«СЕПУЛЬКИ — важный элемент цивилизации ардритов (см.) с планеты Энтеропия (*см*.). *См*. СЕПУЛЬКАРИИ».
Я последовал этому совету и прочёл:
«СЕПУЛЬКАРИИ — устройства для сепуления (*см*.)».
Я поискал «Сепуление»; там значилось:
«СЕПУЛЕНИЕ — занятие ардритов (*см*.) с планеты Энтеропия (см.). *См*. СЕПУЛЬКИ».

Лем С. «Звёздные дневники Ийона Тихого. Путешествие четырнадцатое.»

И апофеозом идиотизма бесконечной рекурсии является шутка креативщиков из Гугла:



Короче, рекурсия – это «У попа была собака…»

РЕКУРСИЯ

Рекурсия - это такой способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.

Итерация - это способ организации обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.

Теорема. Произвольный алгоритм, реализованный в рекурсивной форме, может быть переписан в итерационной форме и наоборот.

Отличие рекурсии от итерации

Теперь нам нужны конкретные примеры из простой математики, чтобы можно было отличить итерацию от рекурсии.

Факториал числа. Факториалом целого неотрицательного числа *n* называется произведение всех натуральных чисел от 1 до *n* и обозначается *n*! Если *f*(*n*) = *n*!, то имеет место рекуррентное соотношение:

$$n! = \begin{cases} f(n) = n * f(n-1), \\ f(0) = 1 \end{cases}$$

Первое равенство описывает шаг рекурсии – метод вычисления *f*(*n*) через *f*(*n* – 1). Второе равенство указывает, когда при вычислении функции следует остановиться. Если его не задать, то функция должна будет работать бесконечно долго, хотя, на самом деле, не бесконечно долго, и даже не очень долго, так как стек не резиновый.

Например, значение *f*(3) можно вычислить следующим образом:

f(3) = 3 · *f*(2) = 3 · 2 · *f*(1) = 3 · 2 · 1 · *f*(0) = 3 · 2 · 1 · 1 = 6

Очевидно, что при вычислении *f*(*n*) следует совершить *n* рекурсивных вызовов.

Отличие рекурсии от итерации

Сравните алгоритмы рекурсивной и итерационной (циклической) реализации вычисления факториала, и вам все станет окончательно ясно.

рекурсивная реализация

```
int f(int n)
{
  if (!n) return 1;
  return n * f(n - 1);
}
```

циклическая реализация

```
int f(int n)
{
  int i, res = 1;
  for (i = 1; i <= n; i++) res = res
* i;
  return res;
}
```

Идея циклической реализации состоит в непосредственном вычислении факториала числа при помощи оператора цикла:

$$f(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Примечание. **!n** это равносильно выражению **n == 0**. Любое число равно false только в том случае, если оно равно 0.

Отличие рекурсии от итерации

Сумма цифр числа. Сумму цифр натурального числа *n* можно найти при помощи функции *f*(*n*), определенной следующим образом:

$$\begin{cases} f(n) = n \bmod 10 + f(n \operatorname{div} 10) \\ f(0) = 0 \end{cases}$$

Условие продолжения рекурсии: сумма цифр числа равна последней цифре плюс сумма цифр числа без последней цифры (числа, деленного нацело на 10).

Условие окончания рекурсии: если число равно 0, то сумма его цифр равна 0.

Например, сумма цифр числа 234 будет вычисляться следующим образом:

f(234) = 4 + *f*(23) = 4 + 3 + *f*(2) = 4 + 3 + 2 + *f*(0) = 4 + 3 + 2 + 0 = 9

Отличие рекурсии от итерации

рекурсивная реализация

```
int f(int n)
{
  if (!n) return 0;
  return n % 10 + f(n / 10);
}
```

циклическая реализация

```
int f(int n)
{
  int res = 0;
  for (n=1; n>0; n = n / 10) res = res + n % 10;
  return res;
}
```

Отбор в разведку. Из ***n*** солдат, выстроенных в шеренгу, требуется отобрать нескольких в разведку. Для совершения этого выполняется следующая операция: если солдат в шеренге больше, чем 3, то удаляются все солдаты, стоящие на четных позициях, или все солдаты, стоящие на нечетных позициях. Эта процедура повторяется до тех пор, пока в шеренге не останется 3 или менее солдат. Их и посылают в разведку. Вычислить количество способов, которыми таким образом могут быть сформированы группы разведчиков ровно из трех человек.

Вход. Количество солдат в шеренге ***n*** ($0 < n \leq 10^7$).

Выход. Количество способов, которыми можно отобрать солдат в разведку описанным выше способом.

Пример входа	***Пример выхода***
10	2
4	0

Решение. Обозначим через $f(n)$ количество способов, которыми можно сформировать группы разведчиков из n человек в шеренге. Поскольку нас интересуют только группы по три разведчика, то $f(1) = 0$, $f(2) = 0$, $f(3) = 1$. То есть из трех человек можно сформировать только одну группу, из одного или двух – ни одной.

Если n четное, то, применяя определенную в задаче операцию удаления солдат в шеренге, мы получим в качестве оставшихся либо $n / 2$ солдат, стоящих на четных позициях, либо $n / 2$ солдат, стоящих на нечетных позициях. То есть $f(n) = 2 \cdot f(n / 2)$ при четном n.

Если n нечетное, то после удаления останется либо $n / 2$ солдат, стоявших на четных позициях, либо $n / 2 + 1$ солдат, стоявших на нечетных позициях. Общее количество способов при нечетном n равно

$f(n) = f(n / 2) + f(n / 2 + 1)$.

Таким образом, получена рекуррентная формула для вычисления значения $f(n)$:

$f(n) = 2 \cdot f(n / 2)$, если n четное

$f(n) = f(n / 2) + f(n / 2 + 1)$, если n нечетное

$f(1) = 0, f(2) = 0, f(3) = 1$

Реализация функции f имеет вид:

```
int f(int n)
{
  if (n <= 2) return 0;
  if (n == 3) return 1;
  if (n % 2) return f(n / 2) + f(n / 2 + 1);
  return 2 * f(n / 2);
}
```

Функция C++ обладает интересной характеристикой — она может вызывать сама себя. (Однако, в отличие от C, в C++ функции main () не разрешено вызывать саму себя.) Эта возможность называется рекурсией. Рекурсия — важный инструмент в некоторых областях программированиях, таких как искусственный интеллект, но здесь мы дадим только поверхностные сведения о принципах ее работы.

Функция C++ может вызывать сама себя, но **функции main() не разрешено вызывать саму себя!**) Это, как вы понимаете, рекурсия.

Рекурсия с одиночным рекурсивным вызовом

Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и т.д., то получается бесконечная последовательность вызовов, если только код не включает в себе нечто, что позволит завершить эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов помещается внутрь оператора if. Например, рекурсивная функция типа void по имени recurs () может иметьследующую форму:

```
void recurs(список Аргументов)
{
операторы1
if (проверка)
recurs {аргументы)
операторы2
}
```

В какой-то ситуации проверка возвращает false, и цепочка вызовов прерывается.

Функция C++ может вызывать сама себя, но функции main() не разрешено вызывать саму себя.) Это, как вы понимаете, рекурсия.

Рекурсия с одиночным рекурсивным вызовом

Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и т.д., то получается бесконечная последовательность вызовов, если только код не включает в себя нечто, что позволит завершить эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов помещается внутрь оператора **if**. Например, рекурсивная функция типа void по имени recurs() может иметь следующую форму:

```
void recurs(список Аргументов)
{
операторы1
if (проверка)
recurs(аргументы)
операторы2
}
```

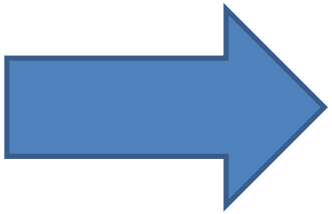
В какой-то ситуации проверка возвращает false, и цепочка вызовов прерывается.

Рекурсия

Рекурсивные вызовы порождают замечательную цепочку событий. До тех пор, пока условие оператора **if** остается истинным, каждый вызов recurs() выполняет операторы1 и затем вызывает новое воплощение recurs(), не достигая конструкции операторы2. Когда условие оператора **if** возвращает false, текущий вызов переходит к операторы2. Когда текущий вызов завершается, управление возвращается предыдущему экземпляру recurs(), который вызвал его. Затем этот экземпляр выполняет свой раздел операторы2 и прекращается, возвращая управление предшествующему вызову, и т.д. Таким образом, если происходит пять вложенных вызовов recurs(), то первый раздел операторы1 выполняется пять раз в том порядке, в котором произошли вызовы, а потом пять раз в обратном порядке выполняется раздел операторы2. После входа в пять уровней рекурсии программа должна пройти обратно эти же пять уровней. Код в листинге ниже демонстрирует описанное поведение.

```
#include<iostream>
void countdown(int n);
int main ()
{
countdown (4) ;  // вызов рекурсивной функции
return 0;
}
void countdown(int n)
{
using namespace std;
cout << "Counting down ... " <<n<<endl;
if (n > 0)
countdown(n-1); // функция вызывает сама себя
cout << n << " : Kaboom!\n";

}
```



```
Counting down ... 4
Counting down ... 3
Counting down ... 2
Counting down ... 1
Counting down ... 0
0 : Kaboom!
1 : Kaboom!
2 : Kaboom!
3 : Kaboom!
4 : Kaboom!
```

Обратите внимание, что каждый рекурсивный вызов создает собственный набор переменных, поэтому на момент пятого вызова она имеет пять отдельных переменных по имени n — каждая с собственным значением.

Рекурсия с множественными рекурсивными вызовами

Рекурсия, в частности, удобна в тех ситуациях, когда нужно вызывать повторяющееся разбиение задачи на две похожие подзадачи меньшего размера. Например, рассмотрим применение такого подхода для рисования линейки. Сначала нужно отметить два конца, найти середину и пометить ее. Затем необходимо применить ту же процедуру для левой половины линейки и правой ее половины. Если требуется больше частей, эта же процедура применяется для каждой из существующих частей. Такой рекурсивный подход иногда называют стратегией "разделяй и властвуй". В листинге ниже данный подход иллюстрируется на примере рекурсивной функции subdivide (). Она использует строку, изначально заполненную пробелами, за исключением символов @ на каждом конце. Затем главная программа запускает цикл из шести вызовов subdivide(), каждый раз увеличивая количество уровней рекурсии и печатая результирующую строку. Таким образом, каждая строка вывода представляет дополнительный уровень рекурсии.

Рекурсия с множественным рекурсивным вызовом

```cpp
#include <iostream>
const int Len = 66;
const int Divs = 6;
void subdivide (char ar[], int low, int high, int level);
int main ()
{
char ruler[Len];
int i;
for (i = 1; i<Len-2; i++)
ruler[i] = ' ';
ruler [Len - 1] = '\0';
int max = Len - 2;
int min = 0;
ruler [min] = ruler [max] = i;
std::cout << ruler << std::endl;
for (i = 1; i<= Divs; i++ )
{
subdivide(ruler,min,max, i) ;
std::cout << ruler << std::endl;
for (int j = 1; j<Len-2; j++)
ruler[j] = ' '; // очистка линейки
}
return 0;
}
void subdivide (char ar[], int low, int high, int level)
{
if (level == 0)
return;
int mid = (high + low) / 2;
ar[mid] = 'I';
subdivide (ar, low, mid, level - 1);
subdivide(ar, mid, high, level - 1);
}
```

```
@                                                               @
@                               I                               @
@               I               I               I               @
@       I       I       I       I       I       I       I       @
@   I   I   I   I   I   I   I   I   I   I   I   I   I   I   I   @
@ I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I @
@IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII@
```

Рекурсия с множественным рекурсивным вызовом. Ханойская башня

Примером рекурсии с множественным рекурсивным вызовом может служить задача «Ханойская башня».

Утверждается, что эту задачу сформулировали и решают до сих пор монахи каких-то монастырей Тибета. Задача состоит в том, чтобы пирамиду из колец (на манер детской игрушки), нанизанную на один из 3-х стержней, перенести на другой такой же стержень, придерживаясь строгих правил:

- пирамидка состоит из *n* колец разного размера, уложенных по убыванию диаметра колец одно на другое;
- перекладывать за одну операцию можно только одно кольцо с любого штыря на любой, но только при условии, что класть можно только меньшее кольцо сверху на большее, но никак не наоборот;
- нужно, в итоге, всю исходную пирамиду, лежащую на штыре № 1, переместить на штырь № 3, используя штырь № 2 как промежуточный.

Например, для 2-х колец результат получается такой вот последовательностью перекладываний: 1 => 2, 1 => 3, 2 => 3.

По преданию эту задачу по перекладыванию *n*=64 золотых дисков на алмазных стержнях решают тибетские монахи, и когда они её, наконец, решат, тогда и наступит конец света - Армагеддон в нашей западной нотации.

Решение здесь (если его таковым можно назвать) состоит в том, чтобы при необходимости переноса пирамиды из *n* колец с штыря с номером *from* на штырь с номером *to* последовательно сделать следующее:

- перенести (каким-то образом) меньшую пирамиду из *n*-1 колец временно на штырь с номером *temp*, чтобы не мешала;
- перенести оставшееся единственное нижнее (наибольшее) кольцо на результирующий штырь с номером *to*, после чего, точно так же как в первом пункте, водрузить пирамиду (*n*-1 колец) с номера *temp* поверх этого наибольшего кольца на штырь с номером *to*.

Здесь важно то, что мы не знаем, каким образом выполнить алгоритм, и не умеем выполнить 1-й и 3-й пункты нашей программы, но надеемся, что алгоритм будет рекурсивно раскручиваться по аналогии, то есть по тому же алгоритму, но для меньшего числа *n*-1 (основополагающий принцип рекурсии), пока *n* не станет равным 1, а там уже совсем просто.

Ханойская башня

И вот как разворачивается решение для различных *n*:

n=2

1 => 2 | 1 => 3 | 2 => 3 |

общее число перемещений 3

n=3

1 => 3 | 1 => 2 | 3 => 2 | 1 => 3 | 2 => 1 | 2 => 3 | 1 => 3 |

общее число перемещений 7

n=4

1 => 2 | 1 => 3 | 2 => 3 | 1 => 2 | 3 => 1 |

3 => 2 | 1 => 2 | 1 => 3 | 2 => 3 | 2 => 1 |

3 => 1 | 2 => 3 | 1 => 2 | 1 => 3 | 2 => 3 |

общее число перемещений 15

n=5

1 => 3 | 1 => 2 | 3 => 2 | 1 => 3 | 2 => 1 |

2 => 3 | 1 => 3 | 1 => 2 | 3 => 2 | 3 => 1 |

2 => 1 | 3 => 2 | 1 => 3 | 1 => 2 | 3 => 2 |

1 => 3 | 2 => 1 | 2 => 3 | 1 => 3 | 2 => 1 |

3 => 2 | 3 => 1 | 2 => 1 | 2 => 3 | 1 => 3 |

1 => 2 | 3 => 2 | 1 => 3 | 2 => 1 | 2 => 3 | 1 => 3 |

общее число перемещений 31

Только не спровоцируйте конец света!

Число перестановок:
для n=10 потребуется 1023 перестановки;
для любого n число перестановок равно 2^n–1, что представляет собой очень высокую степень роста вычислительной сложности задачи — экспоненциальную.

Примечание: решить эту задачу не рекурсивными методами - очень непростое занятие!

УКАЗАТЕЛИ НА ФУНКЦИИ

Функции, как и элементы данных, имеют адреса. Адрес функции — это адрес в памяти, где находится начало кода функции на машинном языке. Обычно пользователю ни к чему знать этот адрес, но это может быть полезно для программы. Например, можно написать функцию, которая принимает адрес другой функции в качестве аргумента. Это позволяет первой функции найти вторую и запустить ее. Такой подход сложнее, чем простой вызов второй функции из первой, но он открывает возможность передачи разных адресов функций в разные моменты времени. То есть первая функция может вызывать разные функции в разное время.

Проясним этот процесс на примере. Предположим, что требуется спроектировать функцию estimate (), которая оценивает затраты времени, необходимого для написания заданного количества строк кода, и вы хотите, чтобы этой функцией пользовались разные программисты. Часть кода estimate() будет одинакова для всех пользователей, но эта функция позволит каждому программисту применить собственный алгоритм оценки затрат времени. Механизм, используемый для обеспечения такой возможности, будет заключаться в передаче estimate() адреса конкретной функции, которая реализует алгоритм, выбранный данным программистом. Чтобы реализовать этот план, понадобится сделать следующее:

- получить адрес функции;
- объявить указатель на функцию;
- использовать указатель на функцию для ее вызова.

Получить адрес функции очень просто: вы просто используете **имя функции без скобок**. То есть, если имеется функция think(), то ее адрес записывается как think. Чтобы передать функцию в качестве аргумента, вы просто передаете ее имя. Удостоверьтесь в том, что понимаете разницу между адресом функции и передачей ее возвращаемого значения:

process(think); // передача адреса think() функции process()

thought(think()); // передача возвращаемого значения think() функции thought(). Вызов process() позволяет внутри этой функции вызвать функцию think(). Вызов thought() сначала вызывает функцию think() и затем передает возвращаемое ею значение функции thought().

Чтобы объявить указатель на тип данных, нужно явно задать тип, на который будет указывать этот указатель. Аналогично, указатель на функцию должен определять, на функцию какого типа он будет указывать. Это значит, что объявление должно идентифицировать тип возврата функции и ее сигнатуру (список аргументов). То есть объявление должно предоставлять ту же информацию о функции, которую предоставляет и ее прототип. Например, предположим, что одна из функций для оценки затрат времени имеет следующий прототип:

```
double pam(int); // прототип
```

Вот как должно выглядеть объявление соответствующего типа указателя:

```
double (*pf)(int); // pf указывает на функцию, которая принимает
                   // один аргумент типа int и возвращает тип double
```

Указатели на функции

Объявление требует скобок вокруг *pf, чтобы обеспечить правильный приоритет операций. **Скобки имеют более высокий приоритет, чем операция *, поэтому *pf(int) означает, что pf() — функция, которая возвращает указатель, в то время как (*pf)(int) означает, что pf — указатель на функцию:**

```
double(*pf)(int); // pf указывает на функцию, возвращающую double
double *pf(int); // pfO — функция, возвращающая указатель на double
```

После соответствующего объявления указателя pf ему можно присваивать адрес подходящей функции:

```
double pam(int);
double(*pf) (int) ;
pf = pam; // pf теперь указывает на функцию pam()
```

Обратите внимание, что функция pam() должна соответствовать pf как по типу возврата, так и по сигнатуре. Компилятор отклонит несоответствующие присваивания:

```
double ned(double);
int ted(int);
double (*pf) (int) ;
pf = ned; // неверно — несоответствие сигнатуры
pf = ted; // неверно — несоответствие типа возврата
```

Вернемся к упомянутой ранее функции estimate (). Предположим, что вы хотите передавать ей количество строк кода, которые нужно написать, и адрес алгоритма оценки — функции, подобной pam(). Тогда она должна иметь следующий прототип:

```
void estimate(int lines, double (*pf)(int));
```

Это объявление сообщает, что второй аргумент является указателем на функцию, принимающую аргумент int и возвращающую значение double. Чтобы заставить

estimate() использовать функцию pam(), вы передаете ей адрес pam:

```
estimate(50, pam); // вызов сообщает estimate(),
                   // что она должна использовать pam()
```

Очевидно, что вся сложность использования указателей на функцию заключается в написании прототипов, в то время как передавать адрес очень просто.

Теперь обратимся к завершающей части этого подхода — использованию указателя для вызова указываемой им функции. Ключ к этому находится в объявлении указателя. Вспомним, что там (*pf) играет ту же роль, что имя функции. Поэтому все, что потребуется сделать — использовать (*pf), как если бы это было имя функции:

```
double pam(int);
double (*pf)(int);
pf = pam; // pf теперь указывает на функцию pam()
double x = pam(4); // вызвать pam(), используя ее имя
double y = (*pf)(5); // вызвать pam(), используя указатель pf
```

В действительности C++ позволяет использовать pf, как если бы это было имя функции:

```
double y = pf(5); // также вызывает pam(), используя указатель pf
```

Первая форма вызова более неуклюжа, чем эта, но она напоминает о том, что код использует указатель на функцию.

Задача «Шарики»

А теперь, в качестве примера использования рекурсивного вызова функции, давайте разберем задачу, которая у многих вызывает затруднения.

Из урны с 10 пронумерованными шариками вынимают по одному шарику. Подсчитать общее количество ситуаций, когда номер хотя бы одного вынутого шарика совпадает с порядковым номером действия "вынимания", например, когда шарик № 3 будет вынут 3-им по порядку.

ЗАДАЧА «ШАРИКИ»

Чтобы понять, какие комбинации шариков надо учитывать, проведем эксперимент с небольшим количеством шариков, составив для них все возможные перестановки.

2 шарика: **1**, **2** (подходит, так как шарик № 1 вынут первым; и № 2 тоже, но это не важно)

2, 1 (нет) *Ответ*: 1

3 шарика: **1**, **2**, **3** (подходит, так как шарик № 1 вынут первым; и № 2 тоже, но это не важно)

1, 3, 2 (подходит, так как шарик № 1 вынут первым)

2, 1, **3** (подходит, так как шарик № 3 вынут третьим)

2, 3, 1 (нет)

3, 1, 2 (нет)

3, **2**, 1 (подходит, так как шарик № 2 вынут вторым) *Ответ*: 4

Один из возможных вариантов алгоритма решения задачи про шарики

1. Задать количество шариков *n*.
2. Создать массив пронумерованных шариков от 1 до *n*.
3. Целочисленная переменная ***i*** - номер шарика (от 1 до *n*), одновременно являющаяся счетчиком действий.
4. Создать функцию ***perestanovka*** от целочисленных *m* и *n*, которая генерирует перестановки, в зависимости от количества шариков (*n*) и в которой фигурирует номер очередного переставляемого шарика (*m*). В этой функции использовать условие: когда номер шага ***i*** равен номеру вынимаемого шарика *m*, учитывать очередную перестановку. Во всех остальных случаях менять местами элементы с номерами *i* и *m*, после чего вызывать функцию ***perestanovka*** со следующим значением шага и опять же менять местами элементы с номерами *i* и *m*.
5. Основная программа: присваивание шарикам порядковых номеров, вызов функции *perestanovka* с параметрами 1 (первый шаг) и *n* (количество шариков).

Вывод рекуррентной формулы

n-количество шариков	P=n!-число перестановок	k-искомый результат	Расчет результата
1	1	1	1
2	2	1	1
3	6	4	(1+1)*2=4
4	24	15	(4+1)*3=15
5	120	76	(15+4)*4=76
6	720	455	(76+15)*5=455
7	5040	3186	(455+76)*6=3186
8	40320	25480	(3186+455)*7=25487
9	362880	229384	(25487+3186)*8=229384
10	3628800	2293839	(229384+25487)*9=2293839
11	39916800	25232230	(2293839+229384)*10=25232230
12	479001600	302786759	(25232230+2293839)*11=302786759
13	6227020800		

$$k_n = (k_{n-1} + k_{n-2})(n-1)$$

Мы с вами изучаем не комбинаторику, а программирование! Эта формула годится для проверки полученного программным способом результата.

Генератор комбинаторных перестановок

```
void generate (int t) // Создает все перестановки шариков, число которых равно t
  {
    if (t==n-1)
      {                                  //Вывод очередной перестановки
        for (int i=0;i<n;++i)
              cout<<a[i]<< " ";
              cout<<endl;
            }
          else
            {
       for (int j=t;j<n;++j)
         {                               //Запускаем процесс обмена
            swap(a[t],a[j]);             //a[t] со всеми последующими
            t++;
            generate(t);                 //Рекурсивный вызов
            t--;
            swap(a[t],a[j]);
         }
      }
  }
```

ЗАВЕРШИТЬ РАБОТУ ПРЕДЛАГАЮ ВАМ САМИМ!

Литература

1. Материалы из открытого университета INTUIT.RU
Алгоритмизация. Введение в язык программирования C++ [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/16740/1301/info
2. Уроки программирования на языке C++. [Электронный ресурс]. Режим доступа: https://ravesli.com/uroki-cpp/#toc-0