Лекция 4

Функции

Функция - отдельная часть кода, которую можно использовать для выполнения чего-либо.

Функцию идентифицирует по ее имени.

На имя функции распространяются те же ограничения, что и на имя переменной. Кроме того, не может быть переменной и функции с одинаковым именем.

Функция способна изменять свое поведение с помощью параметров.

Параметры могут влиять на то, что вычисляется или что выполняется внутри функции.

При необходимости функция может изменять значения своих параметров.

Вызов функции - это специальная инструкция на языке C++.

В вызове должно быть указано:

-имя вызываемой функции (всегда);

-параметры, которые должна использовать функция (при необходимости);

-результат функции (если он имеется) (как использовать).

Функции можно разделить на две группы:

-функции, написанные другими – предопределенные, библиотечные;

-функции, написанные вами

Функции позволяют разработчикам разделить проблему (а также код) на более мелкие части. Меньший код легче писать, тестировать, поддерживать и понимать.

Наличие набора хорошо написанных и хорошо протестированных функций позволяет разработчику создавать код «с использованием готовых блоков».

Два подхода при декомпозиции:

- подход " сверху вниз - top-down " (сначала определяют наиболее общие функции, а затем делят их на более простые и специализированные);

- подход "снизу вверх - bottom-up " (сначала создают узко определенные и узкоспециализированные функций, а затем объединяют их в более сложные структуры).

Каждая функция характеризуется, имеет:

- имя

- параметры

- тип результата

Часть кода, которая определяет все эти элементы, - объявление функции или прототип функции.

```
#include <iostream>

using namespace std;

float square(float x)    // определение функции
```

```
{
  float result = x * x;
  return result;
}
int main()
{
 float arg = 2.0;
 cout << "The second power of " << arg << " is " << square(arg) << endl;
}
The second power of 2 is 4
```

Объявление функции с телом функции образует определение функции.

Объявление функции предназначено для информирования компилятора об имени функции, ее типе возвращаемого значения и типе параметров (если таковые имеются).

Общая форма объявления (прототипа) функции :

```
return_type function_name(parameter_list);
```

В более старых стандартах C++ использовалась особая форма подчеркивания того факта, что функция не имеет параметров, путем помещения слова void в круглые скобки, например:

```
 void fun(void);
```

Пусть нужна функция с именем fun , которая возвращает int в результате и имеет один параметр типа int.

Два эквивалентных объявления:

```
int func(int number);

int func(int);
```

Не разрешается объявлять более одного имени параметра одновременно.

Можно объявлять переменные: int x, y;

Нельзя: void fun(int x, y);

Должны использовать подробную форму при объявлении двух (или более) параметров функции: void fun(int x, int y);

Определение функции определяет код, который должен выполняться при каждом вызове функции. Он отличается от объявления тем, что точка с запятой заменяется телом, содержащим последовательность объявлений и/или инструкций.

Если return_type является void, тело может не содержать инструкции return , но если она все равно используется, она должна иметь форму: return;

Предполагается, что оператор return неявно выполняется в теле функции void непосредственно перед закрывающей скобкой.

Если тип return_type не является void , тело должно содержать по крайней мере один оператор return , указывающий значение результата функции. Нельзя выйти из тела функции, не указав значение результата.

Любое значение, которое вы собираетесь возвращать в теле функции, должно иметь тип, совместимый с типом, указанным в качестве типа функции.

Можно использовать столько операторов return , сколько нужно для эффективной реализации алгоритма.

```
#include <iostream>
using namespace std;
float fahrenheit_to_celsius(float temp)
{
    return ((temp - 32.0) * 5.0) / 9.0;
}
void test_the_function(float temp)
{
    cout << "Fahrenheit " << temp << " corresponds to ";
    cout << fahrenheit_to_celsius(temp) << " Centigrade" << endl;
}
int main()
{
    test_the_function(32.0);
    test_the_function(212.0);
    test_the_function(451.0);
}
```

Результат:

```
Fahrenheit 32 corresponds to 0 Centigrade
Fahrenheit 212 corresponds to 100 Centigrade
Fahrenheit 451 corresponds to 232.778 Centigrade
```

Функция может:

- возвращать значение, если перед именем функции указано имя типа, не являющееся void;

-ничего не возвращать, когда ключевое слово void находится перед именем функции;

Эти два вида функций различаются тем, как они используют оператор return , но есть и другое различие, касающееся вызовов.

Пусть есть две функции:

```
void void_function(int par){...; return;}
```

```
int non_void_function(int par){...; return par * par;}
```

Форма вызова function:

```
void_function(2);
```

Функция non_void_function может быть вызвана двумя способами:

```
value = non_void_function(2);
```

```
non_void_function(2);
```

Это означает, что результат любой непустой функции может быть присвоен переменной или использован любым другим способом, или может быть просто забыт сразу после возврата из функции.

- Передача данных в функцию с использованием аргументов, значения которых присваиваются параметрам функции;

- Передача данных из функции с использованием результата функции; таким способом может быть передано только одно значение, так как

синтаксис оператора return позволяет указать только одно значение; но это ограничение можно обойти, например, когда результатом функции является структура.

Переменная, определенная внутри тела функции, не может быть ни использована, ни доступна извне функции. Такие переменные существуют только при выполнении функции и исчезают, когда функция завершает свое выполнение, они не могут использоваться для хранения значения между вызовами функций.

Такие переменные - локальные переменные.

Глобальные переменные объявляются вне любой функции и доступны для всех функций, объявленных в одном исходном файле.

! Объявление переменной должно предшествовать определению функции, чтобы быть распознаваемым функцией.

Глобальные переменные позволяют функциям получать и предоставлять данные любого рода. Если функция изменяет любую глобальную переменную, которая не использует какой-либо другой механизм передачи данных, говорят, что эта функция имеет побочный эффект.

Побочные эффекты, хотя иногда и полезны, не рекомендуются и считаются признаком плохого стиля программирования, поскольку они затрудняют понимание кода.

```
#include <iostream>
using namespace std;
int globvar = 0;
void func()
{
      cout << "Thank you for invoking me :)" << endl;
```

```
      globvar++;
}
int main()
{
      for (int i = 0; i < 5; i++)
            func();
      cout << endl << "The function was happy "
            << globvar   << " times" << endl;
}
```

Передача параметров по значению

```
#include <iostream>
using namespace std;
void can_i_change_my_argument (int param)
{
     cout << "----------" << endl;
     cout << "I have got: " << param << endl;
     param++;
     cout << "I'm about to give back: " << param << endl;
     cout << "----------" << endl;
}
int main()
{
     int var = 1;
     cout << "var = " << var << endl;
     can_i_change_my_argument(var);
     cout << "var = " << var << endl;
}
```

Результат:

```
var = 1

----------

I have got: 1

I'm about to give back: 2

----------

var = 1
```

Передача параметров по ссылке

```
#include <iostream>
using namespace std;
void can_i_change_my_argument (int& param)
{
     cout << "----------" << endl;
     cout << "I have got: " << param << endl;
     param++;
     cout << "I'm about to give back: " << param << endl;
     cout << "----------" << endl;
}
int main()
{
     int var = 1;
     cout << "var = " << var << endl;
     can_i_change_my_argument(var);
     cout << "var = " << var << endl;
}
```

Результат:

```
var = 1
----------
I have got: 1
I'm about to give back: 2
----------
var = 1
```

Важное отличие в передаче параметров, которое радикально меняет поведение функции.

Разница в том, что после типа параметров ставится знак &.

• тип name – параметр name передается по значению

• тип &name – параметр name передается по ссылке

Параметр передается по значению - значение параметра не заменяет значение аргумента при возврате из функции

Параметр передается по ссылке - параметр является просто синонимом аргумента. Каждое изменение, внесенное в параметр, немедленно влияет на связанный аргумент.

Выбор метода передачи для каждого параметра - делаем индивидуально.

Можно смешивать параметры обоих типов.

Используйте “передача по значению”, если вам не нужно делиться результатами функции, используя значения параметра.

Используйте “передача по ссылке” во всех остальных случаях.

Выбор метода передачи для каждого параметра - делаем индивидуально.

Можно смешивать параметры обоих типов.

Используйте “передача по значению”, если вам не нужно делиться результатами функции, используя значения параметра.

Используйте “передача по ссылке” во всех остальных случаях.

```
#include <iostream>
```

```
using namespace std;
void mixed_styles (int bval, int & bref) {
  bref = bval + 1;
}
int main (void) {
  int var1 = 1, var2;
  mixed_styles(var1, var2);
  cout << "var1 = " << var1 << ", var2 = " << var2 << endl;
  return 0;
}
```

Результат:

```
var1 = 1, var2 = 2
```

Метод “передачи по ссылке” имеет одно важное и очевидное ограничение.

Если параметр объявлен как переданный по ссылке (поэтому ему предшествует знак &), его соответствующий аргумент должен быть переменной.

Аргумент, ссылающийся на параметр “переданный по значению”, может быть выражением, поэтому можно использовать не только переменную, но и литерал или результат вызова функции.

Это ограничение “очевидно”, потому что функция не может поместить значение во что-то другое, кроме переменной. Он не может присвоить новое значение литералу или заставить выражение изменить его результат.

```
#include <iostream>
using namespace std;
int fun(int n)
```

```
{
    return 2 * n;
}
void by_val(int parameter)
{
    parameter++;
}
void by_ref(int& parameter)
{
    parameter++;
}

int main()
{
    int var = 1;
    by_val(var);
    by_ref(var);
    cout << "var = " << var << endl;
}
```

Разрешены вызовы:

```
by_val(var);
by_val(var + 2);
```

```
by_val(var * fun(1));
```

Остальные – ошибка компиляции

Можно ли использовать “передачу по значению” и иметь возможность распространять значение вне функции? – «Да».

Но это не тот способ, который рекомендуют делать. Этот метод унаследован от ЯП С и был единственным доступным способом выполнения двусторонней связи на основе параметров.

Идея - на передаче указателя на значение, а не на само значение.

Если объявить функцию с прототипом, например:

```
void by_ptr(int* ptr);
```

то вы разрешаете функции обрабатывать адреса, указывающие на значения int, следовательно, даете функции возможность изменять значения, на которые указывает параметр.

```
#include <iostream>
using namespace std;
void by_ptr(int* ptr)
{
     *ptr = *ptr + 1;
}
int main()
{
     int variable = 1;
     int *pointer = &variable;
     by_ptr(pointer);
     cout << "variable = " << variable << endl;
}
```

Функция by_ptr принимает один параметр, который является указателем, и обращается к значению, на которое указывает указатель, с помощью оператора

* (разыменования).

Напоминание: если p является указателем на значение, то *p представляет само значение.

По сути, функция `by_ptr()` изменяет переменную, даже не зная о ее существовании.

Результат:

```
variable = 2
```

```
#include <iostream>

using namespace std;

void new_greet(string greet, int repeats)

{

    for (int i = 0; i < repeats; i++)

    cout << greet << endl;

}

int main()

{

  new_greet("Hi !", 3);

}
```

Результат:

```
Hi !

Hi !

Hi !
```

Программа выдает любое приветствие с количеством повторений более 1 раза.

```
#include <iostream>

using namespace std;

void new_greet(string greet, int repeats = 1)

{
```

```
    for (int i = 0; i < repeats; i++)
    cout << greet << endl;
}
int main()
{
    new_greet("Hello", 2);
    new_greet("Good morning");
    new_greet("Hi", 1);t("Hi", 1);
}
```

Результат:

```
Hello
Hello
Good morning
Hi
```

Явно указанное значение параметра делает недействительным значение по умолчанию.

```
#include <iostream>
using namespace std;
void new_greet(string greet = "Good morning", int repeats =
1)
{
     for (int i = 0; i < repeats; i++)
     cout << greet << endl;
}
int main(void) {
     new_greet("Hello", 2);
     new_greet("Hi");
     new_greet();
}
```

Результат:

```
Hello
Hello
Hi
Good morning
```

Ограничения:

- порядок параметров важен;

- параметрами по умолчанию располагаются после остальных параметров, в противном случае компилятор не сможет их идентифицировать;

- если объявлено несколько параметров со значением по умолчанию, и в вызове указан хотя бы один аргумент, то аргументы присваиваются их аналогам параметров в том же порядке, в котором они перечислены в объявлении функции; это означает, что не разрешается использовать значение по умолчанию для первого параметра и указывать явное значение для второго.

Код с точки зрения компилятора

Считывает код function, переводит его в машинный код и сохраняет переведенный код в отдельном месте в памяти, но код нельзя использовать “как есть” без дополнительных шагов.

Код каждой функции должен быть дополнен двумя важными элементами: прологом и эпилогом.

Пролог и эпилог

Рисунок 4.1 иллюстрирует поток управления во время одного вызова функции из предыдущего примера.

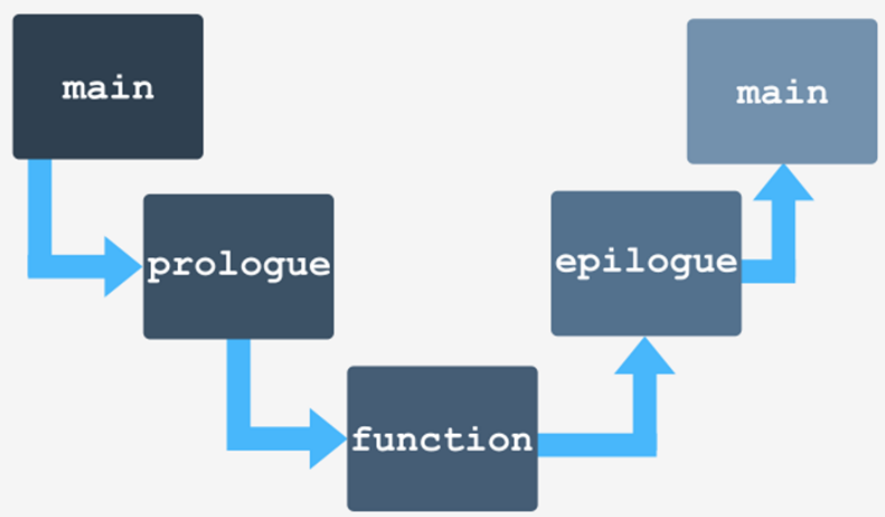

Рисунок 4.1. Поток управления

Такой подход – ряд преимуществ. Код функции, пролог и эпилог занимают одно и то же место в памяти независимо от того, сколько раз вызывается функция. Это означает, что такой вызов экономит память и делает программу более компактной. Но...

Один из парадоксов компьютерного программирования: когда код компактен, он не может быть быстрым одновременно; и наоборот, когда код быстр, он не может быть компактным. Это не научный закон, но в данном случае правило работает очень хорошо.

Пусть программа будет вызывать функцию сотни или тысячи раз. Это означает, что придется заплатить высокую цену (в смысле времени) за все передачи контроля и исполнения пролога/эпилога. Цена выше, когда функция короткая (т.е. короче, чем пролог и эпилог).

Этот пример показывает, что иногда, в четко определенных случаях, было бы лучше избегать цепочки пролог-функция-эпилог и вставлять код функции непосредственно в код вызывающего.

Встроенные функции

Когда функция короткая и быстрая, и ожидаем, что функция будет вызываться очень часто, лучше (и эффективнее по времени) писать код функции при каждом вызове.

Но … общий размер кода будет значительно больше, чем раньше.

Такая компиляции вызовов функций называется подстановкой функций.

Функция, скомпилированная подобным образом, называется встроенной функцией.

Чтобы функция была скомпилирована и вызвана как встроенная функция, необходимо перед объявлением функции - ключевое слово inline.

Не имеет значения, помещено ли встроенное ключевое слово перед или после имени типа; обе строки синтаксически правильны:

```
inline int function (int parameter)

int inline function (int parameter)
```

Если нужно использовать объявление и определение для одной и той же функции, не имеет значения, где разместили ключевое слово inline.

Допустимо: использовать его в объявлении и опустить в определении; использовать его в определении и опустить в объявлении; использовать его в обоих местах.

```
#include <iostream>
using namespace std;
inline int function(int parameter)
{
     return parameter * 2;
}
int main()
{
     int var = 1;
     var = function(var);
     var = function(var);
     var = function(var);
     cout << var << endl;
}
```

Подставляемые или встраиваемые (inline) функции – это функции, код которых вставляется компилятором непосредственно на место вызова, вместо передачи управления единственному экземпляру функции.

Если функция является подставляемой, компилятор не создает данную функцию в памяти, а копирует ее строки непосредственно в код программы по месту вызова. Это равносильно вписыванию в программе соответствующих блоков вместо вызовов функций.

Ключевое слово inline определяет для функции так называемое внутреннее связывание - компилятор вместо вызова функции подставляет команды ее кода. Подставляемые функции используют, если тело функции состоит из нескольких операторов.

Этот подход позволяет увеличить скорость выполнения программы, так как из программы исключаются команды микропроцессора, требующиеся для передачи аргументов и вызова функции.

Пример:

```
/*функция возвращает расстояние от точки с координатами(x1,y1) до точки с координатами (x2,y2)*/

inline float Line(float x1,float y1,float x2, float y2){

  return sqrt(pow(x1-x2,2)+pow(y1-y2,2));

 }
```

Использование подставляемых функций не всегда приводит к положительному эффекту.

Если такая функция вызывается в программном коде несколько раз, то во время компиляции в программу будет вставлено столько же копий этой функции, сколько ее вызовов.

Причины, по которым функция inline будет трактоваться как обычная не подставляемая функция:

- подставляемая функция является рекурсивной;
- функции, которые вызываются более одного раза в выражении;
- функции, содержащие циклы, переключатели и операторы переходов;
- функции, которые имеют слишком большой размер, чтобы сделать подстановку.

Ограничения на выполнение подстановки в основном зависят от реализации.

Если для функции inline компилятор не может выполнить подстановку из-за контекста, в который помещено обращение к ней, то функция считается статической и выдается предупреждающее сообщение.

Другая особенность inline функций - невозможность их изменения без перекомпиляции всех частей программы, в которых эти функции вызываются.

Перегрузка функций

При определении функций необходимо указывать тип возвращаемого функцией значения, количество параметров и тип каждого из них.

Если была написана функция с именем add_values, которая работала с двумя целыми значениями, а в программе необходимо использовать подобную функцию для передачи трех целых значений, то следовало бы создать функцию с другим именем.

Например, add_two_values и add_three_values.

Аналогично, если необходимо использовать подобную функцию для работы со значениями float, то нужна еще одна функция с еще одним именем.

Чтобы избежать дублирования функции, C++ позволяет определять несколько функций с одним и тем же именем.

В процессе компиляции C++ принимает во внимание количество аргументов, используемых каждой функцией, и затем вызывает именно требуемую функцию.

Предоставление компилятору выбора среди нескольких функций называется перегрузкой.

Для разных задач требуются разные инструменты. Хотим, чтобы разные инструменты с одинаковым названием использовались для разных целей.

Функция для нахождения наибольшего из двух и трех чисел float.

```
float max(float a, float b)
{
    if (a > b)
    return a;
    else
    return b;
}
```

Механизм, позволяющий иметь более одной функции с определенным именем - перегрузка (одно и то же имя перегружено разными значениями).

Важное: все перегруженные функции должны быть четко различимы для компилятора. Компилятор не может «думать», какой из перегруженных вариантов необходимо использовать в коде.

В примерах выбор прост: если вызов содержит 3 аргумента, выбирается второй вариант, если 2 - первый. Любой другой вариант вызова считается ошибкой.

При выборе целевой функции компилятор принимает во внимание :

- количество аргументов: например, если есть три перегруженные функции (с 2, 3 и 4-мя параметрами), и вызов указывает 2 аргумента, в качестве цели может использоваться только первая из функций (эта функция называется "лучшим кандидатом").

- типы аргументов: если существует более одной функции с одинаковым количеством параметров, целевая функция выбирается на основе соответствия типов параметров

Перегрузка функций – это создание нескольких функций с одним именем, но с разными параметрами. Под разными параметрами понимают, что должно быть разным количество аргументов функции и/или их тип.

Перегрузка функций позволяет определять несколько функций с одним и тем же именем и типом возвращаемого значения.

Перегрузка функций также называется полиморфизмом функций. "Поли" - много, "морфе" – форма, то есть это функция, отличающаяся многообразием форм.

Под полиморфизмом функции понимают существование в программе нескольких перегруженных версий функции, имеющих разные значения. Изменяя количество или тип параметров, можно присвоить двум или нескольким функциям одно и тоже имя. При этом никакой путаницы не будет, поскольку нужная функция определяется по совпадению используемых параметров.

Это позволяет создавать функцию, которая сможет работать с целочисленными, вещественными значениями или значениями других типов без необходимости создавать отдельные имена для каждой функции.

Примечание: тип возвращаемого значения return не принимается во внимание, когда компилятор ищет наилучшего кандидата для определенного вызова.

В этом смысле две функции, которые отличаются только типом возвращаемого значения, неразличимы для компилятора.

Количество различных типов данных в C++ значительно, поэтому механизм, отвечающий за поиск наилучшего кандидата, должен использовать некоторые упрощения, чтобы не создавать отдельную функцию для каждого отдельного типа данных.

Какая из двух перегруженных функций является лучшим кандидатом для вызова?

```
void play_with_number (int x)
```

```
{
    ...
}
void play_with_number (float x)
{
    ...
}
play_with_number (1);
```

Ответ: первая, с объявлением параметра int x.

Изменим пример. Ответ – нет подходящего кандидата для вызова – почему?

```
void play_with_number (int x)
{
    ...
}
void play_with_number (float x)
{
    ...
}
play_with_number(1.);
```

Литерал 1. – это не тип float, а тип double. Компилятор C++ пытается «продвинуть» типы, если нет точного соответствия (float не является double). «Продвижение» типов - от менее точного к более точному, а не наоборот.

Любой float может быть повышен до double, но double не может быть понижен до float .

Ошибка компиляции (компилятор сообщит что он не может найти лучшего кандидата).

Есть два варианта:

• можно написать третий экземпляр функции play_with_number() с одним параметром типа double;

• можно убедить компилятор, что литерал имеет тип float; добавив суффикс f к числу:

```
play_with_number(1.f);
```

Рекурсивная функция

Рекурсия в широком смысле – это определение объекта посредством ссылки на себя.

Рекурсия в программировании – это пошаговое разбиение задачи на подзадачи, подобные исходной.

Рекурсивный алгоритм – это алгоритм, в определении которого содержится прямой или косвенный вызов этого же алгоритма.

В ЯП процедурной парадигмы предусмотрено использование рекурсивных функций в решении задач.

Функция называется рекурсивной, если в своем теле она содержит обращение к самой себе с измененным набором параметров.

При этом количество обращений конечно, так как в итоге решение сводится к базовому случаю, когда ответ очевиден.

Пример. В арифметической прогрессии найти an, если известны

a1 = -2.5, d =0.4, не используя формулу n -го члена прогрессии.

По определению: an=an-1+d, an-1=an-2+d, an-2=an-3+d,... a2=a1+d.

Нахождение an для номера n сводится к решению аналогичной задачи, но только для номера n -1, что сводится к решению для номера n -2, и т.д., пока не будет достигнут номер 1 (значение a1 дано по условию задачи).

```
float arifm (int n, float a, float d) {
 if (n<1) return 0;   // для неположительных номеров
 if (n==1) return a;   // базовый случай: n=1
  return arifm(n-1,a,d)+d; // общий случай
}
```

В рекурсивных функциях несколько раз используется return.

В базовом случае возвращается конкретный результат (в примере – значение a), общий случай предусматривает вызов функцией себя же, но с меняющимися значениями отдельных параметров (в примере изменяется только номер члена последовательности, не меняются разность и первый член прогрессии).

В программировании выделяют прямую и косвенную рекурсию.

Прямая рекурсия предусматривает непосредственное обращение рекурсивной функции к себе, но с иным набором входных данных.

Косвенная (взаимная) рекурсия - последовательность взаимных вызовов нескольких функций, организованная в виде циклического замыкания на тело первоначальной функции, но с иным набором параметров.

Этапы решения задач рекурсивными методами:

- параметризация – выделяют параметры, которые используются для описания условия задачи, а затем в решении;

- база рекурсии – определяют тривиальный случай, при котором решение очевидно и не требуется обращение функции к себе;

- декомпозиция – выражают общий случай через более простые подзадачи с измененными параметрами.

Целесообразность применения рекурсии в программировании обусловлена спецификой задач, в постановке которых указывается на возможность сведения задачи к подзадачам, аналогичным самой задаче.

Эффективность рекурсивного или итерационного способов решения одной и той же задачи определяется в ходе анализа работоспособности программы на различных наборах данных.

Рекурсия не является универсальным способом в программировании. Ее следует рассматривать как альтернативный вариант при разработке алгоритмов решения задач.

Область памяти для хранения всех промежуточных значений локальных переменных при каждом рекурсивном обращении, образует рекурсивный стек.

Для каждого текущего обращения формируется локальный слой данных стека.

Завершение вычислений происходит посредством восстановления значений данных каждого слоя в порядке, обратном рекурсивным обращениям.

Количество рекурсивных обращений ограничено размером области памяти, выделяемой под программный код.

При заполнении всей предоставленной области памяти попытка вызова следующего рекурсивного обращения приводит к ошибке переполнения стека.

ПРИМЕР. Нахождение факториала целого неотрицательного числа n!

Параметризация: n – неотрицательное целое число.

База рекурсии: для n =0 факториал равен 1.

Декомпозиция: n!=(n-1)!*n.

```
long factor (int n) {
 if (n<0) return 0;   // для отрицательных чисел
 if (n==0) return 1;   // базовый случай: n=0
 return factor(n-1)*n;  // общий случай (декомпозиция)
 }
```

Есть задачи, для которых можно предложить рекурсивные алгоритмы решения, в то время как итерационные алгоритмы были бы сложными и искусственными.

ПРИМЕР. Задача Ханойская башня

Ханойская башня - одна из популярных головоломок XIX века. Даны три стержня, на один из которых нанизаны n колец, кольца отличаются размером и лежат меньшее на большем. Задача - перенести пирамиду из n колец за наименьшее число ходов с одного стержня на другой. За один раз разрешается переносить только одно кольцо, нельзя класть большее кольцо на меньшее.

Существует древнеиндийская легенда, согласно которой в городе Бенаресе под куполом главного храма, в месте, где находится центр Земли, на бронзовой площадке стоят три алмазных стержня. В день сотворения мира на один из этих стержней было надето 64 кольца. Бог поручил жрецам перенести кольца с одного стержня на другой, используя третий в качестве вспомогательного. Жрецы обязаны соблюдать условия:

- переносить за один раз только одно кольцо;
- кольцо можно класть только на кольцо большего размера или на пустой стержень.

Согласно легенде, когда, соблюдая все условия, жрецы перенесут все 64 кольца, наступит конец света.

Для 64 колец это = 18 446 744 073 709 551 615 перекладываний, если учесть скорость одно перекладывание в секунду, получится ~ 584 542 046 091 лет, то есть апокалипсис наступит нескоро.

Пример: перекладывание 7 колец со стержня A на B через вспомогательный C.

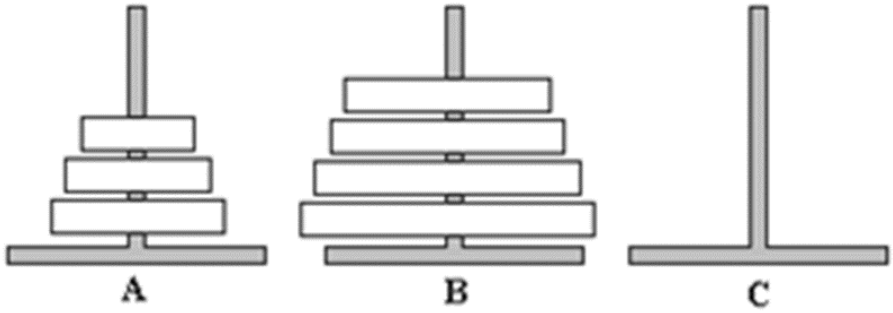

Рис. 4.2. Ханойская башня

Кольцо со стержня A можно перенести на стержень B или C, кольцо со стержня B можно перенести на стержень C, нельзя перенести его на стержень A.

Задача: определить последовательность минимальной длины переноса колец.

Решение задачи - последовательность допустимых переносов вида:

A->B, A->C, B->A, B->C, C->A, C->B

Если кольцо всего одно, то задача решается за один перенос: A->B

Для перемещения двух колец требуется три действия: A->C, A->B, C->B

Решение задачи для трех колец содержит семь действий, для четырех – 15.

Рекурсивная функция для произвольного числа колец.

Параметризация. Функция имеет 4 параметра: число переносимых колец; стержень, на который первоначально нанизаны кольца; стержень, на который требуется перенести кольца; вспомогательный стержень.

База рекурсии. Перенос одного стержня.

Декомпозиция. Последовательность переноса колец изображена на рисунке.

Алгоритм переноса n колец со стержня A на стержень B, используя стержень C:

- перенести n –1 кольцо со стержня A на C, используя стержень B в качестве вспомогательного стержня;

- перенести последнее кольцо со стержня A на стержень B;

- перенести n –1 кольцо со стержня C на B, используя стержень A в качестве вспомогательного стержня.

При переносе n –1 кольца можно воспользоваться тем же алгоритмом, на нижнее кольцо с самым большим диаметром можно не обращать внимания.

Перенос одного кольца в программе выражается в том, что выводится соответствующий ход.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int hanoj(int n, char A, char B, char C);
int main( ){
 char x='A',y='B',z='C';
 int k,h;
printf("\nВведите количество колец: ");
 scanf("%d",&k);
 h=hanoj(k,x,z,y);
 printf("\nКоличество перекладываний = %d",h);
```

```
 system("pause");
 return 0; }
//Функции перемещения колец с A на C через B
int hanoj(int n, char A, char B, char C){
int num;
if (n == 1) {printf("\n %c -> %c", A, C); num = 1;}
 else {
num=hanoj(n-1, A, C, B);
 printf("\n %c -> %c", A, C);
num++;
 num+=hanoj(n-1, B, A, C);
 }
 return num;
}
```

Оператор с тремя аргументами

expression1 ? expression2 : expression3

Работа оператора:

• вычисляет значение expression1

• если вычисленное значение не равно нулю (также включает значение true), оператор возвращает значение expression2

• если вычисленное значение = 0 , оператор возвращает значение expression3

Результат выражения: `i = i > 0 ? 1 : 0;` переменной i будет присвоено значение 1, если ее предыдущее значение было >0, и 0 в противном случае.

Можно добиться того же с помощью условного оператора:

```
if(i > 0)
   i = 1;
  else
    i = 0;
```