Функции

Определение функции

- Функция – это самостоятельная единица программы, спроектированная для реализации конкретной задачи
- В языке C/C++ функция – строительный блок всех программ. Вызов функций приводит к выполнению некоторых действий

Почему нужны функции?

- Они избавляют от повторного программирования. Если конкретную задачу в программе необходимо выполнить несколько раз, достаточно написать соответствующую функцию только один раз, а затем вызывать ее всегда, когда требуется
- Можно применять одну функцию в различных программах

Общий вид функции

Общий вид определения функции выглядит следующим образом:

```
тип_возвращаемого_значения имя_функции(список_параметров)
{
  тело функции
}
```

тип_возвращаемого_значения определяет тип переменной, которую возвращает функция в качестве своего значения. Функция может возвращать переменные любого типа кроме массива, но может возвращать указатель на массив

имя_функции – правильный идентификатор

список_параметров – список типов переменных и их имен, разделенных запятой

Общий вид списка формальных параметров функции выглядит следующим образом:

```
имя_функции(тип имя_переменной1, …, тип имя_переменнойN)
```

Примеры

Функция с двумя аргументами целого типа, возвращающая целое число

```
int func(int a, int b) // правильное объявление параметров функции
  {
    тело функции
  }

int func(int a, b) // неправильное объявление параметров функции
  {
    тело функции
  }
```

Прототип функции

- В современных, правильно написанных программах на языке С каждую функцию перед использованием необходимо объявлять. Обычно это делается с помощью прототипа функции. В первоначальном варианте языка С прототипов не было; но они были введены уже в Стандарт С89. Хотя прототипы формально не требуются, но их использование очень желательно. (Впрочем, в С++ прототипы обязательны!)
- Прототипы дают компилятору возможность тщательнее выполнять проверку типов, подобно тому, как это делается в таких языках как Pascal. Если используются прототипы, то компилятор может обнаружить любые сомнительные преобразования типов аргументов, необходимые при вызове функции, если тип ее параметров отличается от типов аргументов. При этом будут выданы предупреждения обо всех таких сомнительных преобразованиях. Компилятор также обнаружит различия в количестве аргументов, использованных при вызове функции, и в количестве параметров функции.
- В общем виде прототип функции должен выглядеть таким образом:
  тип имя_функции(тип имя_парам1, тип имя_парам2, ..., имя_парамN);

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

Область видимости функции

- Правила, определяющие область видимости, устанавливают, видит ли одна часть программы другую часть кода или данных и может ли к ним обращаться
- Каждая функция представляет собой отдельный блок операторов ⇒ код функции является закрытым и недоступным для любых операторов, расположенных в других функциях, кроме операторов вызова
- Код, который составляет тело функции, скрыт от остальной части программы, если он не использует глобальных переменных: не может влиять на другие части программы и, наоборот
- Переменные, определенные внутри функции, являются локальными – они создаются в начале выполнения функции, а при выходе - уничтожаются. Локальная переменная не может сохранять свое значение в промежутках между вызовами функции (исключение - переменные, объявленные со спецификатором класса памяти static)
- Все функции имеют файл в качестве области действия (file scope) - функцию нельзя определять внутри другой функции

Аргументы функции

- Формальный параметр (аргумент) – переменная, объявленная в заголовке функции
- Формальные параметры функции доступны внутри всей функции. Параметр создается в начале выполнения функции, и уничтожается при выходе из нее
- Фактический параметр (аргумент) - это конкретное значение, которое присваивается переменной, называемой формальным аргументом. Фактический аргумент может быть константой, переменной или более сложным выражением. Независимо от типа фактического аргумента он вначале вычисляется, а затем его величина передается функции.

```
/* Возвращает 1, если символ c входит в строку s и 0 в противном случае. */
int is_in(char *s,  char c) // определение функции
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
…
char str[]="function", ch='u';
int a;
a=is_in(str, ch); // вызов функции
```

Передача параметров по значению и по ссылке

В языках программирования существует два способа передачи аргументов в подпрограммы:

- *по значению* – формальным параметрам подпрограммы присваиваются копии значений фактических аргументов, и все изменения формальных параметров не отражаются на фактических аргументах
- *по ссылке* – формальным параметрам подпрограммы присваиваются адреса фактических аргументов, и внутри подпрограммы открывается доступ к фактическому аргументу. Это значит все изменения формальных параметров отражаются на аргументах

Передача параметров по значению

В языке C/C++ по умолчанию применяется передача параметров *по значению.*

```
#include <stdio.h>
int sqr(int x); // прототип функции
int main(void)
{
  int t=10;
  printf("%d %d", sqr(t), t); // вызов функции
  return 0;
}
int sqr(int x) // определение функции
{
  x = x*x;
  return (x);
}
```

Передача по ссылке

Если нужно передать параметры по ссылке, то формальные параметры объявляются как указатели

```
void swap(int*x, int *y);
int main(void)
{
    int i, j;
    i=10;
    j=20;
    swap(&i, &j); // передаются адреса переменных i, j
    return 0;
}
void swap(int *x, int *y)
{
    int temp;
    temp = *x; // сохраняем значение, записанное по адресу x
    *x = *y; // записываем значение, записанное по адресу y, в ячейку в адресом x
    *y =  temp; // записываем сохраненное значение
}
```

Результат работы программы:

i и j перед обменом значениями: 10 20

i и j после обмена значениями: 20 10

Передача массива в функции

В языке C/C++ весь массив нельзя передать в качестве аргумента функции. Можно передать указатель на массив, то есть имя массива без индексов.

Например,

```
int main(void)
{
  int mass[12];
  read_mass(mass);
}
```

Передача одномерного массива

Если аргументом функции является одномерный массив, то ее формальный параметр можно объявить тремя способами:

- как указатель
- как массив фиксированного размера
- как массив неопределенного размера

Пример 1:

```
void read_mass(int *p) // указатель
{
   …
}
```

Пример 2:

```
void read_mass(int p[13]) //  массив фиксированного размера
{
   …
}
```

Пример 3:

```
void read_mass(int p[]) // массив неопределенного размера
{
   …
}
```

Эти три объявления эквивалентны.

Передача строк

Строки, объявленные как массив символов, передаются в функции аналогично одномерным массивам. Формальный параметр объявлять нужно как указатель. Конец строки внутри функции определяется по нулевому байту

```
#include <stdio.h>
#include <ctype.h>
void print_upper(char *string);
int main(void)
{
  char s[80];
  printf("Введите строку символов: ");
  gets(s);
  print_upper(s);
  printf("\ns теперь на верхнем регистре: %s", s);
  return 0;
}
/* Печатать строку на верхнем регистре. */
void print_upper(char *string)
{
  register int t;
  for(t=0; string[t]; ++t)  {
    string[t] = toupper(string[t]);
    putchar(string[t]);
  }
}
```

Вот что будет выведено в случае фразы "This is a test.":

```
Введите строку символов: This is a test.
```

Вторая версия предыдущей программы: содержимое массива s остается постоянным

```
#include <stdio.h>
#include <ctype.h>
void print_upper(char *string);
int main(void)
{
  char s[80];
  printf("Введите строку символов: ");
  gets(s);
  print_upper(s);
  printf("\ns не изменялась: %s", s);
  return 0;
}
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)
    putchar(toupper(string[t]));
}
```

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

```
Введите строку символов: This is a test.
THIS IS A TEST.
s не изменилась: This is a test.
```

Оператор return

Оператор имеет два важных применения:

- он обеспечивает немедленный выход из функции, т.е. заставляет выполняющуюся программу передать управление коду, вызвавшему функцию
- этот оператор можно использовать для того, чтобы возвратить значение

Возврат из функции

Функция может завершать выполнение и осуществлять возврат в вызывающую программу двумя способами

- первый способ используется тогда, когда после выполнения последнего оператора в функции встречается закрывающая фигурная скобка (})
- для завершения выполнения используется оператор return — или потому, что необходимо вернуть значение, или чтобы сделать код функции проще и эффективнее.

```
#include <string.h>
#include <stdio.h>

void pr_reverse(char *s);

int main(void)
{
  pr_reverse("Мне нравится C");
  return 0;
}
void pr_reverse(char *s)
{
  register int t;
  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Количество операторов return

В функции может быть несколько операторов return

```
#include <stdio.h>
int find_substr(char *s1, char *s2);

int main(void)
{
  if(find_substr("C - это забавно", "is") != -1)
    printf("Подстрока найдена.");

  return 0;
}

/* Вернуть позицию первого, вхождения s2 в s1. */
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

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

Категории пользовательских функций

- Вычислительные функции – выполняют операции над своими аргументами и возвращают полученное в результате этих операций значение
  - Например, стандартные библиотечные функции sqrt() и sin(), которые вычисляют квадратный корень и синус своего аргумента соответственно.
- Функции, обрабатывающие информацию и возвращающие значение, которое показывает, успешно ли была выполнена эта обработка.
  - Например, библиотечная функция fclose(), которая закрывает файл. Если операция закрытия была завершена успешно, функция возвращает 0, а в случае ошибки она возвращает EOF.
- Функции, не имеющие явно возвращаемых значений. В сущности, такие функции являются чисто процедурными и никаких значений выдавать не должны. Все функции, которые не возвращают значение, должны объявляться как возвращающие значение типа void. Такие функции нельзя применять в выражениях
  - Например, функция exit(), которая прекращает выполнение программы.

Возвращаемые значения

- Все функции, кроме тех, которые относятся к типу void, с помощью оператора return возвращают значение
- В соответствии со стандартом C89, если функция, возвращающая значение, не использует для этого оператор return, то в вызывающий модуль возвращается «мусор»
- В языке C++ (в стандарте C99) все функции, тип которой отличен от void, должны возвращать значение с помощью оператора return

Если функция не объявлена как имеющая тип void, она может использоваться как операнд в выражении

Поэтому каждое из следующих выражений является правильным:

```
x = power(y);
if(max(x,y) > 100) printf("больше");
for(ch=getchar(); isdigit(ch); ) ... ;
```

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

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

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

Обязательно присваивать возвращенное значение какой-либо переменной?

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
  int x, y, z;

  x = 10;   y = 20;
  z = mul(x, y);          /* значение, возвращаемое функцией mul(), присваивается переменной
     z */
  printf("%d", mul(x,y));  /* возвращаемое значение не присваивается, но используется
     функцией printf() */
  mul(x, y);              /* возвращаемое значение теряется, потому что не присваивается
     никакой из переменных и не используется как часть какого-либо выражения */

  return 0;
}

int mul(int a, int b)
{
  return a*b;
}
```

Возврат указателей

В объявлении функции, которая возвращает указатель, тип возвращаемого указателя должен декларироваться явно. Например, нельзя объявлять возвращаемый тип как int *, если возвращается указатель типа char *! Если требуется, чтобы функция возвращала "универсальный" указатель, то тип результата функции следует определить как void *.

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

```
#include <stdio.h>
char *match(char c, char *s);  /* прототип */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p)  /* символ найден */
    printf("%s ", p);
  else
    printf("Символа нет.");
```

Функция типа void

Одним из применений ключевого слова void является явное объявление функций, которые не возвращают значений

```
#include <stdio.h>
void print_vertical(char *str);  /* прототип */
int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

И еще одно замечание: в ранних версиях C ключевое слово void не определялось. Таким образом, в программах, написанных на этих версиях C, функции, которые не возвращали значений, просто имели по умолчанию тип int — и это несмотря на то, что они не возвращали никаких значений!

Что возвращает функция main()?

- Функция main() возвращает целое число, которое принимает вызывающий процесс — обычно этим процессом является операционная система. Возврат значения из main() эквивалентен вызову функции exit() с тем же самым значением. Если main() не возвращает значение явно, то вызывающий процесс получает формально неопределенное значение. На практике же большинство компиляторов C автоматически возвращают 0, но если встает вопрос переносимости, то на такой результат полагаться с уверенностью нельзя.

Рекурсия

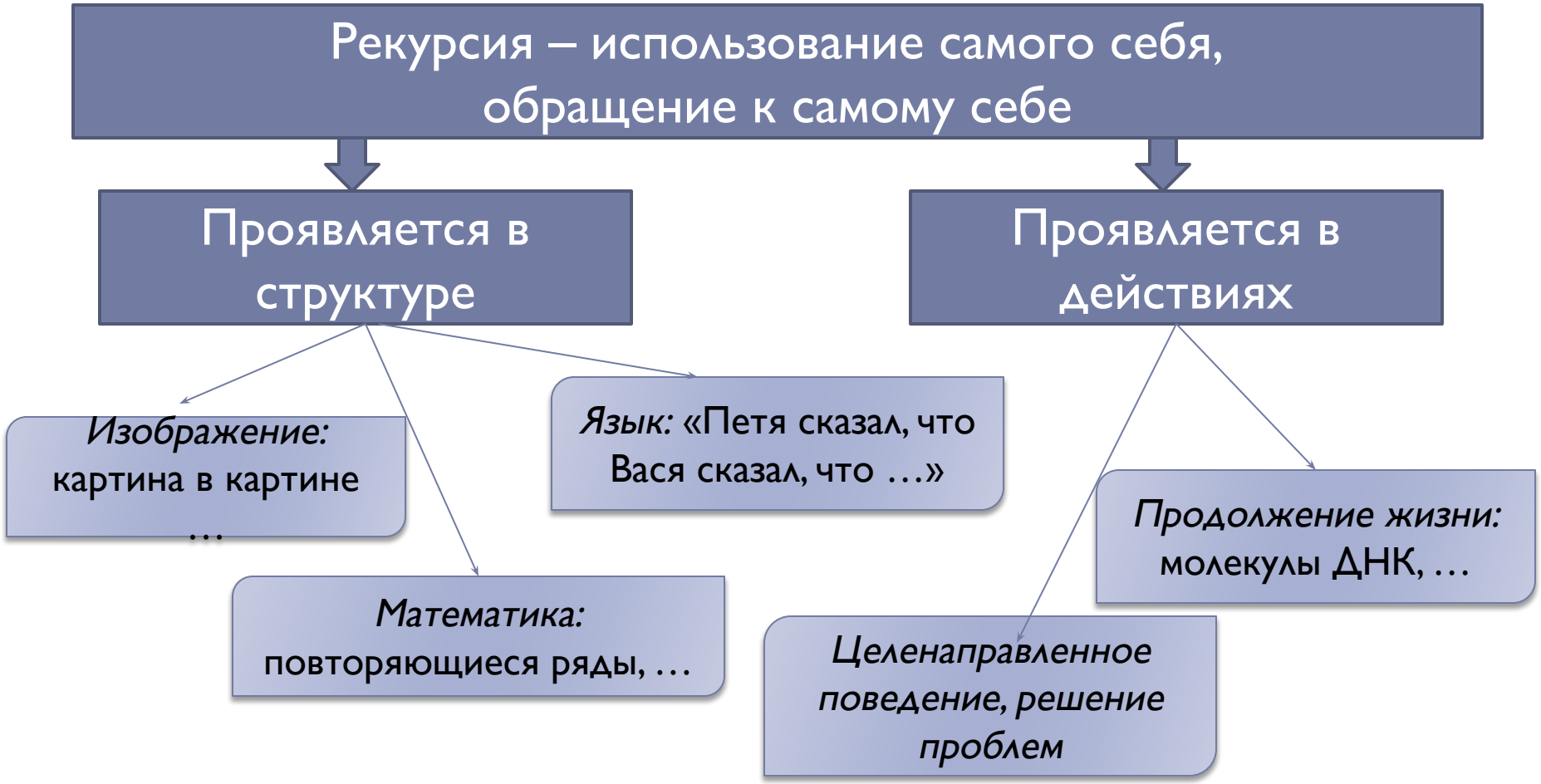
В языке С функция может вызывать сама себя. В этом случае такая функция называется рекурсивной.
Рекурсия — это процесс определения чего-либо на основе самого себя, из-за чего рекурсию еще называют рекурсивным определением.
Простым примером рекурсивной функции является factr(), которая вычисляет факториал целого неотрицательного числа. Факториалом числа n (обозначается n!) называется произведение всех целых чисел, от 1 до n включительно (для 0, по определению, факториал равен 1.). Например, 3! — это 1×2×3, или 6.
Здесь показаны factr() и эквивалентная ей функция, в которой используется итерация:

```
/* рекурсивная функция */
int factr(int n) {
  int answer;

  if (n==1) || (n == 0) return (1);
  answer = factr(n-1)*n; /* рекурсивный вызов */
  return(answer);
}


/* нерекурсивная функция */
int fact(int n) {
  int t, answer;
```

Что такое рекурсия



Что такое рекурсия

Рекурсия – организация вычислений,
это образ мыслей и методология решения задач

Повторяющиеся вычисления

Разбиение проблемы и разделение ее на подзадачи

Механизм возврата

Определения функции через её предыдущие и ранее определенные значения

Способ организации вычислений, при котором функция вызывает сама себя с другим аргументом

Примеры рекурсивного задания функций

- f(0)=0
  f(n)= f(n-1)+1
- f(0)=1
  f(n)= n*f(n-1)
- f(0)=1
  f(1)=1
  f(n)= f(n-1)+ f(n-2), n>=2
- f(0)=1
  f(n)= f(n-1)+ f(n-2)+…+1 = f(i)+1
- f(0)=1
  f(1)=2
  f(n)= f(n-1)*f(n-2)

Механизм реализации рекурсии

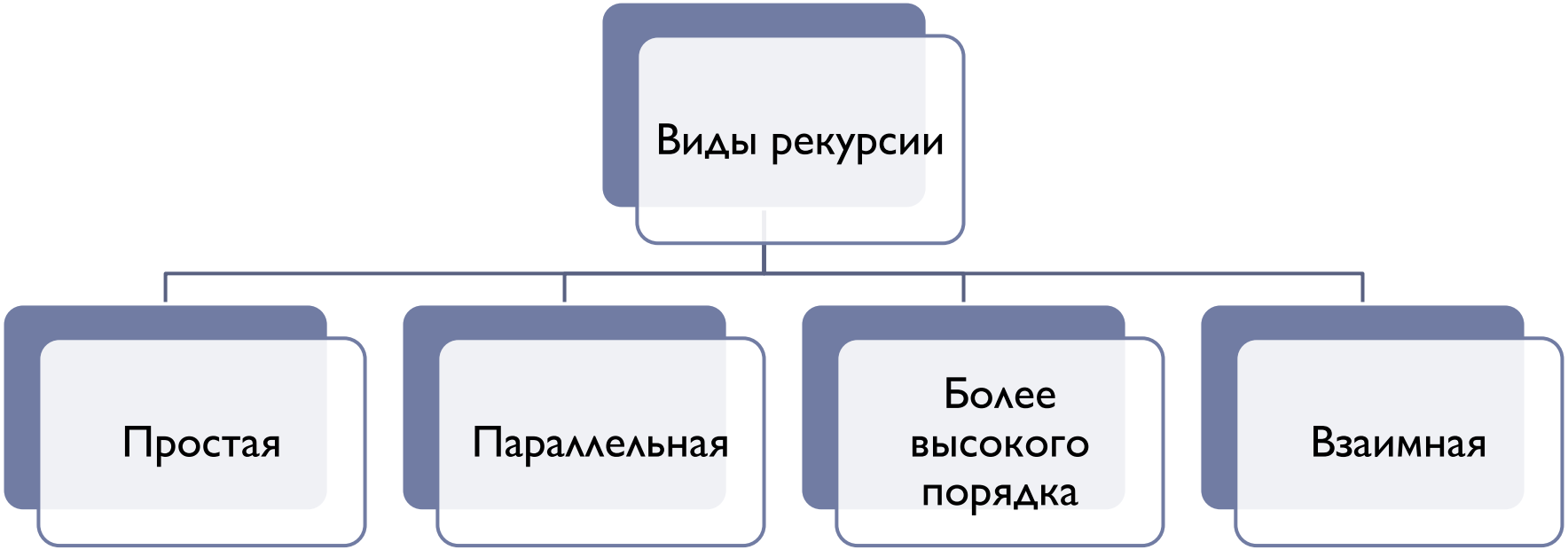
1. Рекурсивная функция всегда содержит хотя бы одну терминальную ветвь и условие окончания

2. Когда процедура доходит до рекурсивной ветви, то функционирующий процесс приостанавливается

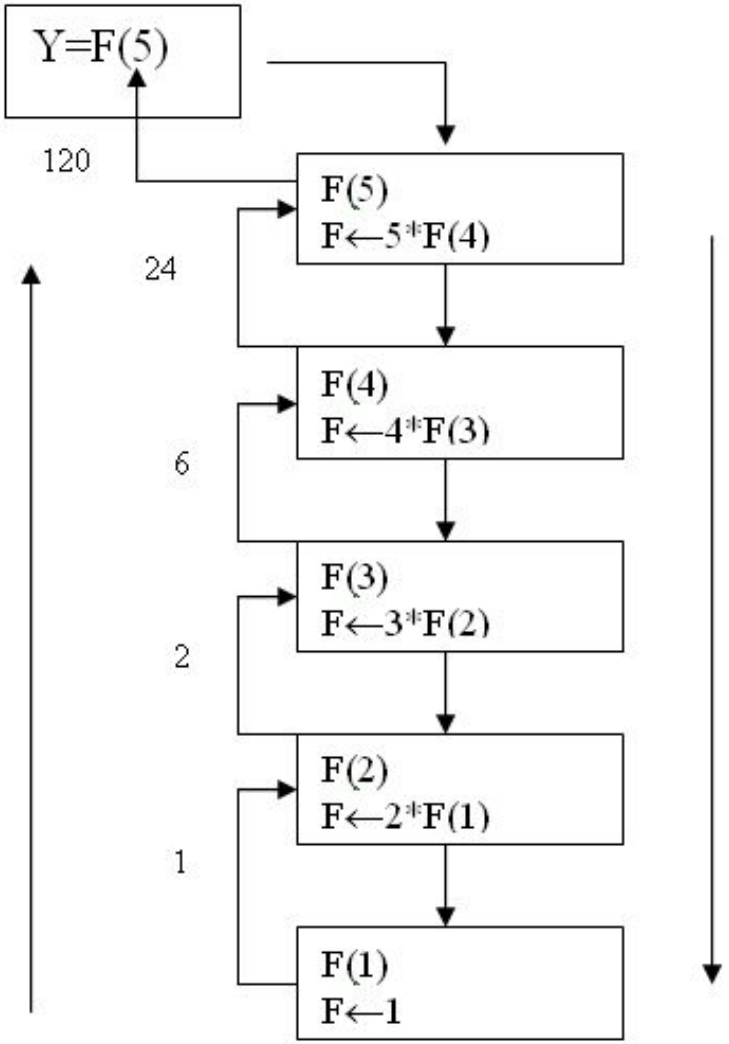
3. Запускается новый такой же процесс, но уже на новом уровне

4. Прерванный процесс начнет исполняться лишь после окончания нового процесса

5. В свою очередь новый процесс может приостановиться и ожидать …


Виды рекурсии
Простая
Параллельная
Более высокого порядка
Взаимная

Дерево рекурсии при вычислении факториала – F(5)



Работа рекурсивной функции

Пусть n=4

answer = factr(3)*4; return (answer)

вызов функции — возврат значения функции

answer = factr(2)*3; return (answer)

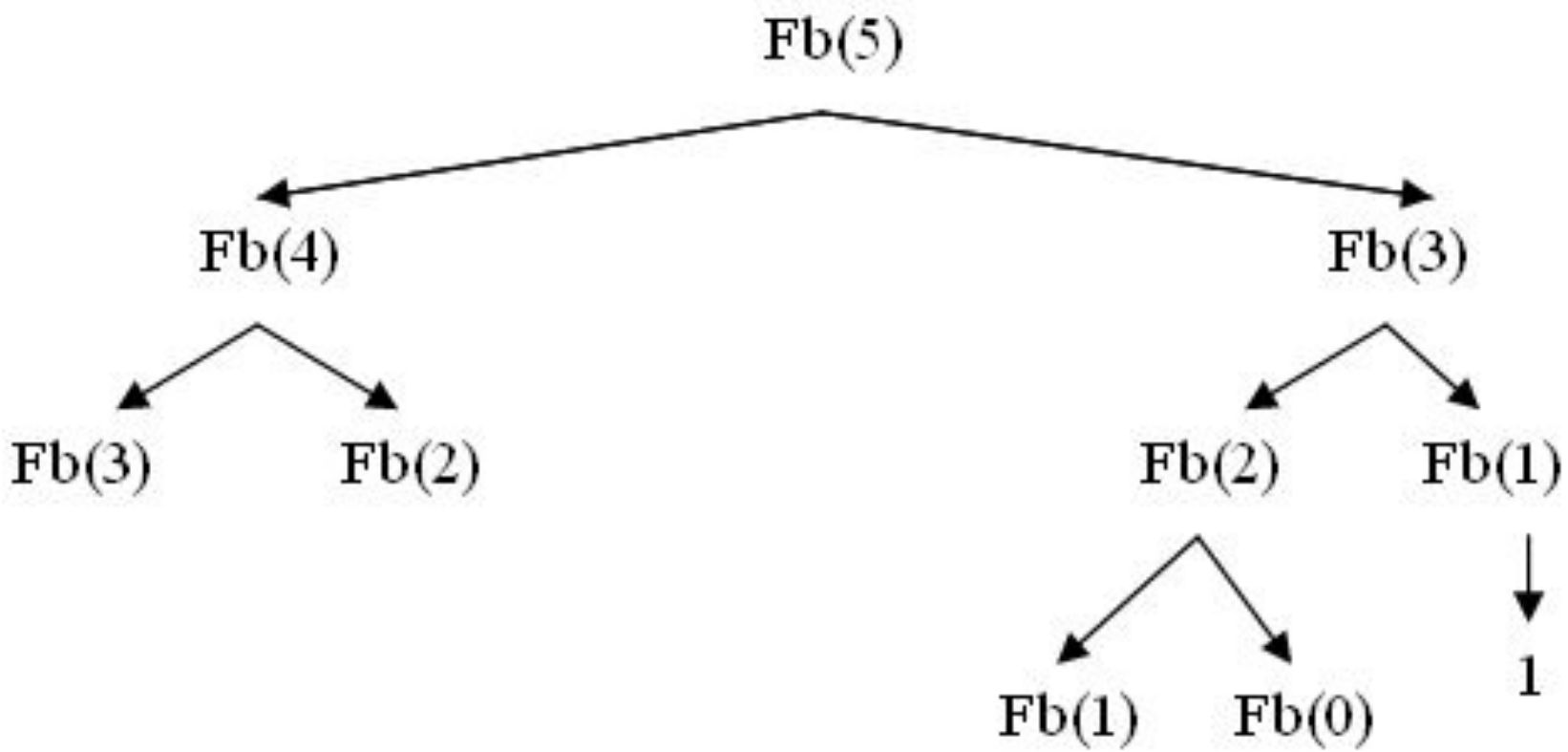
вызов функции — возврат значения функции

answer = factr(1)*2; return (answer)

вызов функции — возврат значения функции

return (1)

Фрагмент дерева рекурсии при вычислении чисел Фибоначчи – F(5)



Анализ трудоемкости рекурсивных реализаций алгоритмов

- Связан как с количеством операций, выполняемых при одном вызове функции, так и с количеством таких вызовов
- Более детальное рассмотрение приводит к необходимости учета затрат как на организацию вызова функции и передачи параметров, так и на возврат вычисленных значений и передачу управления в точку вызова
- Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках IBM PC совместимых компьютеров этот механизм реализован через программный стек. Как передаваемые в процедуру или функцию фактические параметры, так и возвращаемые из них значения помещаются в программный стек специальными командами процессора. Дополнительно сохраняются значения необходимых регистров и адрес возврата в вызывающую процедуру.

Механизм вызова процедуры с использованием программного стека