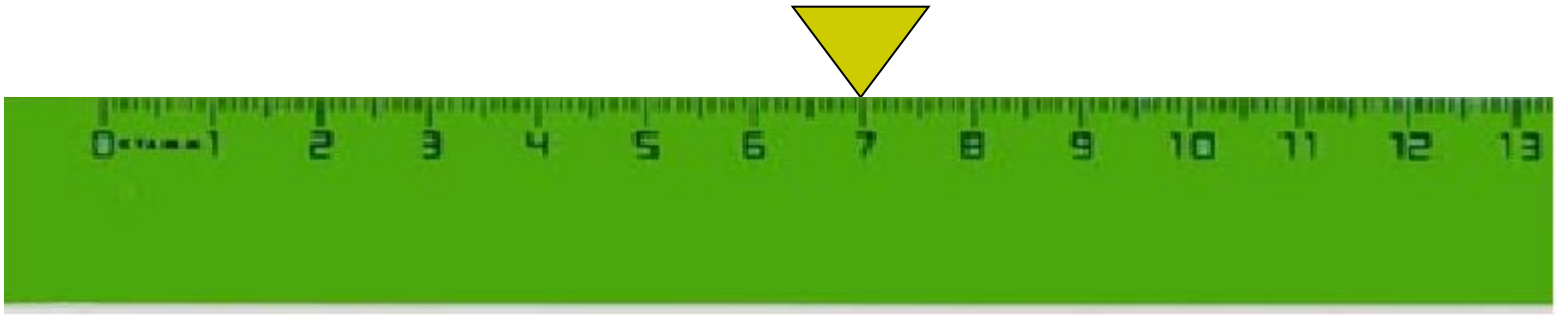
Модуль 7

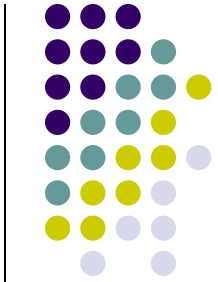
Функции

1. Описание функции
2. Вызов функции, передача параметров, возврат значения
3. Классы памяти
4. Ссылки
5. Рекурсия
6. Перегрузка функций
7. Указатель на функцию

(14 пар)

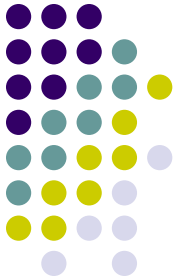


Модульное программирование



- Разделение программы на небольшие части, называемые модулями
  - Небольшой модуль легче написать и отладить
  - Модули можно использовать повторно
  - Модули можно объединить в библиотеки и предоставить возможность их использования другим программистам
  - Разбиение программы на модули повышает качество кода
- В С модули реализуются с помощью функций
- Функция – именованный фрагмент кода, который может быть вызван многократно
- Функции в С
  - Стандартные (в составе библиотек)
  - Определенные программистом

Стандартные функции

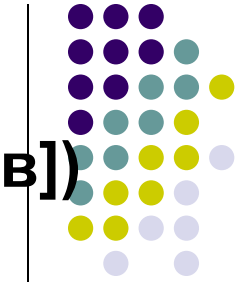


- Описаны в файле <stdio.h>
  - printf(“x= %d”,x);
  - scanf(“%d”,&x);
- Описаны в файле <math.h>
  - y=sqrt(900);
  - z=sin(y);
- Описаны в файле <time.h>
  - time(0);

Описание функции программистом

тип_результата имя_функции([список_параметров])

```
{        … //тело функции
        return выражение;  //возврат результата
}
double sqr(double x)
{    return x*x;
}
```

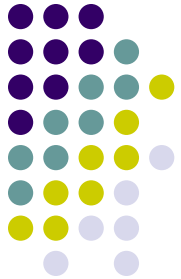


Вызов функции

имя_функции([список_аргументов]);

```
void main()
{    double z;
     z=sqr(3.5);
     cout<<z;
     cout<<sqr(48);
}
```

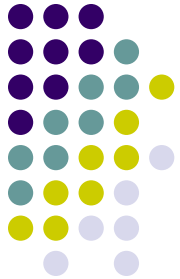
Пример. Функция возвращает 1, если аргумент положителен, -1, если отрицателен, и 0 – если равен 0



```
int signum (int x)
{
    if (x>0) return 1;
    else if (x<0) return -1;
       else return 0;
}
```

```
int signum (int x)
{
    if (x>0) return 1;
    if (x<0) return -1;
    return 0;
}
```

Тип функции void

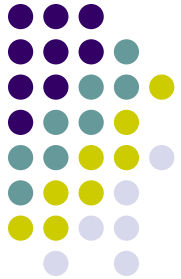


- означает, что функция не возвращает значение
- В теле такой функции оператор return используется без указания выражения:

  return;
- Оператор return может также отсутствовать, тогда выход из функции происходит по достижению последней закрывающей скобки

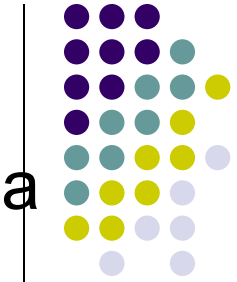
```
void print (int a)
{
   cout<<“Значение= “<<a<<“\n”;
}
```

Функция должна быть описана до первого вызова



```
#include <iostream>
using namespace std;
//описание функции
double sqr(double x)
{
    return x*x;
}
void main()
{
    setlocale(LC_ALL,"rus");
    double z;
    z=sqr(3.2); //вызов функции
    cout<<"Результат= "<<z<<"\n";
    system("pause");
}
```

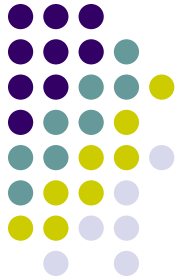
Прототип функции



- Используется в случае, когда функция должна быть вызвана до ее описания.
- Сообщает компилятору интерфейс функции и дает возможность проверить правильность вызова.

```
double sqr(double); //прототип
void main()
{ …
   z=sqr(3.5);  //вызов
}
double sqr(double x)    //описание
{    return x*x;
}
```

Правила описания функции



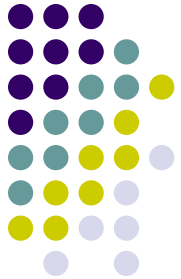
- Функция должна выполнять только ***одну***, небольшую задачу. Хорошим стилем программирования считается, если объем функции не превышает половины страницы кода.
- Имя функции должно иметь ясный смысл, отражать назначение функции.

  sumOfArray() – функция для определения суммы всех элементов массива.
- Если тип результата или параметров не указан, компилятор предполагает int.

  double x, y

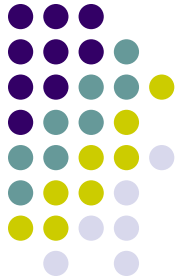
  интерпретируется как double x, int y.
- Нельзя описать одну функцию внутри другой.

Вызов функции и передача параметров



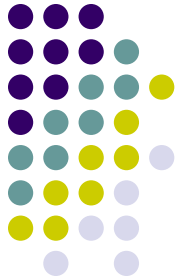
- Аргументы ставятся в соответствие параметрам по порядку в списке. Типы соответствующих аргументов и параметров должны совпадать.
- В языке С аргументы в функцию передаются **по значению,** т.е. вызванная функция получает временную копию каждого аргумента.
- Функция не может изменять оригинальный аргумент в вызвавшей программе.

Последовательность действий при вызове функции



1. Создаются временные переменные для каждого формального параметра, который приведен в заголовке функции (под них выделяется место в памяти).
2. Устанавливается соответствие между аргументами в вызове функции (фактическими параметрами) и формальными параметрами в ее заголовке. Аргументы ставятся в соответствие параметрам по порядку в списке. Типы соответствующих аргументов и параметров должны совпадать.
3. Каждому параметру присваивается копия соответствующего аргумента.
4. Управление передается в функцию. Код функции выполняется до тех пор, пока не встретится оператор return. Записанное в нем значение передается в место вызова.
5. При выходе из функции временные копии формальных параметров уничтожаются (память освобождается, они становятся недоступны)
6. Далее выполняется код, который находится после вызова функции.

```
#include <iostream>
using namespace std;
//описание функции
double fun (int a, double b)
{
    return a+b;
}
void main()
{
    setlocale(LC_ALL,"rus");
    int k=5;
    double d,l;
    //вызов 1
    d=fun(3,7.6);
    cout<<"d= "<<d<<"\n";
    //вызов 2
    l=fun(k,d);
    cout<<"l= "<<l<<"\n";
    system("pause");
}
```



a	5
b	10.6

Пример одинаковых имен фактических и формальных параметров

```
int sum (int a, int b)
{  return a+b;  }
void main()
{  int a=2, b=3, c, d;
   c=sum(a,b);
   d=1;
   cout<<sum(c,d);
}
```

На консоль выводится 6

Переменные main()

a	2
b	3
c	5
d	1

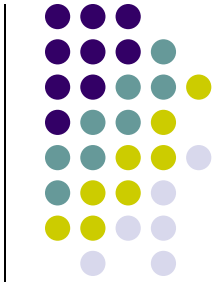
Параметры sum() 1 вызов

a	2
b	3

Параметры sum() 2 вызов

a	5
b	1

Преобразование типов при вызове функции

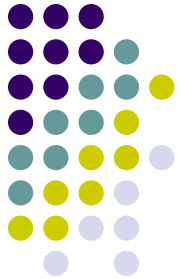


- При вызове функции происходит автоматическое приведение аргументов к соответствующему типу по общим правилам преобразования типов в С

```
double sqr( double x);
void main()
{  cout<<sqr(4);  }
```

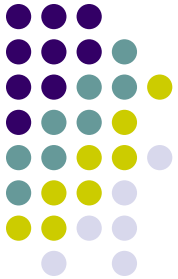
Целый аргумент 4 автоматически преобразуется к типу double

Будет выведено 16.000



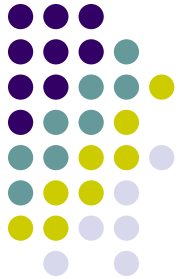
Описать функцию расчета длины гипотенузы треугольника, в которую передаются длины двух катетов. С помощью этой функции вычислить гипотенузы треугольников с катетами 2 и 3, 10 и 12.

Функция перестановки значений переменных



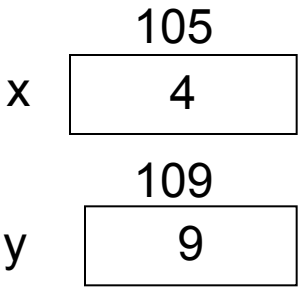
```
void swap(int a, int b)
{
    int temp=a;
    a=b;
    b=temp;
}
void main()
{  int x=4,y=5;
   swap(x,y);
    …
}
```

Передача в функцию указателей

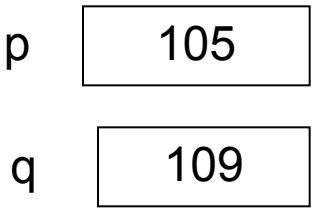


```
void swap(int *p,int *q)
{   int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}
void main()
{   int x=4,y=9;
    swap(&x,&y);
}
```

Переменные функции main()



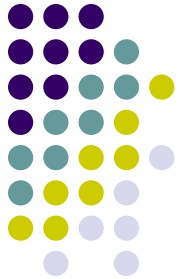
Переменные функции swap()



&x – адрес в памяти переменной x

*p – содержимое памяти по адресу p

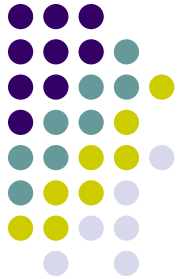
Передача в функцию имени массива



- Имя массива – это адрес его нулевого элемента
- Поэтому функция, получив имя массива, знает, где в памяти находится массив, и может изменять его значения.
- Кроме имени массива, в функцию следует передать его размерность.
- Массив как параметр функции можно описать двумя способами:

1. указать имя и две пустые скобки : a[];
2. объявить как указатель: *a.

Пример: функция, которая заменяет в массиве все 0 на 1.



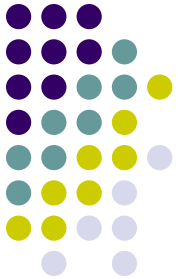
1 способ:

```
void convertArray(int mas[], int n)
{   for(int i=0;i<n;i++)
        if (mas[i]==0) mas[i]=1;
}
```

2 способ:

```
void convertArray(int *mas, int n)
{   for(int i=0;i<n;i++,mas++)
        if (*mas==0) *mas=1;
}
```

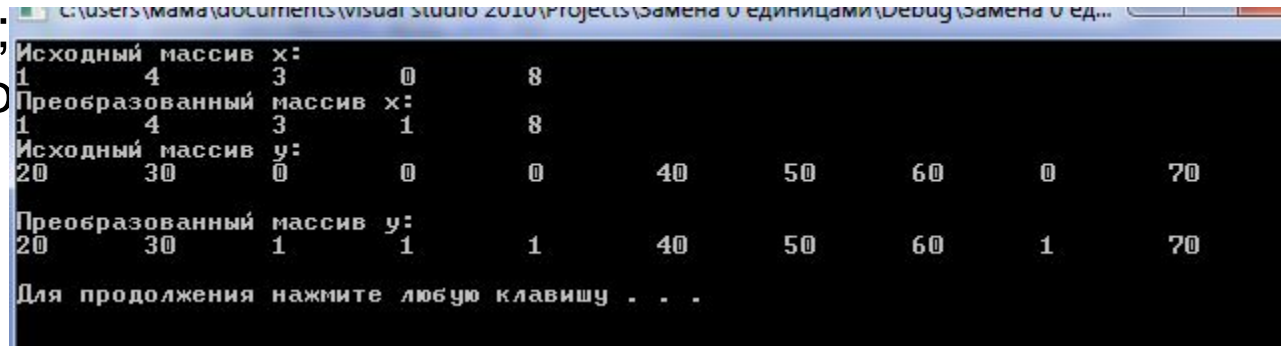
Пример: функция, которая выводит массив на печать



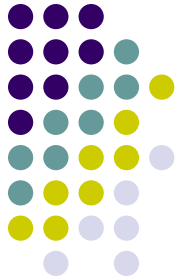
```
void printArray(int *mas,int n)
{
   for(int i=0;i<n;i++,mas++)
    cout<<*mas<<"\t";
   cout<<"\n";
}
```

Программа, которая использует функции работы с массивами

```
void main()
{   setlocale(LC_ALL,"rus");
    int x[5]={1,4,3,0,8};
    int y[10]={20,30,0,0,0,40,50,60,0,70};
    cout<<"Исходный массив x:\n";
    printArray(x,5);
    convertArray(x,5);
    cout<<"Преобразованный массив x:\n";
    printArray(x,5);
    cout<<"Исходный массив y:\n";
    printArray(y,10);
    convertArray(y,10);
    cout<<"Преобразо
    printArray(y,10);
    system("pause");
}
```



Параметр-константа

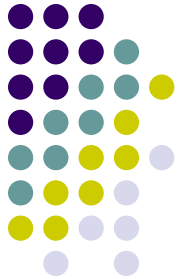


- Работа с массивом внутри функции позволяет изменять сам объект, т.е. элементы массива. Если такое поведение нежелательно, то нужно объявить параметр-массив как константу:

```
void printArray(const int a[], int n);
```

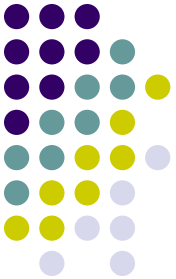
Или

```
void printArray(const int *a, int n);
```



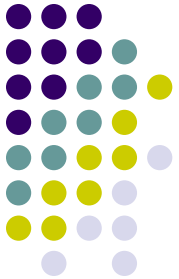
Описать функцию определения среднего арифметического в массиве и с помощью этой функции рассчитать среднее значение в двух массивах различной длины (3 или 7 элементов). Массивы инициализировать случайными числами от 0 до 20.

Функции и двумерные массивы



- Двумерный статический массив компилятор интерпретирует как массив из массивов-строк.
- Имя двумерного массива из N строк и M столбцов – это указатель на указатель на нулевую строку (массив из M элементов).
- Поэтому при передаче в функцию имени двумерного массива число столбцов будет фиксировано.
- Количество же строк можно передавать в качестве параметра.

1 способ передачи двумерного массива в функцию – как двумерный массив с фиксированным числом столбцов.

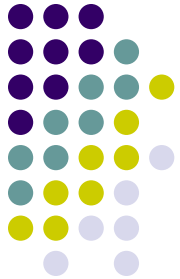


```
#define N 4
#define M 5
//инициализация двумерного массива случайными числами
void initArray(int mas[][M],int n)
{
    for(int i=0;i<n;i++)
     for(int j=0;j<M;j++)
          mas[i][j]=rand()%11-5;
}
```

- Вызов:

```
int a[N][M];
initArray(a,N);
```

2 способ передачи двумерного массива в функцию – как указатель на массив-строку.

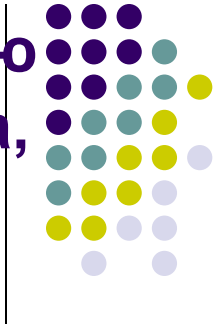


```
//печать элементов двумерного массива в виде
//таблицы
void printArray(int (*mas)[M],int n)
{
    for(int i=0;i<n;i++)
    {
      for(int j=0;j<M;j++)
            cout<<mas[i][j]<<"\t";
      cout<<"\n";
    }
}
```

- Вызов аналогичен:

```
printArray(a,N);
```

3 способ передачи двумерного массива в функцию – передать указатель на нулевой элемент массива, количество строк и количество столбцов.

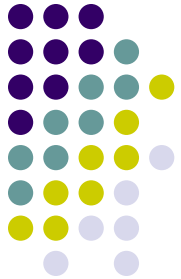


```
//сумма элементов двумерного массива
int sumOfArray(int *mas, int n, int m)
{
    int sum=0;
    for(int i=0;i<n;i++)
     for(int j=0;j<m;j++)
     {
          sum+=*mas;
          mas++;
     }
    return sum;
}
```

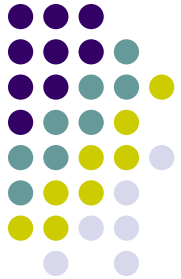
- Вызов:

```
sumOfArray(&a[0][0],N,M);
```

Напишите функцию, которая подсчитывает количество элементов двумерного массива, меньших заданного числа (это число также передается как параметр)



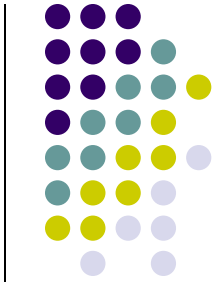
Параметры функции по умолчанию



- Если параметру функции присвоено значение в заголовке функции, то это значение используется по умолчанию (если параметр не задан).
- Значения по умолчанию могут быть только в конце списка параметров.

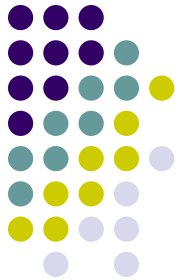
```
void print(int x, int y=0, char c='+');
print(6,2,'%');
print(7,8);
print(10);
```

Функция с неограниченным числом аргументов



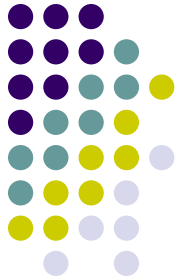
- Используется многоточие (…) в списке параметров.
- должен быть хотя бы один параметр, для которого известно имя и тип.
- Многоточие может быть только в конце списка параметров.
- В теле функции нужно каким-то образом, путем анализа известных параметров, получить информацию о количестве и типах аргументов, переданных в каждом конкретном случае.

Функция суммирует произвольное число целых чисел.



```
#include <iostream>
using namespace std;
int sum(int,...);
void main()
{
	setlocale(LC_ALL,"rus");
	//последний ноль в списке параметров - признак конца
	cout<<"Сумма трех чисел= "<<sum(1,2,3,0)<<"\n";
	cout<<"Сумма пяти чисел равна= "<<sum(1,2,3,4,5,0)<<"\n";
	system("pause");
}
int sum(int a,...)
{
	int sum=0;
	int *p=&a; //берем адрес первого аргумента
	while (*p) //пока значение по адресу p не равно 0
	{
	  sum+=*p;
	  p++; //переходим к следующему аргументу
	}
	return sum;
}
```

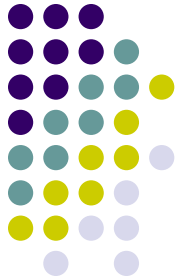
Локальные переменные



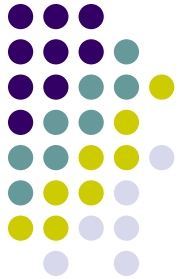
- Областью действия локальной переменной является блок, внутри которого эта переменная описана.
- Локальные переменные создаются при входе в блок и уничтожаются при выходе из него.

```
double sqr (double x)
 {  double z;
     z=x*x;
     return z;
  }
  void main()
  { double y=sqr(2.5);
     cout<<z;   //ошибка! z не существует!
  }
```

Одинаковые имена локальных переменных в разных функциях (блоках) не конфликтуют



```
#include <iostream>
using namespace std;
double sqr(double x)
{
     double y=x*x;
     return y;
}
void main()
{
    setlocale(LC_ALL,"rus");
    double x=2.5, y=4.5;
     cout<<"Результат 1= "<<sqr(x)<<"\n";
     cout<<"Результат 2= "<<sqr(y)<<"\n";
     system("pause");
}
```

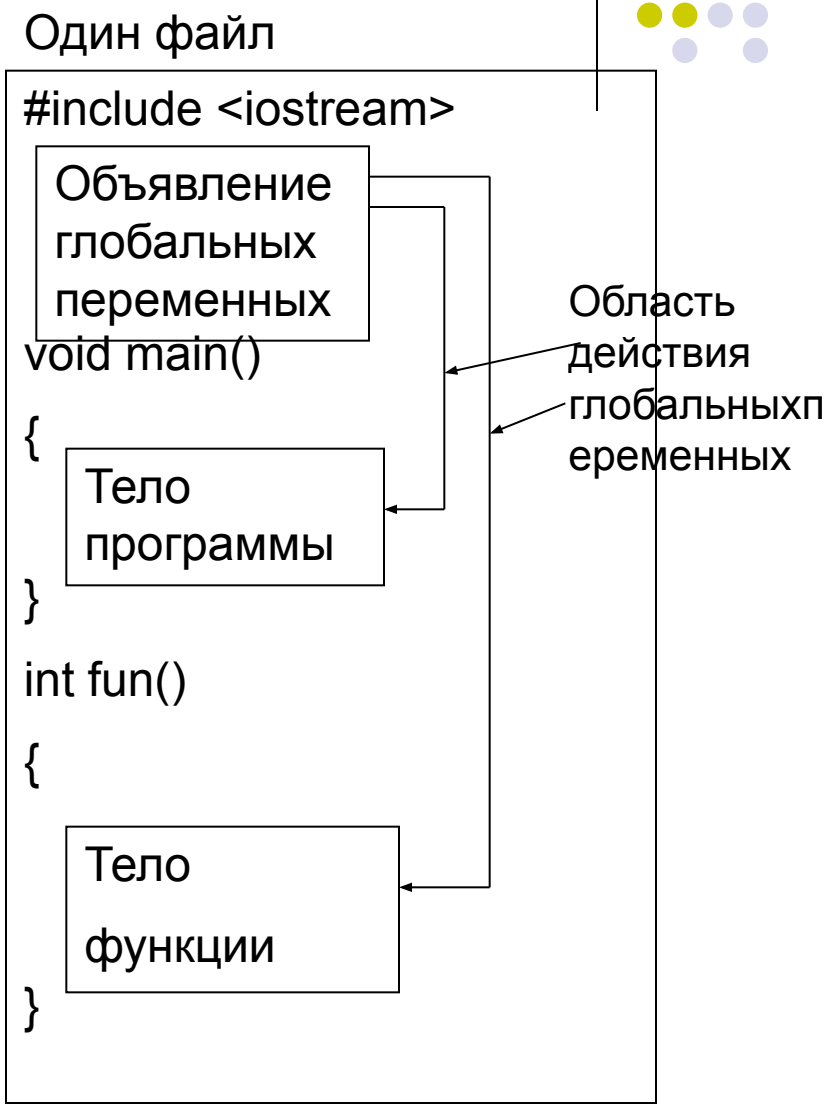


- Нельзя вернуть из функции указатель на локальную переменную (поскольку локальная переменная уничтожается при выходе из функции).

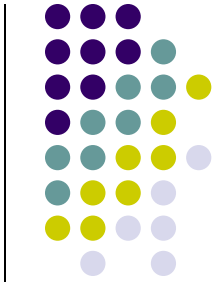
```
int* f()
{
   int a = 5;
   return &a;      // нельзя!
}
```

Глобальные переменные

- Объявляются вне любой функции
- Имеют область действия до конца файла
- Инициализируются 0 по умолчанию
- ***Следует избегать*** использования в программах глобальных переменных.

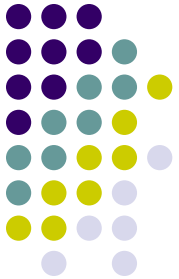


Пример глобальной переменной



```
#include <iostream>
using namespace std;
double counter; //глобальная переменная инициализируется 0
double sqr(double x)
{
	double y=x*x;
	counter++;
	return y;
}
void main()
{
	setlocale(LC_ALL,"rus");
	double x=2.5, y=4.5;
	cout<<"Результат 1= "<<sqr(x)<<"\n";
	cout<<"Счетчик вызовов= "<<counter<<"\n";
	cout<<"Результат 2= "<<sqr(y)<<"\n";
	cout<<"Счетчик вызовов= "<<counter<<"\n";
	system("pause");
}
```

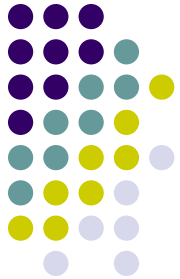
Разрешение конфликтов имен



- По умолчанию предполагается локальная переменная

```
#include <iostream>
int a=5;
int max (int x, int y)
{   int a;
    if (x>y) a=x;      //изменяется локальная, а не глобальная
     переменная
    else a=y;
    cout<<“a=“<<a<<“\n”; //вывод локальной переменной
    return a;
}
void main()
{   cout<<“a=“<<a<<“\n”; //вывод глобальной переменной
    cout<<max(3,8);
}
```

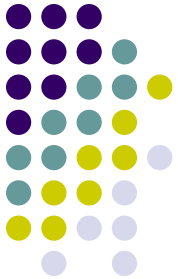
Оператор разрешения (::)



```
#include <iostream>
using namespace std;
int number=1001; //глобальная переменная
void shownumbers(int number) //параметр - локальная переменная
{
    cout<<"Локальная переменнная = "<<number<<"\n";
    cout<<"Глобальная переменная = "<<::number<<"\n";
}
void main()
{
    setlocale(LC_ALL,"rus");
    shownumbers(2002);
    system("pause");
}
```

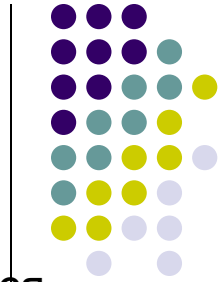
Локальная переменная = 2002
Глобальная переменная = 1001

Область действия и область видимости



- Область действия – это те фрагменты кода, где переменная существует (для локальных переменных: от точки объявления до конца блока, для глобальных – от точки объявления до конца файла).
- Область видимости – те фрагменты кода, где к переменной можно обратиться непосредственно, не используя оператор разрешения.

Статические переменные



- Локальная переменная с модификатором static не уничтожается при выходе из функции
- Пример: Создание серии чисел, причем следующее число вычисляется через предыдущее

```
#include <iostream>
using namespace std;
int series(void)
{
    static int series_num;
    series_num = series_num+23;
    return(series_num);
}
void main()
{
     setlocale(LC_ALL,"rus");
     cout<<"Серия чисел:\n";
     for(int i=0;i<10;i++)
       cout<<series()<<"\t";
     cout<<"\n";
     system("pause");
}
```

Ссылки

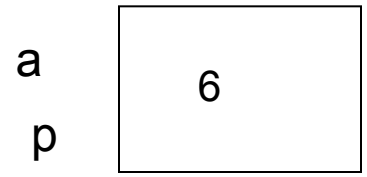
- Ссылка – это псевдоним переменной.
- Она инициализируется при объявлении и изменению не подлежит.
- Формат объявления:

тип &имя_ссылки = имя_переменной;

// тип ссылки и переменной должны быть одинаковыми.

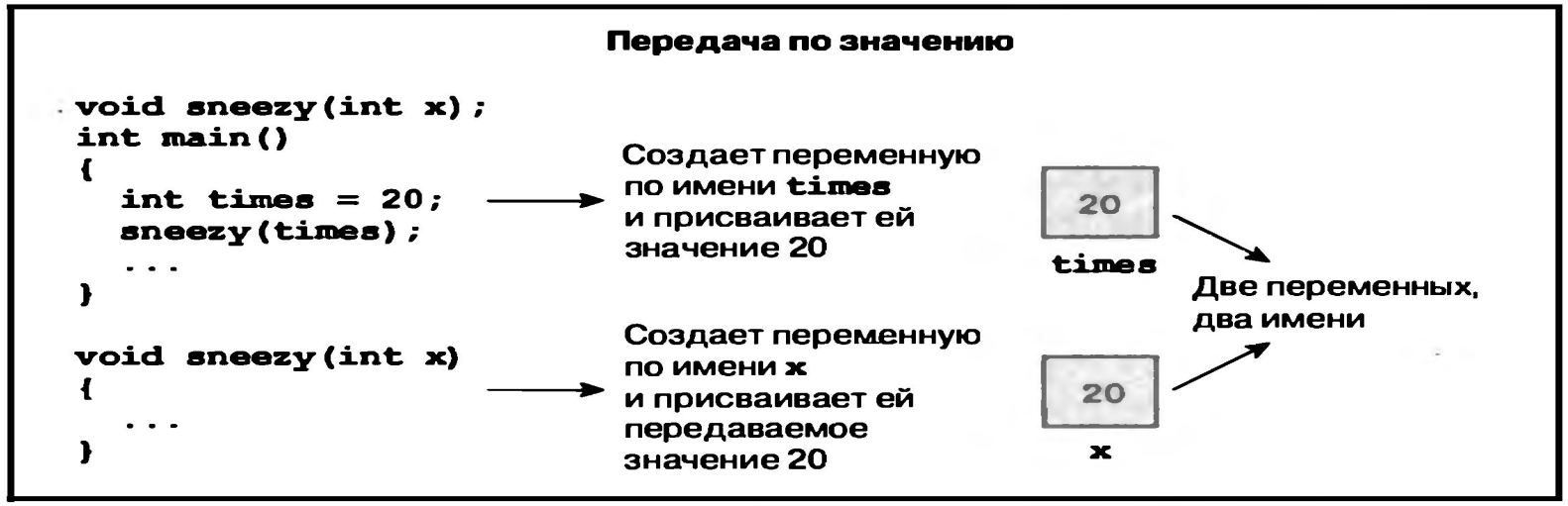
Пример.

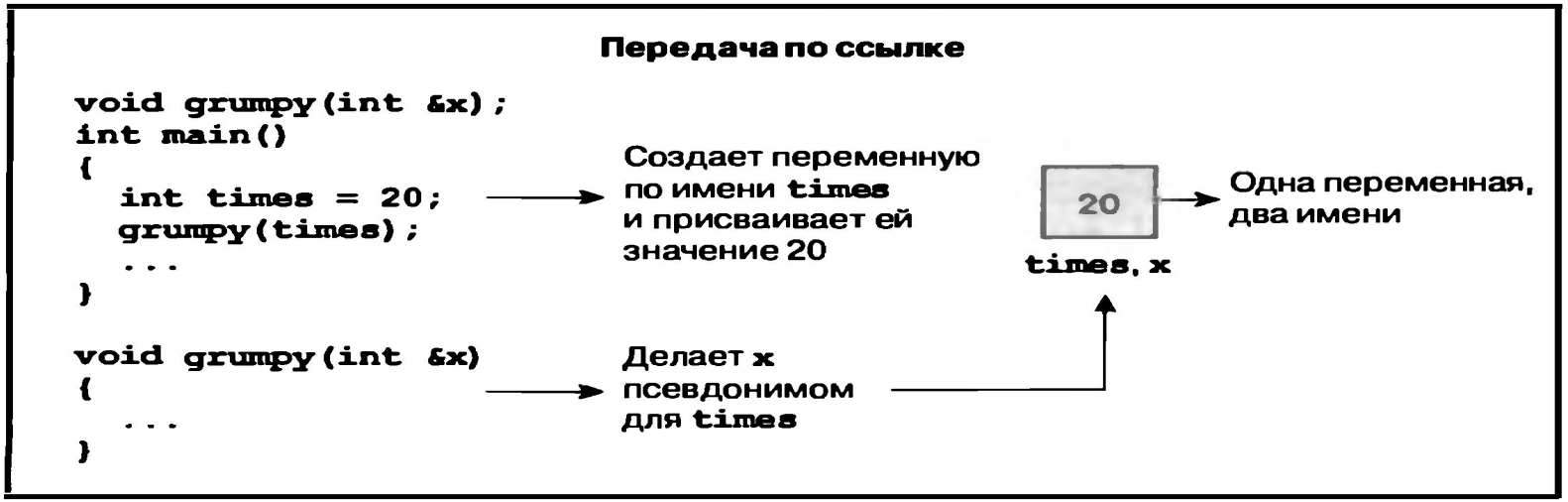
```
int a = 5; //объявляем переменную
int &p = a; //объявляем ссылку. теперь p это псевдоним a
cout << a << ' ' << p << endl; //выведет 5 5
a = 6;
cout << a << ' ' << p << endl; //выведет 6 6
```



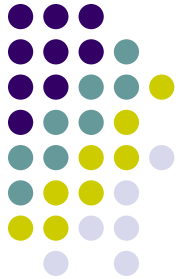
Ссылки как параметры функции

- Если ссылка-параметр функции, то из функции можно напрямую работать с самой переменной





Сравнение ссылок и указателей как параметров функции

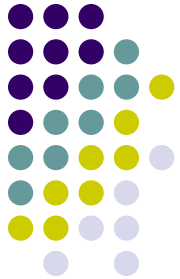


```
//ссылки
void myswap(int &a,int &b)
{
  int temp=a;
  a=b;
  b=temp;
}
void main()
{
  int x=5,y=6;
  myswap(x,y);
        cout<<x<<' '<<y<<endl;
}
```

```
//указатели
void myswap(int *a,int *b)
{
  int temp=*a;
  *a=*b;
  *b=temp;
}
void main()
{
  int x=5,y=6;
  myswap(&x,&y);
        cout<<x<<' '<<y<<endl;
}
```

1) Код с использованием ссылок выглядит проще и лаконичнее.
2) При передаче данных по ссылке не происходит копирование, что ускоряет вызов функции и экономит память.

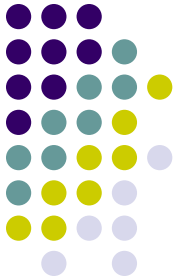
Защита информации при использовании ссылок



- При использовании ссылок любое изменение параметра внутри функции отражается на оригинале, это снижает безопасность кода, повышает вероятность ошибки.
- Если нужно защитить данные при передаче по ссылке (и при этом сохранить выигрыш в скорости), то можно использовать модификатор const:

```
int sum_by_reference(const int &reference)
// функция, принимающая аргумент по ссылке
// const не даёт изменить передаваемый аргумент внутри функции
```

Ссылки как результат функции



- Результат такой функции можно использовать в левой части оператора присваивания

Функция определяет максимум в массиве и выдает ссылку на него

1 вариант:

```
int &maxim(int a[], int n)
{
      int imax=0;
      for (int i=1; i<n; i++)
           if(a[i]>a[imax]) imax=i;
      return a[imax];
}
```

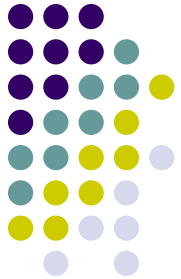
2 вариант:

```
int &maxim(int *a,int n)
{
     int *max=a;
     for(int i=0;i<n;i++,a++)
          if(*a>*max) max=a;
     return *max;
}
```

Вызов в main():

```
maxim(mas,n)=0;
```

Отличия ссылок от указателей

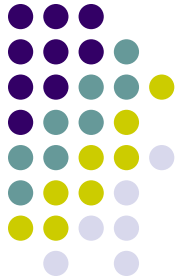


- Указатели ссылаются на участок в памяти, используя его адрес. А ссылки ссылаются на объект по его имени (тоже своего рода адрес).
- (Ссылка- особый вид указателя, который автоматически разыменовывается).
- Указатель – переменная, а ссылка – константа.
- При объявлении ссылка обязательно должна быть инициализирована, а указатель- не обязательно.
- Основное назначение указателей – это динамическое выделение памяти и эффективность при работе с массивами. Ссылки предназначены для организации прямого доступа к объекту (например, при передаче в функцию).

Стек

- **Стек** – это структура данных, которая реализует принцип «Последний пришел – первый вышел» (т.е. LIFO=Last In – First Out).

Стек вызовов

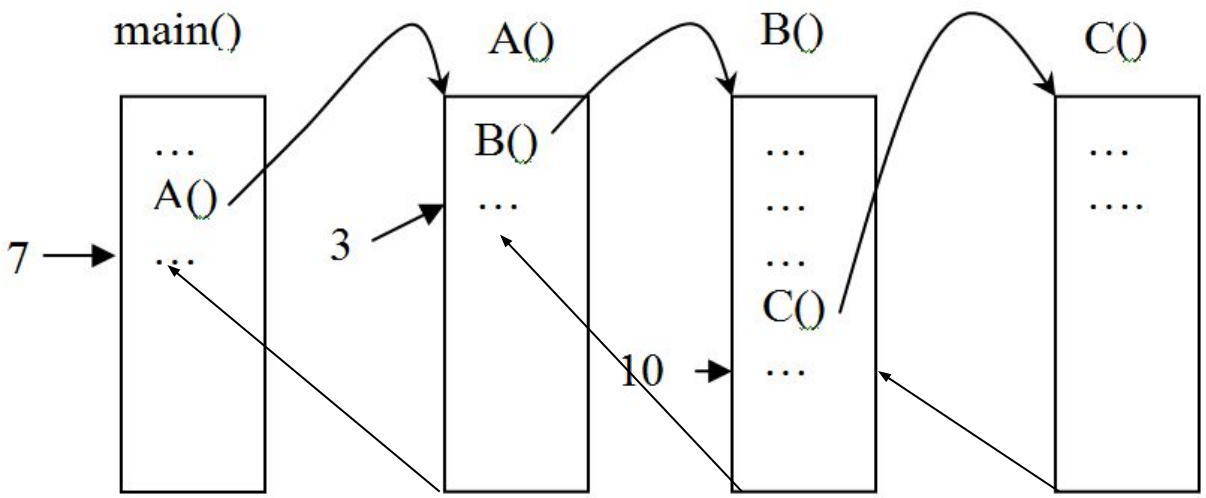


При каждом вызове функции в стеке помещается фрагмент данных, который содержит:

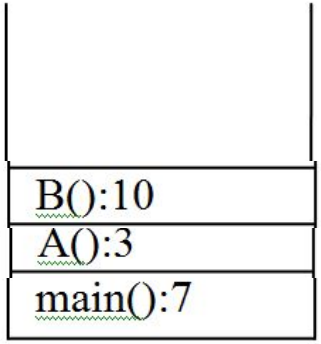
- a) адрес возврата, т.е. адрес кода, куда нужно передать управление после завершения работы функции;
- b) значения параметров функции;
- c) значения локальных переменных функции.

В момент, когда работа функции закончена, из вершины стека извлекается этот фрагмент данных и содержащийся в нем адрес возврата используется для передачи управления в вызвавшую функцию.

Заполнение стека вызовов



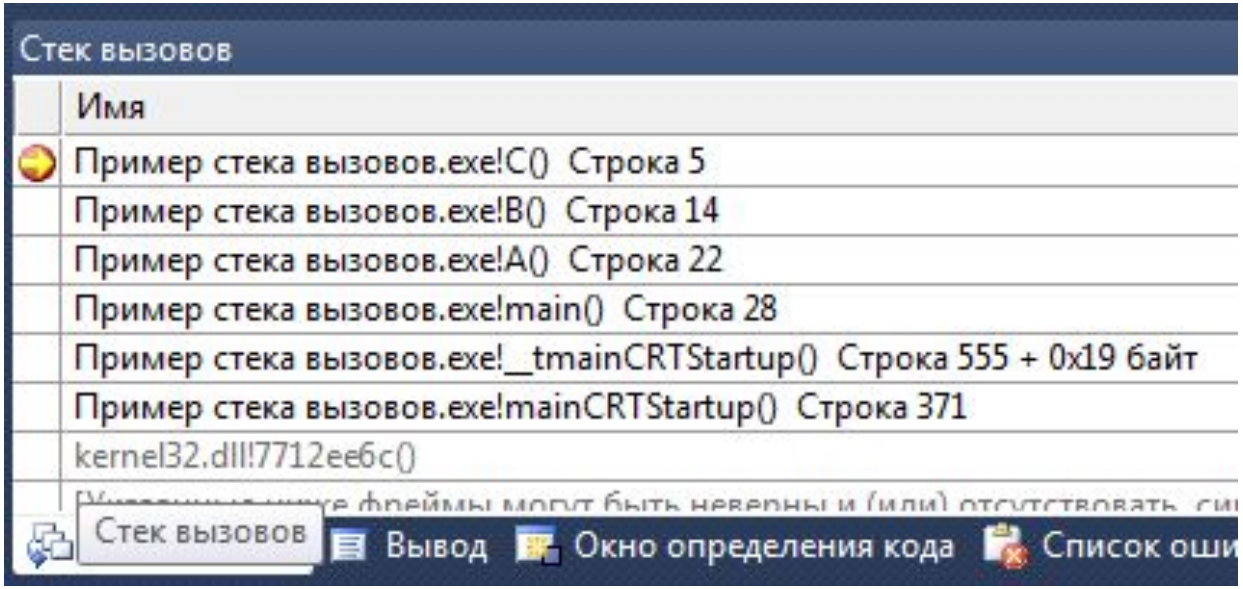
Стек вызовов



Стек вызовов в Visual Studio

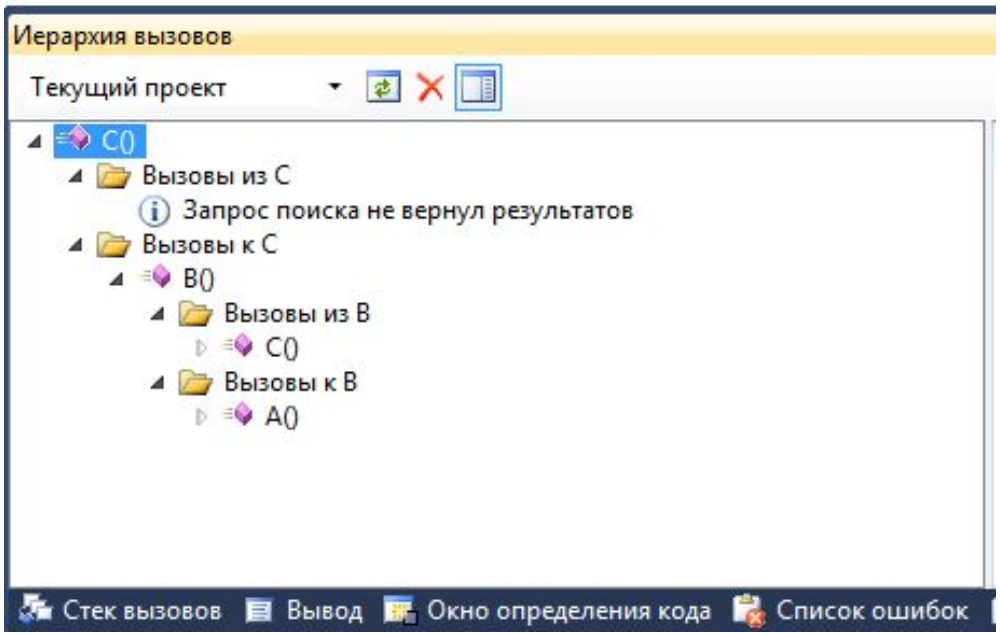
- в режиме отладки

Отладка->Окна->Стек вызовов

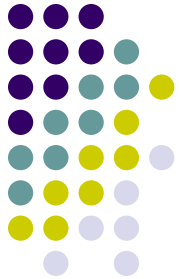


Иерархия вызовов

- выделить в исходном коде название функции;
- вызвать контекстное меню (правой кнопкой мыши);
- выбрать команду ***Показать иерархию вызовов***.

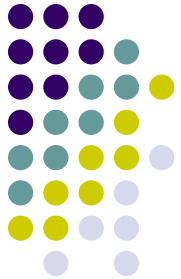


Рекурсия – это прием программирования, когда функция вызывает саму себя



- Рекурсия может быть непосредственная и косвенная

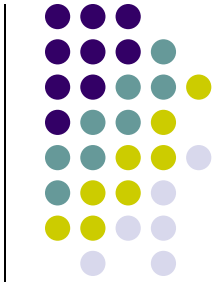
Пример. Вычисление факториала (итерации)



n!=1*2*3*…*(n-1)*n
0!=1
1!=1

```
int factorial(int n)
{
    int f=1;
    for(int i=1;i<=n;i++) //перебираем числа от 1 до n
     f*=i;                //и накапливаем их произведение
    return f;
}
```

Пример. Вычисление факториала (рекурсия)



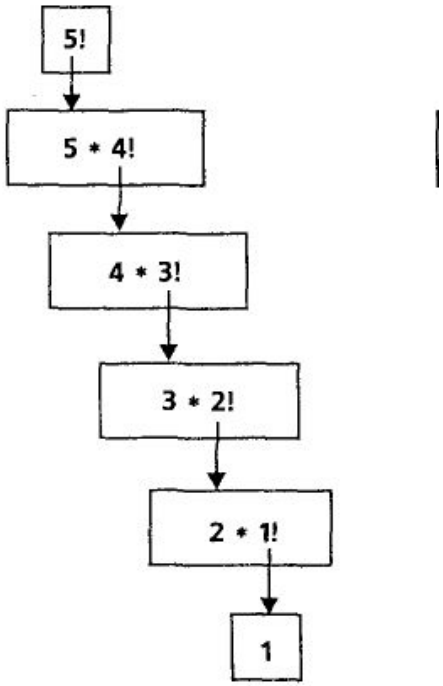
- Рекурсивное вычисление факториала основано на соотношении: n!=(n-1)!*n

```
int factorial(int n)
{
    if(n<=1) return 1; //выход из рекурсии
    return factorial(n-1)*n; //вызов с меньшим значением параметра
}
```

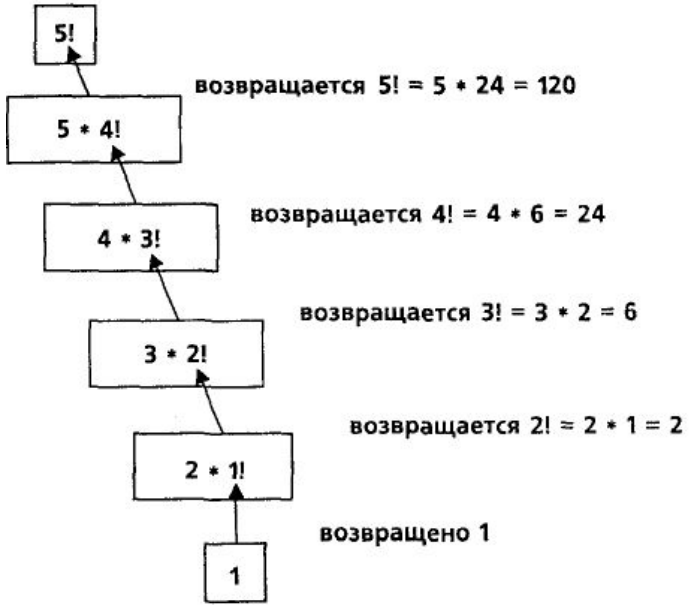
Функция вызывает свою новую копию для того, чтобы решить задачу меньшей сложности

Последовательность вызовов и возвратов при рекурсии

```
int factorial(int n)
{
        if (n<=1) return 1;
        else return n*factorial(n-1);
}
```

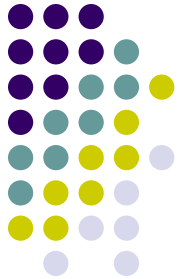


а) Процесс рекурсивных вызовов



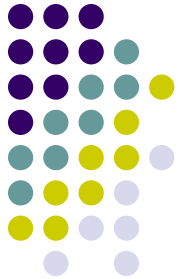
б) Значения, возвращаемые после каждого рекурсивного вызова

Правила написания рекурсивных функций



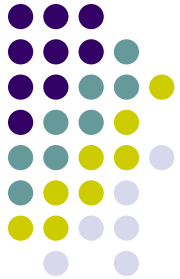
- Рекурсивная функция должна иметь терминальную ветвь, т.е. обычный возврат значения. Обычно эта терминальная ветвь располагается в начале функции, до рекурсивного вызова.
- Все рекурсивные вызовы должны вести к терминальной ветви (например, аргумент уменьшается при каждом вызове).

Рекурсия или цикл?



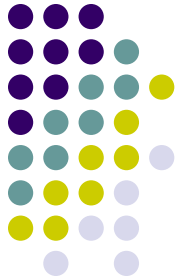
- Любую рекурсию теоретически можно заменить на итерации (цикл).
- Рекурсия требует больше памяти и времени для выполнения вычислений, чем вариант задачи с использованием цикла.
- Но рекурсия может дать существенный выигрыш в красоте кода и понятности алгоритма для некоторых задач.

Пример. Числа Фибоначчи



- Числа Фибоначчи начинаются с 0 и 1. Каждое следующее число равно сумме двух предыдущих. Т.е. последовательность чисел Фибоначчи такова:
- 0, 1, 1, 2, 3, 5, 8, 13, 21,…
- Функция должна определять число Фибоначчи с номером i.

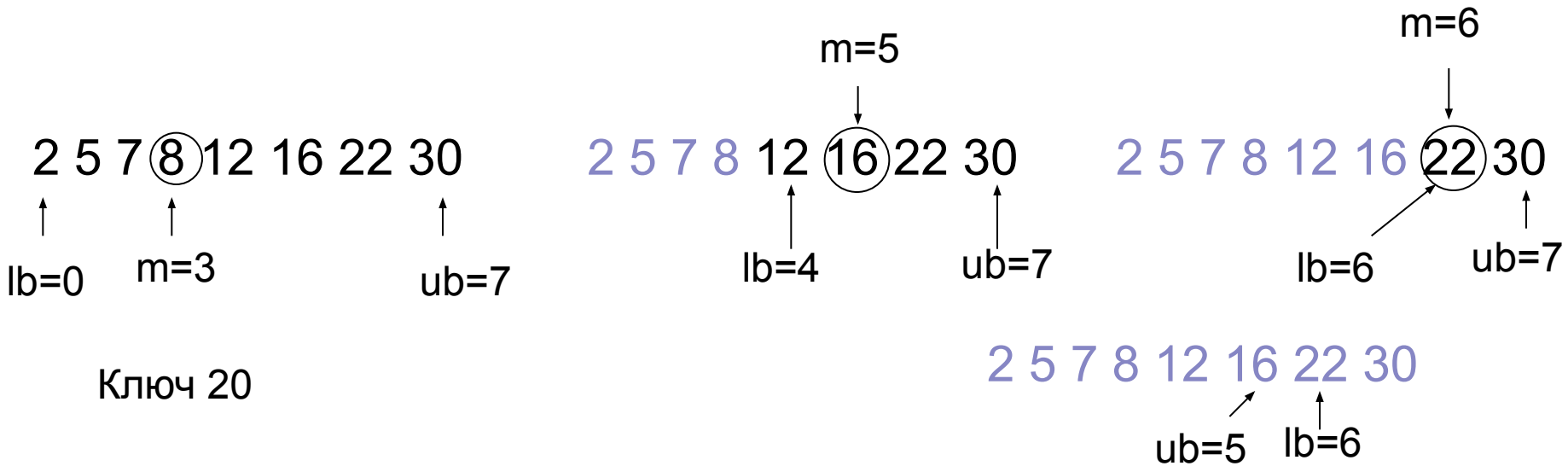
Числа Фибоначчи



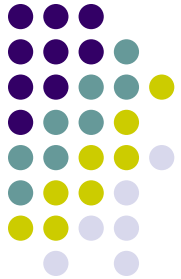
```
int fibonacci(int i)
{
    if(i<0) return -1; //проверка неправильного вызова
    if(i==1) return 0;
    if(i==2) return 1;
    return fibonacci(i-1)+fibonacci(i-2);

}
```

Бинарный поиск в отсортированном массиве

- Пусть массив упорядочен по неубыванию.
- Пусть lb=0 – нижняя граница поиска
- ub=n-1 – верхняя граница поиска.
- Середина интервала: m=(lb+ub)/2;
- Если средний элемент оказался равен искомому, возвращаем его индекс.
- Если искомый элемент меньше, чем средний (elem<a[m]), то поиск осуществляется в левой части области (правую часть отбрасываем).
- Если искомый элемент больше среднего – то в дальнейший поиск в правой половине.
- Если границы поиска пересеклись, то данного элемента в массиве нет



Пример. Бинарный поиск в отсортированном массиве

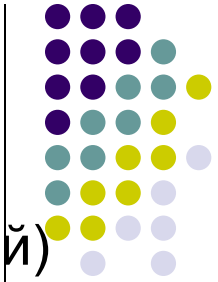


```
int binarySearch(int elem,int a[],int lb,int ub)
{  //lb- левый индекс подмассива
   // ub- правый индекс подмассива
      int m;
      if (lb>ub) return -1; //возврат, если ничего не найдено
      m=(lb+ub)/2; //середина
      if (elem== a[m]) return m; //возврат, если найдено
      if (elem>a[m]) return binarySearch(elem,a,m+1,ub); //поиск
//в подмассиве справа от середины
      if(elem <a[m]) return binarySearch(elem,a,lb,m-1);//поиск
//в подмассиве слева от середины
}
```

Первый вызов в main()

```
int ind=binarySearch(elem, a,0,n-1);
```

Быстрая сортировка (quick sort)

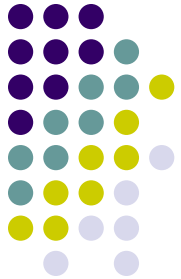


- Из массива выбирается опорный элемент (обычно средний)
- Все элементы, меньшие (либо равные) его, перемещаем в левую часть массива, а большие элементы (либо равные) – в правую часть массива
- В результате массив разбивается на две части: в первой все элементы, меньшие опорного, а во второй – большие опорного
- Затем функция сортировки вызывается для каждой из этих частей

Функция должна принимать указатель на массив и два индекса, обозначающие нижнюю и верхнюю границу сортируемой области

- void quickSort (int a[],int left,int right)

Алгоритм разделения сортируемой области на две части



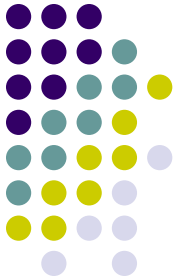
- Запомним значение среднего элемента:

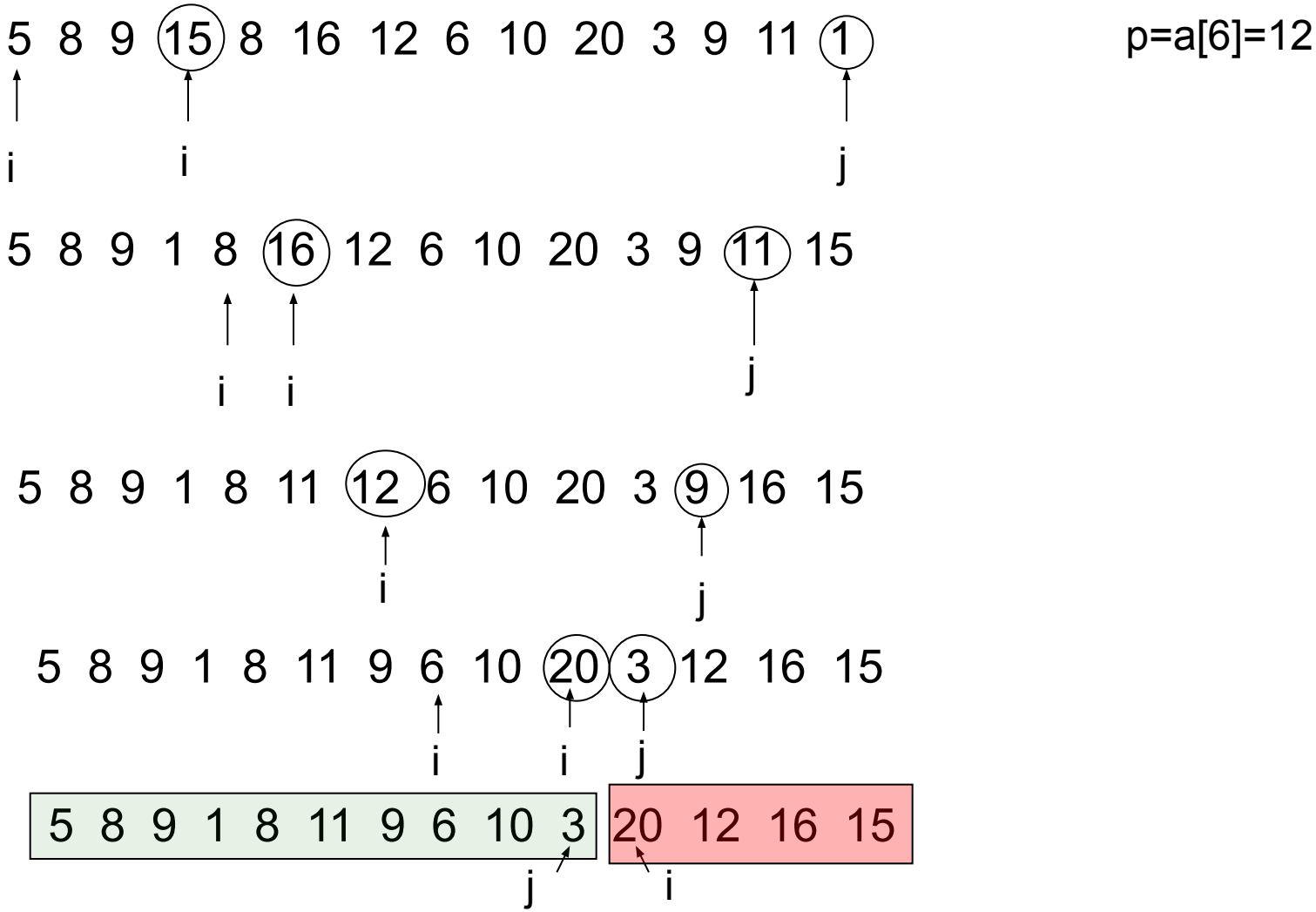
  p=a[(left+right)/2]
- Введем два текущих указателя: i и j. Сначала они указывают на левую и правую границу (i=left; j=right)
- Будем перемещать указатель i вправо (увеличивать), пока не найдем элемент, больший или равный опорному:

  while(a[i]<p) i++;
- Будем перемещать указатель j влево (уменьшать), пока не найдем элемент, меньший или равный опорному:

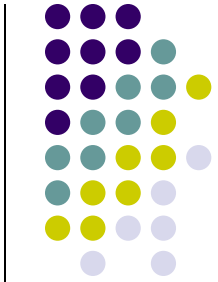
  while(a[j]>p) j--;
- Поменяем местами элементы с индексами i и j, индексы i и j продвигаем еще на шаг
- Процесс продолжается до тех пор, пока индексы i и j не “разойдутся”, т.е. пока i<=j

Пример разделения сортируемого массива





Функция быстрой сортировки

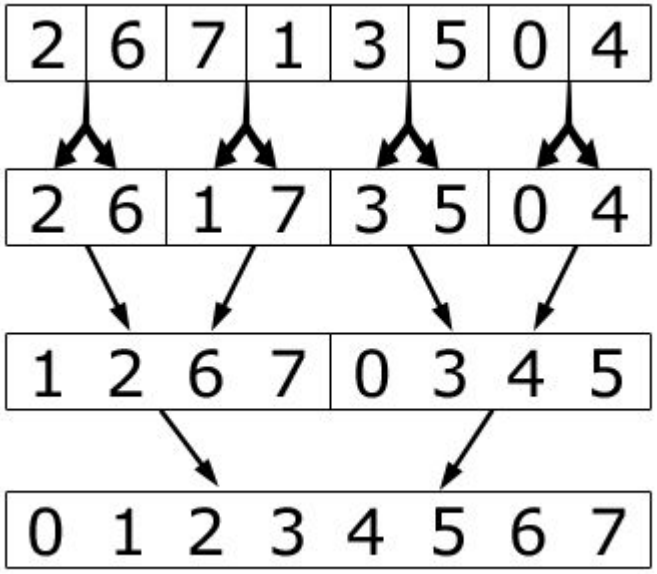


```
void quickSort(int a[],int left,int right)
{
    int i=left,j=right;
    int temp, p=a[(left+right)/2]; //опорный элемент
    //процедура разделения
    while(i<=j) //пока индексы не разошлись
    {
      while(a[i]<p) i++; //левый индекс продвигаем до элемента, >= опорного
      while(a[j]>p) j--; //правый индекс продвигаем до элемента, <= опорного
      if(i<=j) //если индексы еще не разошлись
      {
        temp=a[i]; a[i]=a[j]; a[j]=temp; //меняем местами
        i++; j--; //продвигаем индексы на шаг
      }
    }
    //рекурсивные вызовы
   if(j>left) quickSort(a,left,j); //отсортировать левый подмассив
   if(i<right) quickSort(a,i,right); //отсортировать правый подмассив
}
```

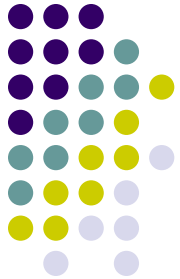
Первый вызов: quickSort(a,0,n-1);

Сортировка слиянием (merge sort)

- Массив рекурсивно разбивается пополам до тех пор, пока размер очередного подмассива не станет равен 1
- Каждая половина сортируется отдельно
- Затем выполняется процедура слияния двух упорядоченных подмассивов в один



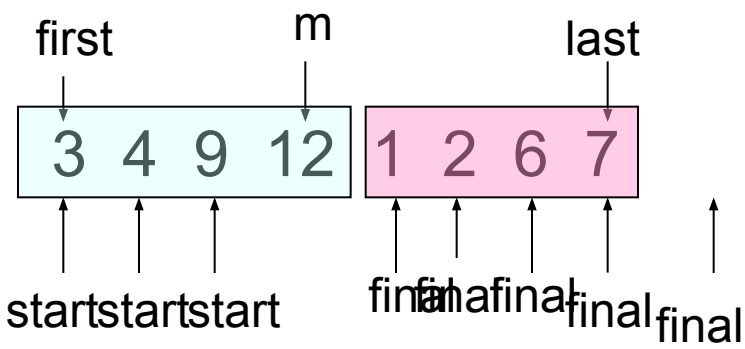
Основная функция сортировки слиянием

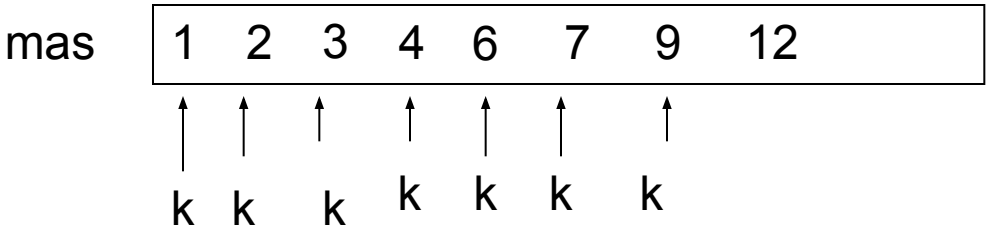


```
void mergeSort(int *a, int first,int last){
    if (first>=last) return;
    //сортировка левой половины
    mergeSort(a,first,(first+last)/2);
    //сортировка правой половины
    mergeSort(a,(first+last)/2+1,last);
    merge(a,first,last); //слияние двух частей
}
```

Функция слияния двух упорядоченных половин массива

```
void merge(int *a, int first, int last){
    int mas[20]; //вспомогательный массив
    int m=(first+last)/2; //середина массива
    int start=first;  //начало первой части массива
    int final=m+1;  //начало второй части массива
    int k=0; //индекс в результирующем масиве
    while(start<=m&&final<=last)
        if(a[start]<a[final]){
            mas[k]=a[start]; k++; start++;    }
        else{
            mas[k]=a[final]; k++; final++;   }
    while(start<=m){
     mas[k]=a[start]; k++; start++; }
    while(final<=last){
     mas[k]=a[final]; k++; final++; }
    for(int i=0;i<k;i++)
     a[first+i]=mas[i];  //переписываем в исходный массив
    }
}
```


first
m
last
3 4 9 12
1 2 6 7
start
start
start
final
final
final
final
final


mas
1 2 3 4 6 7 9 12
k k k k k k k

Задача о Ханойской башне

- В одной из древних легенд говорится следующее :
- "... В храме Бенареса находится бронзовая плита с тремя алмазными стержнями. На один из стержней Бог при сотворении мира нанизал 64 диска разного диаметра из чистого золота так, что наибольший диск лежит на бронзовой плите, а остальные образуют пирамиду, сужающуюся кверху. Это - башня Брамы. Работая день и ночь, жрецы переносят диски с одного стержня на другой, следуя законам Брамы :
- Диски можно перемещать с одного стержня на другой только по одному;
- Нельзя класть больший диск на меньший.
- Когда все 64 диска будут перенесены с с одного стержня на другой, и башня, и храмы, и жрецы-брамины превратятся в прах и наступит конец света.”
- Нужно написать программу, которая выводит инструкцию по перенесению дисков со стержня на стержень и подсчитать количество таких действий.
- Входным параметром программы является количество дисков, которые изначально лежат на стержне №1

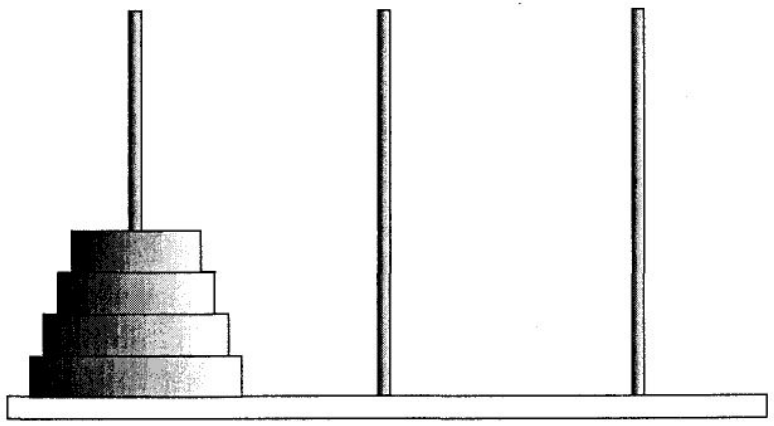
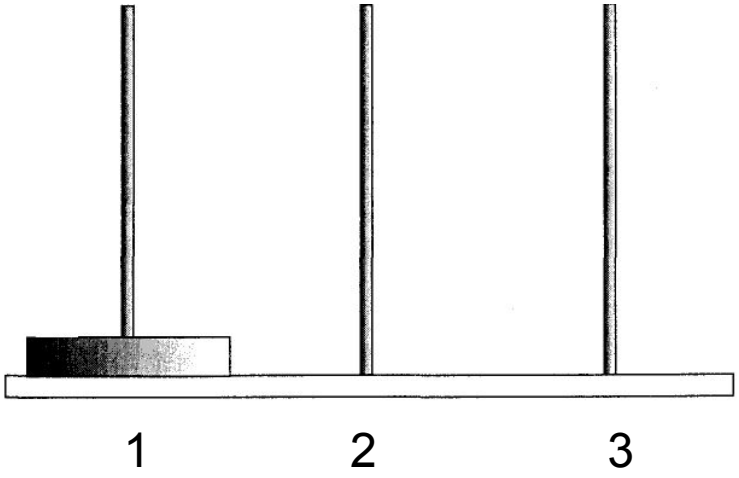


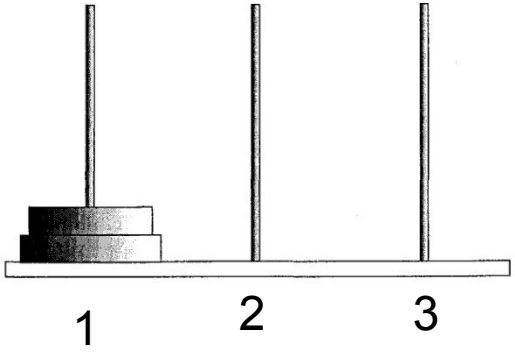
Рис. 5.18. Ханойская башня для случая четырех дисков

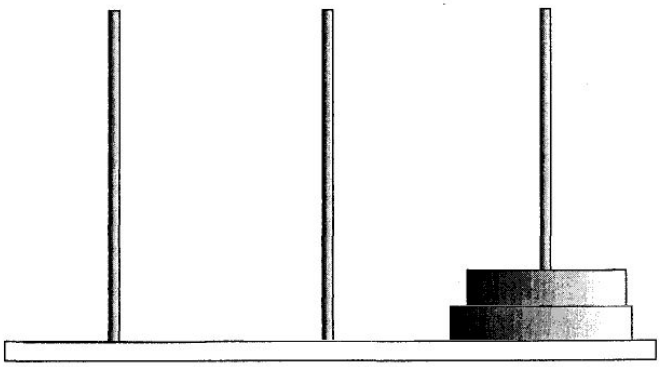
Частный случай – один диск



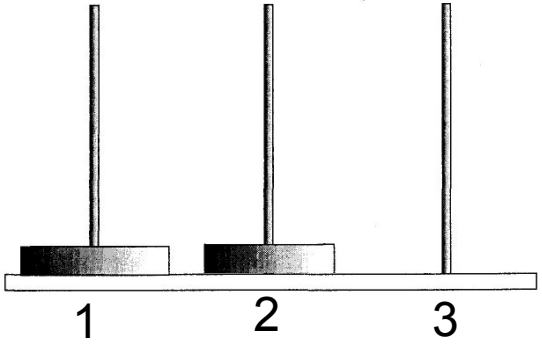
1->3

Частный случай – два диска





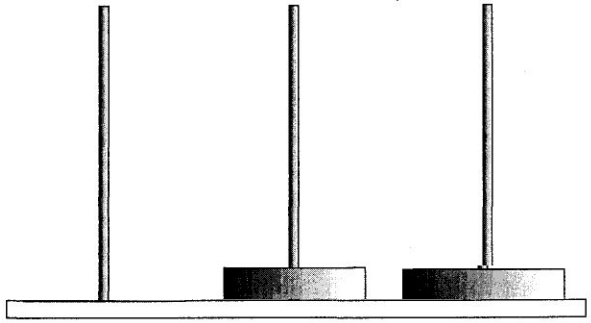
1->2



1->3

Перенести вершину на вспомогательный стержень,

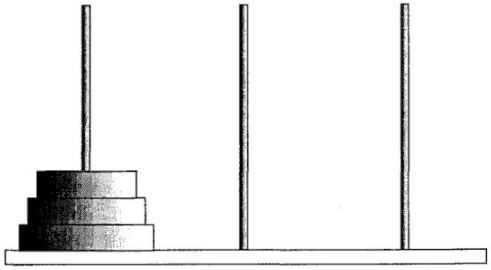
Перенести нижний диск на целевой стержень

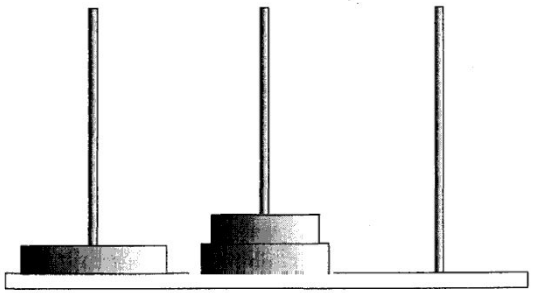


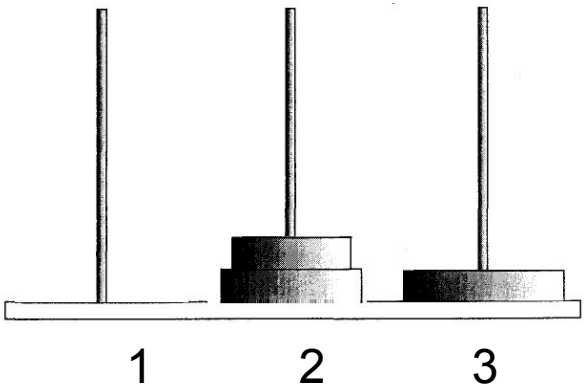
2->3

Перенести вершину на целевой стержень

Частный случай- три диска





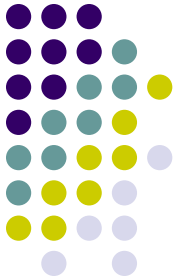


Переместить верх пирамиды (n-1) диск на стержень 2, используя стержень 3 как вспомогательный

1->3

Задача аналогична предыдущей: нужно перенести 2 диска со стержня 2 на стержень 3, используя стержень 1 как вспомогательный

Встраиваемые функции

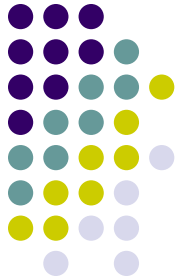


- Подставляется в текст программы на этапе компиляции.
- Не увеличивается время выполнения программы, т.к. нет передачи управления функции и обратно

```
inline double sqr(double x)
{ return x*x; }
```

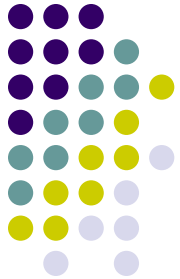
- Этот спецификатор носит рекомендательный характер и выполняется компилятором по мере возможности.

Причины игнорирования компилятором спецификации inline



- Слишком большой размер функции.
- Функция является рекурсивной.
- Функция повторяется в одном и том же выражении несколько раз.
- Функция содержит цикл, switch или if.

Макросы



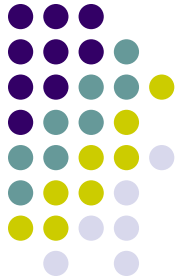
#define Имя_макроса(Параметры) (Выражение)

```
#define SQR(X)  (X)*(X)
void main()
{ int y;
  y=SQR(4);
}
```

До начала компиляции преобразуется в код:

```
void main()
{ int y;
  y=(4)*(4);
}
```

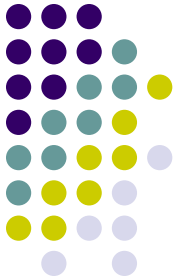
Препроцессор выполняет текстовую подстановку!



- #define SQR(X) (X)*(X)
- y=SQR(a+2); → y=(a+2)*(a+2);

- #define SQR(X) X*X
- y=SQR(a+2); → y=a+2*a+2;

Перегрузка функций – определение нескольких функций с одинаковым именем, которые выполняются по-разному



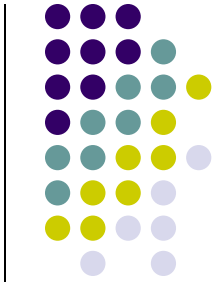
- Функция, которая вычисляет площадь прямоугольника в $см^2$. Параметрами являются стороны прямоугольника в сантиметрах.

```
float areaRectangle (float a, float b)
{ return a*b;}
```

- Функция, которая вычисляет площадь прямоугольника в $см^2$. Параметрами являются количество м и см для каждой стороны. Например, 2м 43 см и 1м 84 см.

```
float areaRectangle(float a_m, float a_cm, float b_m, float b_cm)
{return (a_m*100+a_cm)*(b_m*100+b_cm); }
```

Вызывается на исполнение та функция, список параметров которой соответствует фактическим параметрам



```
void main()
{
cout<<“S1= “<<areaRectangle(32,43)<<“\n”; //вызов варианта 1
cout<<“S2= “<<areaRectangle(2,43,1,84)<<“\n”; //вызов варианта 2
}
```

Сигнатура функции – это комбинации имени функции и ее параметров

- Параметры функций могут отличаться:

А) количеством

Б) типом

В) порядком следования

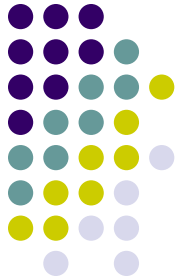
Это все ***разные*** функции:

```
void print(int a, int b);
void print(double a, double b);
void print(int a, double b, double c)
```

```
print(3,5);
print(3.6, 0.2);
print(4, 0.3, 65.2);
```

Компилятор определяет, какую именно функцию нужно вызвать, анализируя набор фактических параметров.

Пример. Написать функцию вычисления максимума из нескольких чисел. Функция должна работать для двух или трех чисел типа int и double



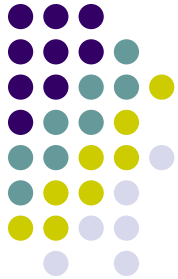
В перегруженных функциях лучше не использовать параметры по умолчанию!

```
double multy (double x) { return x * x * x; }
double multy (double x, double y)
{ return x * y * y; }
double multy (double x, double y, double z)
{ return x * y * z; }}
```

- multy(0.4);
- multy(4.0, 12.3);
- multy(0.1, 0.2, 6.5);

```
double multy (double a=1.0, double b=1.0, double c=1.0, double d=1.0)
{ return a * b + c * d; }
```

```
multy(0.1, 1.2);      //два параметра или четыре(два по умолчанию)?
```



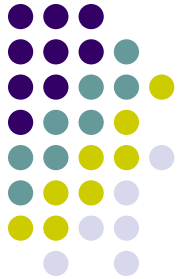
Шаблоны функций в языке C++ позволяют создать общее определение функции, применяемой для различных типов данных.

Например, нужно создать функцию определения модуля для чисел типа int и double. Используя перегрузку, получаем:

```
int myAbs(int x)
{ int y= (x>=0)?x:-x;
  return y;
}
double myAbs(double x)
{double y=(x>=0)?x:-x;
  return y;
}
```

Используем шаблон:

```
template <typename MyT>
MyT myAbs(MyT x)
{MyT y=(x>=0)?x:-x;
 return y;
}
```



```
template <typename myT>
myT Abs (myT n)
{ return n < 0 ? -n : n; }
```

- template – начало конструкции шаблона.
- typename означает “имя типа”
- myT (можно использовать любой идентификатор) является параметром типа.
- Конкретный тип myT определяется при вызове функции.

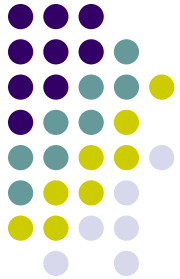
```
cout << "Result - 5 = " << Abs(-5);
```

```
cout << "Result  5.5 = " << Abs(5.5);
```

```
int Abs (int n)
{ return n < 0 ? -n : n; }
```

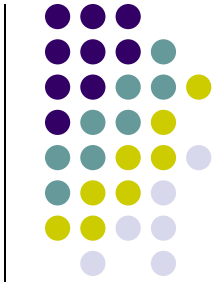
```
double Abs (double n)
{ return n < 0 ? -n : n; }
```

Генерация кода функции по шаблону в процессе компиляции



- Описание шаблона в тексте программы не вызывает генерацию кода само по себе. Код функции создается в тот момент, когда компилятор встречает вызов функции с параметром конкретного типа.
- Следующий вызов с тем же типом данных параметра не вызовет генерацию новой функции, а использует ее уже существующую копию.
- Если же в очередном вызове функции тип передаваемого параметра не совпадет ни с одним из предыдущих вызовов, то компилятор создаст новую версию функции.

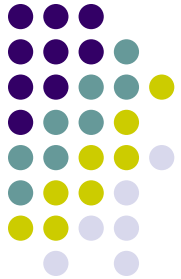
Использование различных типов в шаблоне



```
template <typename T, typename T1>
T myMax(T a, T1 b)
{
  return (a>b)?a:b;
}
```

Результат функции имеет тот же тип, что и первый параметр. Второй параметр может быть любого типа. Варианты вызовов:

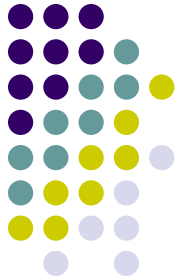
```
int i=2,j=3,k;
double x=3.5, y=2.5,z;
k=myMax(i,j);
k=myMax(i,x);    //нежелательно, т.к. потеря данных
z=myMax(x,y);
z=myMax(x,j);
```



- ВНИМАНИЕ!!! Каждый параметр типа, встречающийся внутри угловых скобок, должен ОБЯЗАТЕЛЬНО появляться в списке параметров функции. В противном случае произойдет ошибка на этапе компиляции.

```
template <typename T1, typename T2>
T1 Max(T1 A , T1 B)
{ return A > B ? A : B; }
// ОШИБКА! список параметров должен включать T2 как
   //параметр типа.
```

Объявление обычной функции переопределяет шаблон

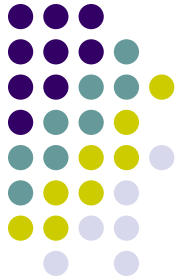


- Если в тексте программы есть описание обычной функции и шаблона с одним и тем же именем, то используется обычная функция

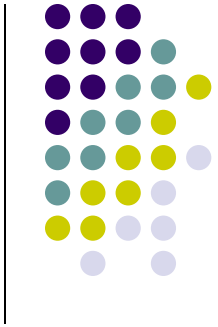
```
template <typename T>
T sum(T x, T y)
{ return x+y;}

void sum(char a,char b)
{ cout<<a <<“ и “<<b; }
```

```
sum(‘x’,’y’); //вызывается обычная функция

z=sum(5,6); //создается функция по шаблону
```

Реализовать шаблон функции определения максимума в массиве. Функция должна работать с массивами любого типа (int, double, char,…)



Адрес функции

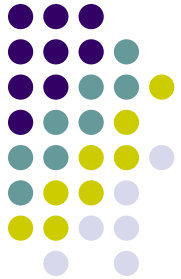


- Функция имеет физическое месторасположение в памяти, адрес начала которого иначе называется “точка входа”.
- Имя функции без скобок и параметров является указателем-константой на функцию. Значением его служит адрес размещения функции в памяти (точка входа).
- Можно провести аналогию с массивом: имя массива – адрес начала массива, который является константой.
- Например, если имеется функция

int think(char);

- то think – адрес этой функции в памяти.

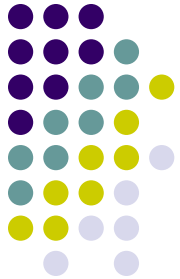
Указатель на функцию



тип_функции (*имя_указателя)(спецификация_параметров);

```
int think(char); //прототип функции
int  (*funPtr)(char); //объявление указателя на функцию
funPtr=think; //указатель получает значение адреса функции

//2 способа вызова функции через указатель
k=(*funPtr)('*'); //эквивалентно вызову k=think('*');
k=funPtr('*');    // также вызов k=think('*');

int *funPtr (char); //ошибка!
```

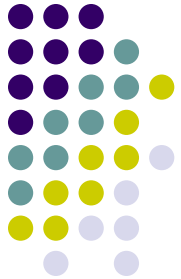
Объявление указателя на функцию



```
//две функции одинакового типа и набора параметров
int sum(int a,int b)
{  return a+b;  }
int sub(int a,int b)
{  return a-b;  }
void main()
{
//объвление указателя на функцию
int (*funPtr)(int,int);
int a=5,b=6;
funPtr=sum; //указатель получает значение адреса начала функции суммы
cout<<a<<" + "<<b<<" = "<<funPtr(a,b)<<"\n"; //обращение к sum через //указатель
system("pause");
}
```

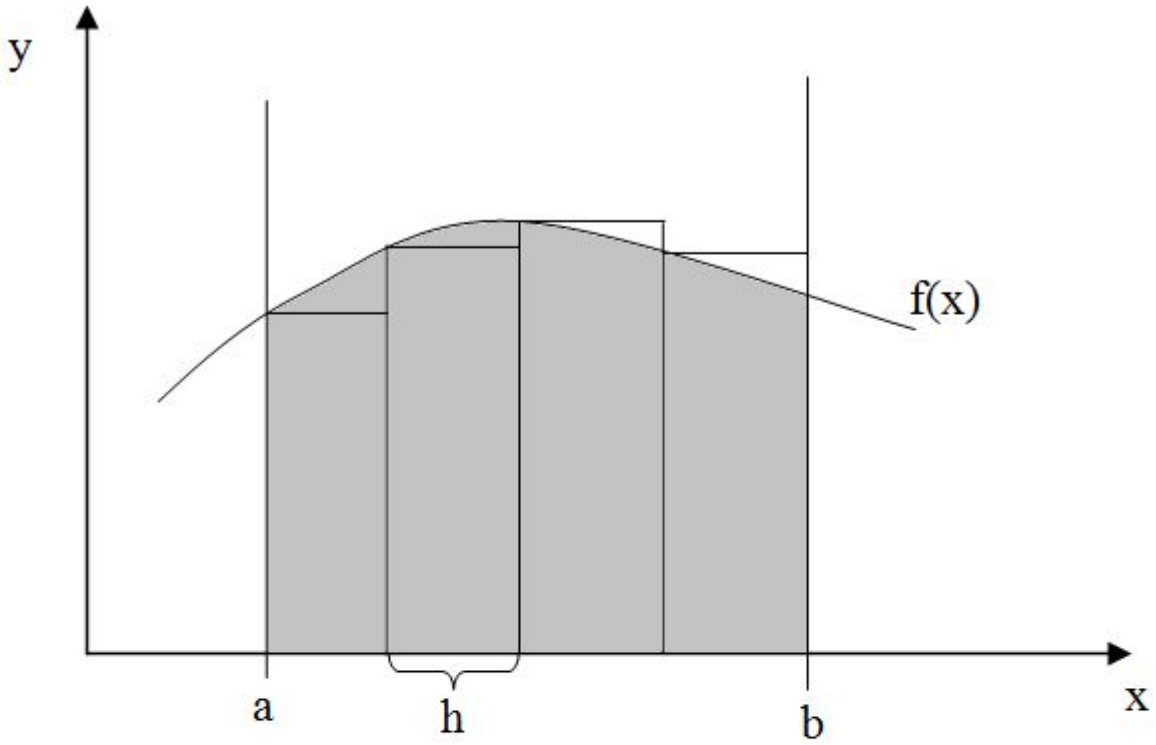
(*funPtr)(a,b) //второй способ вызова

Использование указателей на функцию



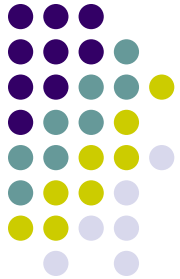
- Указатель на функцию используется как аргумент другой функции
- Создается массив указателей на функции для реализации меню

Расчет определенного интеграла



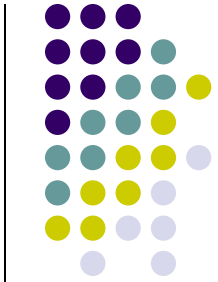
$$\int_a^b f(x)dx \approx \sum_i f(x_i) \cdot h$$

Пример передачи в функцию указателя на функцию



```
//a и b – пределы интегрирования
//n – число интервалов разбиения
// pf – указатель на подынтегральную функцию
double integral(double a, double b, int n, double(*pf)(double))
{
    double s=0, h=(b-a)/n; //h- ширина интервала
    for(double x=a; x<b+h; x+=h)
            s+=pf(x)*h;
    return s;
}

cout<<"Интеграл синуса от 0 до Пи= " <<integral(0.,PI,40,sin)<<"\n";
```

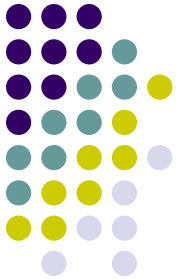
Пример: создание меню работы с данными



- Пусть есть ряд функций с одинаковым типом и набором параметров:

```
void vvod(void)
{
    cout<<"Ввод данных\n";
}
void udal(void)
{
    cout<<"Удаление данных\n";
}
void prosmotr(void)
{
    cout<<"Просмотр данных\n";
}
void quit(void)
{
    cout<<"Конец работы\n";
    system("pause");
    exit(0);
}
```

Пример использования массива указателей на функции



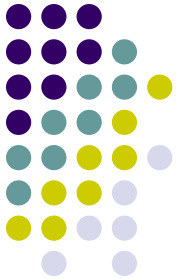
- Объявляем массив указателей на функции

```
void(*funptr[4])(void)={vvod,udal,prosmotr,quit};
```

- Запрашиваем выбор пользователя и вызываем соответствующую функцию из массива:

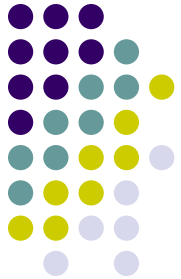
```
int answ;
do
{
    answ=menu(); //получение выбора пользователя
    funptr[answ-1](); //вызов соответствующей функции
}
while (true); //бесконечный цикл - выход по exit
```

Функция меню:вводит число от 1 до 4



```
int menu(void)
{
    setlocale(LC_ALL,"rus");
    int answer;
    do
    {
      cout<<"1 - Ввод\n";
      cout<<"2 - Удаление\n";
      cout<<"3 - Просмотр\n";
      cout<<"4 - Выход\n";
      cout<<"Ваш выбор: ";
      cin>>answer;
      if(answer<1||answer>4)
            cout<<"Неверный выбор!\n";
    }
    while (answer<1||answer>4);
    return answer;
}
```

Пример массива указателей на функции

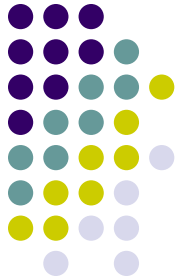


```
float (*ptrArray[6])(char);

float a = (*ptrArray[2])('f');
```

или

```
float a = ptrArray[2]('f');
```

Альтернативный синтаксис для функций (хвостовой возвращаемый тип C++11)

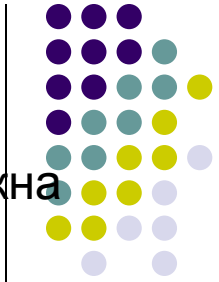


Прототип double h(int x, float y);
может быть записан с помощью альтернативного синтаксиса следующим образом:

```
auto h(int x, float y) -> double;
```

Аналогично:

```
template<typename T1, typename T2>
auto gt(T1 x, T2 y) -> decltype (x + y)
{
    return x + y;
}
```



```
//шаблон для создания функции максимума из двух чисел.
//функция может принимать два числа любого типа, а возвращать должна
//тот тип, который является старшим из них
//работает только в 13-й Visual Studio
#include <iostream>
using namespace std;
template <typename MyT1, typename MyT2>
auto mymax(MyT1 a, MyT2 b)->decltype(a+b)
{
    return a>b?a:b;
}
void main()
{
    setlocale(LC_ALL,"rus");
    int i=5,j=6,k;
    double x=5.5,y=6.6,z;
    k=mymax(i,j); //результат - целый
    z=mymax(x,y); //результат - вещественный
    z=mymax(i,x); //результат - вещественный
    z=mymax(x,i); //результат - вещественный
    system("pause");
}
```