ФУНКЦИИ

1. Основные понятия.
2. Вызов функции.
3. Передача параметров функции.
4. Функции с переменным числом параметров
5. Область действия переменных.
6. Классы памяти

В лекции рассматриваются особенности использования функций при составлении программ на языке С++.

При разработке программ обычно ***основной алгоритм*** разбивается на более мелкие части – ***подалгоритмы,*** до тех пор пока не получатся относительно простые для решения задачи. Затем эти подалгоритмы перекладываются на соответствующий язык программирования в виде подпрограмм.

1. Основные понятия

В языке С++ подпрограммы реализуются посредством функций.

Функция – это логически самостоятельная именованная часть программы, предназначенная для выполнения определенной задачи, которой могут передаваться параметры и которая может возвращать какое-то значение.

Функция — это именованная последовательность инструкций. Она может возвращать результат, который также называется возвращаемым значением. (Б.Страуструп)

Использование функций позволяет:

- упростить процесс программирования;
- сократить время разработки программ;
- повысить наглядность программ;
- уменьшить количество ошибок при программировании.

Функции языка С++ могут быть:

- ***головная***: имеет имя ***_tmain( )***, выполнение программы начинается всегда с этой функции;
- ***стандартные***: функции, встроенные в язык и доступные для любой программы. Например, ***sin(x)***, ***_getch(), rand()*** и т. д. Стандартные функции доступны программисту через библиотеки функций (расширение ***.lib***), где они хранятся в откомпилированном виде и подключаются к С++-программе во время редактирования связей. Объявления (прототипы) всех стандартных функций размещены в файлах включения (расширение ***.h***). Эти файлы должны включаться в программу посредством директивы препроцессора ***#include***. Например, если в программе предполагается применять тригонометрические функции (***cos(x),..***), то директивой ***#include*** в нее надо включить соответствующий заголовочный файл с прототипами этих функций:

```
#include <math.h>
...
тело программы
```

- ***пользовательские***: функции, разработанные пользователем для реализации некоторого подалгоритма. Они доступны только той программе, в которой записаны.

С использованием функций в языке С++ связаны три понятия:

определение функции;
объявление функции;
вызов функции.

```
int sum(int, int);/* Объявление функции - прототип */
int main()
{ int a,b,c;
cout << "a=";
cin >> a;
cout << "b=";
cin >> b;

c=sum(a, b); // Вызов функции
cout << "\n sum = " << c;
return 0;
}
int sum(int a, int b)
{ // Определение функции
return a+b;
          }
```

1.1. Определение функции

Определение функции имеет следующий формат:

// заголовок функции
[тип_результата] имя(список_формальных_параметров)
// тело функции
{
описание данных
операторы
[return выражение;]
}

Тип_результата – тип возвращаемого значения. В случае отсутствия спецификатора типа предполагается, что функция возвращает целое значение *(int).* Если функция не возвращает никакого значения, то на месте типа записывается спецификатор *void.* В списке параметров для каждого параметра должен быть указан тип. При отсутствии параметров список может быть пустым или иметь спецификатор *void.*

Имя функции – уникальный идентификатор, по которому будет производиться ссылка на эту функцию; имя функции не может совпадать ни с именем уже определенной функции, ни с именем уже существующей переменной.

Список_формальных_параметров – список параметров (или аргументов), передаваемых в функцию, содержащий любую комбинацию элементов «тип + имя», разделяемых запятыми; круглые скобки обязательны.

Список формальных параметров может заканчиваться запятой и многоточием (,...). Это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет по крайней мере столько аргументов, сколько формальных параметров задано перед последней запятой. Функции может быть передано больше аргументов, чем задано до запятой и многоточия. Над такими аргументами не производится контроль типов.

Поле «*список_формальных_параметров*» – не обязательная часть в определении функции. Если в функцию не передаются никакие аргументы, это поле пустое или содержит ключевое слово ***void***.

Формальный параметр может быть любого основного типа, структурой, объединением, перечислением, указателем или массивом.

Если формальный параметр представлен в списке, но не объявлен, то предполагается, что параметр имеет тип ***int***. Например:

```
int f1( x, y)   // переменные x и y не объявлены,
                //считается, что они имеют тип int
...
```

Имена формальных параметров используются в теле функции в качестве ссылок на передаваемые величины.

Параметры разделяются на формальные и фактические. ***Формальные параметры*** – это те параметры, которыми оперирует функция и которые указаны в ее заголовке. ***Фактические параметры*** – это параметры, значения которых присваиваются соответствующим формальным параметрам при вызове функции. Таким образом, при изменении формальных параметров фактические параметры не изменяются.

Тело функции представляет собой последовательность объявлений и операторов, определяющих действие функции. Важным оператором тела функции является оператор возврата в точку вызова *[return выражение];* Оператор *return* имеет двойное назначение. Он обеспечивает возврат в вызывающую функцию и может использоваться для передачи вычисленного значения функции. В теле функции может быть несколько операторов *return,* но может не быть и ни одного. В последнем случае возврат в вызывающую программу происходит тогда, когда выполнен последний оператор тела функции.

Встретив определение функции, компилятор создает самостоятельный фрагмент кода программы, который на этапе компоновки объединяется с другими функциями.

Таким образом, определение функции задает имя, формальные параметры и тело функции. Оно может также определять тип возвращаемого значения функции.

1.2. Объявление функции (прототип)

Прототип функции – это явное объявление функции, которое предшествует ее определению.

Прототип функции указывается до вызова функции вместо ее описания для того, чтобы компилятор мог выполнить проверку соответствия типов аргументов и параметров. Прототип функции по форме такой же, как и заголовок функции, в конце его ставится «;». Параметры функции в прототипе могут иметь имена, но компилятору они не нужны.

Компилятор использует прототип функции для сравнения типов аргументов с типами параметров. Язык С++ не предусматривает автоматического преобразования типов в случаях, когда аргументы не совпадают по типам с соответствующими им параметрами, т. е. язык С++ обеспечивает строгий контроль типов.

При наличии прототипа вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией. Часто прототипы функций помещают в отдельные файлы − ***заголовочные файлы*** (эти файлы имеют расширение .h).

Прототип функции имеет следующий формат:

[тип_результата] имя(список формальных параметров|void);

Например:

```
double f1(int, double);
void f2(double a1, char a2);
```

Таким образом, *прототип функции* задает имя функции, типы и число формальных параметров, тип значения, возвращаемого функцией.

В отличие от формата определения функции в прототипе функции имена параметров в списке формальных параметров указывать нет необходимости.

Если прототип функции не задан, а встретился вызов функции, компилятор создает неявный прототип, исходя из информации, доступной из вызова функции. Тип возврата создаваемого прототипа функции ***int***. На основании типов и числа фактических аргументов создается список объявлений формальных параметров.

2. Вызов функции

Вызов функции может быть оформлен в виде простого оператора, если у функции отсутствует возвращаемое значение, или в виде оператора-выражения, если существует возвращаемое значение.

В первом случае оператор имеет следующий формат:

имя_ функции (список_фактических_аргументов);

во втором случае выражение записывается следующим образом:

x = имя_функции (список_фактических_аргументов);

Значение вычисленного выражения является возвращаемым значением функции. Возвращаемое значение передается в место вызова функции и является результатом ее работы.

Число и типы фактических аргументов должны совпадать с числом и типом формальных параметров функции.

Аргумент или, другими словами, фактический параметр может быть любой величиной основного типа, структурой, перечислением, объединением или указателем.

Вызов функции сводится к следующим действиям:

– считываются или вычисляются значения фактических аргументов;

– управление передается функции, имя которой указано в обращении;

– значения фактических аргументов присваиваются последовательно формальным параметрам, начиная с первого;

– вызываемая функция выполняется последовательно, оператор за оператором;

– вычисляется выражение в операторе ***return;***

– управление передается в вызывающую функцию на место обращения к функции; сюда передается значение, вычисленное функцией в операторе ***return;***

– если ***return*** отсутствует, то возврат из функции к инициатору ее вызова происходит при обнаружении закрывающей фигурной скобки.

3. Передача параметров функции.

В общем случае существуют два стиля передачи параметров функции:

– вызов функции с передачей значений;

– вызов функции с передачей по ссылке (адресов переменных).

Вызов функции с передачей значений (call-by-value) осуществляется путем записи в стек (область памяти для временного хранения данных) **копий** переменных, перечисленных в списке фактических параметров. При этом изменения, внесенные в параметры подпрограммы, не влияют на аргументы, используемые при ее вызове.

Пример 1. Составить программу, в которой использовать функцию для определения куба числа, используя вызов функции с передачей значения.

Решение:

```
// Прототип функции Cub()
double Cub(double r);

// Головная функция программы
int _tmain()
{
double x,y=3.14;
x=Cub(y);
return 0;
}

//Определение функции Cub()
double Cub(double r)
```

```
{
   return r*r*r;
}
```

При выходе из функции стек «очищается», следовательно, теряются и значения параметров, переданных функции. Поэтому в теле функции нельзя изменить значения переданных ей переменных, так как функция работает с их копиями. Схематично данный процесс можно представить следующим образом:

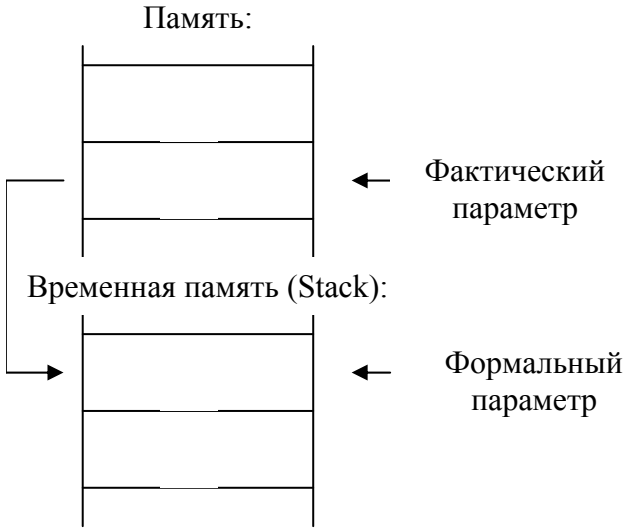


Рисунок 1. Вызов функции с передачей значений

Рассмотрим в качестве примера функцию, обменивающую значения двух переменных:

```
int _tmain()
{...
sw( x, y);       // Вызов функции sw()
...
}

void sw(int a,int b)     // Определение функции sw()
{
int c=a;
a=b;
b=c;
}
```

Здесь, поскольку функция ***sw()*** работает с копиями передаваемых переменных ***a*** и ***b***, размещенных в стеке, то она не выполняет своей задачи. Она обменивает значения этих параметров, не оказывая влияния на реальные значения аргументов ***a*** и ***b***.

Вызов функции по значению применяется в тех случаях, когда общий объем передаваемых в функцию параметров невелик и функция не возвращает большой объем данных.

Передача аргументов в функцию по ссылке

В языке C++ введѐн новый составной тип данных – ссылочная переменная. Ссылка представляет собой имя, которое является псевдонимом для ранее объявленной переменной. Для объявления ссылочной переменной используется символ **&**

```
int x;
```

```
int &r = x; // создание независимой ссылки
x = 10; // эти две инструкции
r = 10; // идентичны
r = 100; // x=100;
// здесь дважды печатается число 100
cout << x << '\t'<< r <<"\n";
```

независимые ссылки лучше не использовать, поскольку чаще всего им можно найти замену, а их неаккуратное применение может исказить ваш код.

Основное назначение ссылок – использование в качестве формальных параметров функций. Используя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями.

- При передаче по ссылке в вызываемую функцию передаются ссылки на переменные – аргументы.
- Копии аргументов с именами формальных параметров не создаются.
- Вместо этого, по ссылке обеспечивается доступ к участкам памяти, занимаемым аргументами.
- Говоря другими словами, обработка аргументов ведется «на месте».
- Чтобы передать значения аргументов по ссылке, в качестве формальных параметров указывают переменные –ссылки.
- Все изменения формальных параметров, сделанные в функции, происходят с аргументами.
- Если в качестве аргумента по ссылке передается константа, то формальный параметр-ссылка должен быть объявлен с модификатором const.

```
void swapargs(int &x, int &y);
int main()
{
int i, j;
i = 10;
j = 19;
cout << "i: " << i << ", ";
cout << "j: " << j << "\n";
swapargs(i, j);
cout << "После перестановки: ";
cout << "i: " << i << ", ";
cout << "j: " << j << "\n";
getch();
return 0;
}
void swapargs(int &x, int &y)
{ int t;
t = x; x = y; y = t;
}
```

Вызов функции с передачей адресов предполагает, что в качестве параметров функции передаются не копии переменных, а копии адресов переменных. Используя адреса, функция осуществляет доступ к нужным ячейкам памяти для изменения их значений. Поэтому в приведенной выше программе в качестве параметров требуется передать функции указатели на переменные. В теле же самой функции для доступа и изменения значения переменных необходимо использовать операцию разадресации. В этом случае программа будет иметь вид:

```
int _tmain()
{...
sw( &x, &y);            // Вызов функции sw()
...
}
void sw(int *a,int *b)   // Определение функции sw()
{
int c=*a;
*a=*b;
*b=c;
}
```

Обратите внимание на то, что объявление x и y ссылочными параметрами избавляет вас от необходимости использовать оператор "*" при организации обмена значениями. Как уже упоминалось, такая "навязчивость" с вашей стороны стала бы причиной ошибки. Поэтому запомните, что компилятор автоматически генерирует адреса аргументов, используемых при вызове функции swap (), и автоматически разыменовывает ссылки x и y.

Итак, подведем некоторые итоги. После создания ссылочный параметр автоматически ссылается (т.е. неявно указывает) на аргумент, используемый при вызове функции. Более того, при вызове функции не нужно применять к аргументу оператор "&". Кроме того, в теле функции ссылочный параметр используется непосредственно, т.е. без оператора "*". Все операции, включающие ссылочный параметр, автоматически выполняются над аргументом, используемым при вызове функции. Наконец, присваивая некоторое значение параметру-ссылке, вы в действительности присваиваете это значение переменной, на которую указывает эта ссылка. Поэтому, применяя ссылку в качестве параметра функции, при вызове функции вы в действительности используете переменную.

В языке С ссылки не поддерживаются. Поэтому единственный способ организации в С вызова по ссылке − использовать указатели.

4. Функции с переменным числом параметров.

Язык С++ допускает использование переменного числа параметров. Признаком функции с переменным числом параметров является многоточие (...) в списке параметров прототипа функции. Например:

```
float f1(int,float,...);
```

Встретив (**...**), компилятор прекращает контроль соответствия типов параметров для такой функции.

Пример 2. Составить программу с использованием функции с переменным числом параметров, которая принимает переменное число аргументов и выводит их значения на печать:

Решение:

```
int _tmain()
{
int a1=10,a2=11,a3=12;
f1(2,a1);       // вызов функции с двумя параметрами
f1(3,a1,a2);      // вызов функции с тремя параметрами
f1(4,a1,a2,a3);  // вызов функции с четырьмя параметрами
getch();
return 0;
}
void f1(int x,...)
{
```

```
int *p=&x;
cout<<"\n";
for(;x!=0;x--)
{
 cout<<*p<<"\t"; p++;
}
}
```

5. Область действия переменных

Правила действия областей видимости (действия) любого языка программирования − это правила, которые позволяют управлять доступом к объекту из различных частей программы. Другими словами, правила действия областей видимости определяют, какой код имеет доступ к той или иной переменной. Эти правила также определяют продолжительность «жизни» переменной. В С++ определено две основные области видимости: *локальные* и *глобальные.*

Локальная область видимости создается блоком и находится внутри фигурных скобок. Переменная, объявленная внутри любого блока кода, называется *локальной* (по отношению к этому блоку), неизвестна за пределами собственного блока, существует только во время выполнения данного блока, разрушается при выходе из него, ее значение при этом теряется. Операторы вне области видимости (действия) переменной не имеют к ней доступа (не видят ее).

Примером являются блоки if, for, switch и т.д.

Если имя переменной, объявленной во внутреннем блоке, совпадает с именем переменной, объявленной во внешнем блоке, то "внутренняя" переменная *скрывает,* или переопределяет, "внешнюю" в пределах области видимости внутреннего блока. Рассмотрим пример.

```
int i, j;
i = 10;
j = 100;
if(j > 0) {
int i; // Эта переменная i отделена от внешней переменной i.
i = j / 2;
cout << "Внутренняя переменная i: " << i << '\n';
 }
cout << "Внешняя переменная i: " << i << '\n';
```

Результат:

```
Внутренняя переменная i: 50
Внешняя переменная i: 10
```

Переменные, объявленные внутри функций, ***локальны*** по отношению к этим функциям. Переменные, объявленные вне функций, включая ***_tmain()*** – ***глобальные***. Из записи фрагмента программы:

```
int global;
int _tmain()
{
int local;
...
}
```

следует, что любые программные операторы могут пользоваться переменной ***global***, а к переменной ***local*** имеют доступ только операторы внутри функции ***main().***

Локальные переменные запоминаются в стеке (стековая, автоматическая память). В начале работы функция выделяет память в стеке для запоминания своих локальных переменных, как показано на рисунке.

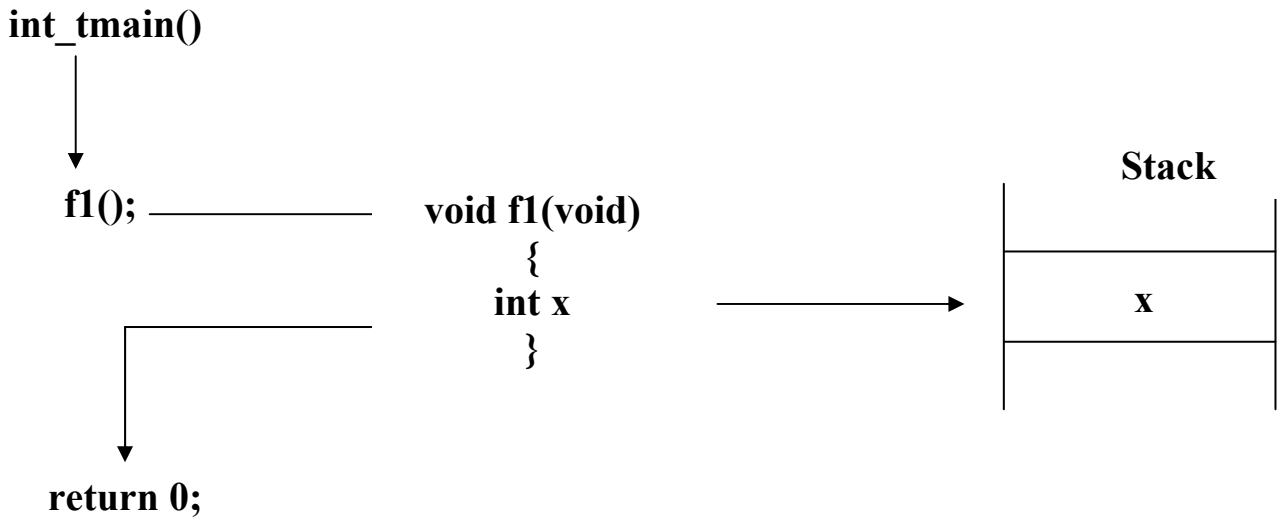


Рисунок 2. Выделение памяти в стеке для локальных переменных

Эта память существует, только пока функция активна. После завершения выполнения функции сама функция очищает выделенную стековую память, отбрасывая все хранящиеся там переменные. Локальные переменные не сохраняют свои значения между вызовами функций, в которых они объявлены.

Локальные переменные также называются *автоматическими* (имеют класс памяти *auto*). Этот термин относят к объектам, которые создаются и разрушаются программой автоматически.

Ключевое слово auto можно использовать для объявления локальных переменных. Но поскольку все локальные переменные являются по умолчанию auto-переменными, то к этому ключевому слову практически никогда не прибегают.

```
auto int n;
```

Поскольку область действия локальной переменной ограничена функцией, в которой она объявлена, две функции бесконфликтно могут объявлять локальные переменные с идентичными именами.

Пример 5. Составить программу с использованием в различных функциях одноименных переменных.

Решение:

```
//прототипы функций
void f1(void);
void f2(void);

//головная функция программы
int _tmain()
{
f1();
  getch();
}
//определение функции f1()
```

```
void f1(void)
{
  char a = 'A';
  cout<<"\n Lokalnaya peremennaya 1: a="<<a;
  f2();
  cout<<"\n Lokalnaya peremennaya 1: a="<<a;
}
//определение функции f2()
void f2(void)
{
  char a = 'B';
  cout<<"\n Lokalnaya peremennaya 2: a="<<a;
}
```

Глобальные переменные существуют в течение всего жизненного цикла программы. Они запоминаются в сегменте данных программы (статическая память) и занимают память независимо от того, нужны они или нет. Могут использоваться в любом месте кода, сохраняя свои значения.

Глобальная область видимости — это декларативная область, которая "заполняет" пространство вне всех функций.

Глобальные переменные объявляются в любом месте программы, но обязательно за пределами функции ***_tmain()***. Обычно их объявления располагают непосредственно перед ***_tmain()***. В следующем фрагменте программы

```
#include <stdio.h>
int glob1;
double glob2;
int _tmain()
{...}
```

объявлены две глобальные переменные ***glob1*** и ***glob2***. Любой оператор в любой функции программы может выполнять чтение и запись значений этих двух переменных.

Пример. Переменная **count** – глобальна.

```
void func1();
void func2();
int count; // Это глобальная переменная.

int _tmain()
{
int i; // Это локальная переменная.
for(i=0; i<10;i++){
// Здесь результат сохраняется в глобальной переменной count.
count = i * 2;
func1();
}
getch();
return 0;
}

void func1()
{
cout <<"count: "<<count; // Доступ к глобальной
// переменной count.
```

```
cout << '\n'; // Вывод символа новой строки.
func2();
}

void func2()
{
int count; // Это локальная переменная.
// Здесь используется локальная переменная count.
for(count=0; count<3; count++)
cout<<'.';
}
```

```
count: 0
...count: 2
...count: 4
...count: 6
...count: 8
...count: 10
...count: 12
...count: 14
...count: 16
...count: 18
...
```

Функция **_tmain()**, так и функция **func1()** используют глобальную переменную **count**. Но в функции **func2()** объявляется локальная переменная **count**. Поэтому здесь при использовании имени count подразумевается именно локальная, а не глобальная переменная **count**. Помните, что, если в функции глобальная и локальная переменные имеют одинаковые имена, то при обращении к этому имени *используется локальная переменная, не оказывая при этом никакого влияния на глобальную*. Это и означает, что локальная переменная скрывает глобальную с таким же именем.

Глобальные переменные инициализируются при запуске программы на выполнение. Если объявление глобальной переменной включает инициализатор, то переменная инициализируется заданным значением. В противном случае (при отсутствии инициализатора) глобальная переменная устанавливается равной нулю.

Хранение глобальных переменных осуществляется в некоторой определенной области памяти, специально выделяемой программой для этих целей. Глобальные переменные полезны в том случае, когда в нескольких функциях программы используются одни и те же данные, или когда переменная должна хранить свое значение на протяжении выполнения всей программы. Однако без особой необходимости следует избегать использования глобальных переменных, и на это есть три причины.

• Они занимают память в течение всего времени выполнения программы, а не только тогда, когда действительно необходимы.

• Использование глобальной переменной в "роли", с которой легко бы "справилась" локальная переменная, делает такую функцию менее универсальной, поскольку она полагается на необходимость определения данных вне этой функции.

• Использование большого количества глобальных переменных может привести к появлению ошибок в работе программы, поскольку при этом возможно проявление неизвестных и нежелательных побочных эффектов. Основная проблема, характерная для разработки больших C++-программ, − случайная модификация значения переменной в каком-то другом месте программы. Чем больше глобальных переменных в программе, тем больше вероятность ошибки.

6. Классы памяти

Существуют следующие классы памяти переменных:

- автоматический (auto),
- внешний (extern),
- статический (static),

– регистровый (register)

Спецификатор mutable применяется только к объектам классов.

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти **auto**, т.е. являются локальными.

Автоматические переменные

Спецификатор **auto** (достался языку C++ от C «по наследству») объявляет локальную переменную. Но он используется довольно редко, поскольку локальные переменные являются «автоматическими» по умолчанию.

```
auto int n;
```

Внешние переменные

В многофайловых проектах необходимо обеспечить видимость глобальных переменных, используемых программой в целом. Однако можно включать только одну копию глобальной переменной в программе и нельзя их повторить в каждом файле (иначе дублирование). Поэтому глобальрные переменные объявляют в одном файле, а в остальных используют **extern-**объявления.

Файл F1	**Файл F2**

int x, y;
char ch;
. . .
void func()
{
x = 123;
}
	extern int x, y;
	extern char ch;
	. . .
	void func22()
	{
	x = y/10;
	}

Ключевое слово **extern** предоставляет компилятору информацию о типе и имени глобальных переменных, повторно не выделяя для них памяти. Во время компоновки этих двух модулей все ссылки на внешние переменные будут определены.

Спецификатор **extern** позволяет также указать, как та или иная функция связывается с программой (т.е. каким образом функция должна быть обработана компоновщиком). По умолчанию функции компонуются как C++ -функции. Но, используя спецификацию компоновки (специальную инструкцию для компилятора), можно обеспечить компоновку функций, написанных на других языках программирования. Общий формат спецификатора компоновки выглядит так:

extern "язык" прототип_функции

Здесь элемент язык означает нужный язык программирования. Например, эта инструкция позволяет скомпоновать функцию myCfunc() как C-функцию

extern "C" void myCfunc();

Статические переменные

Переменные типа **static** хранят свои значения в пределах своей функции или файла.

Если к локальной переменной применен модификатор **static**, то для нее выделяется постоянная область памяти практически так же, как и для глобальной переменной. Это позволяет статической переменной поддерживать ее значение между вызовами функций (значение **static**-переменной не теряется при выходе из функции).

Однако она известна только блоку, в котором объявлена.

Локальные **static** –переменные инициализируются только однажды, в начале выполнения программы, а не при каждом входе в функцию, в которой они объявлены.

```
fun()
{
static int i=5; // статическая переменная
. . .                    // тело функции
}
```

Если модификатор **static** применен к глобальной переменной, то компилятор создаст глобальную переменную, котора будет известна только для файла, в котором она объявлена. Функции в других файлах не могут изменить ее содержимое.

```
static int i; // внешняя статическая
//переменная
main()
{
. . . // тело функции
}
```

static –переменные применяются С-программистами. Для управления доступом к глобальным переменным в С++ рекомендуется использовать пространство имен.

Регистровые переменные

Для компилятора модификатор **register** означает предписание обеспечить такое хранение соответствующей переменной, чтобы доступ к ней можно было получить максимально быстро. Обычно переменная в этом случае будет храниться либо в регистре центрального процессора (ЦП), либо в кэш-памяти (быстродействующей буферной памяти небольшой емкости). Доступ к регистрам ЦП (или кэш-памяти) принципиально быстрее, чем доступ к основной памяти компьютера. Таким образом, переменная, сохраняемая в регистре, будет обслужена гораздо быстрее, чем переменная, сохраняемая, например, в оперативной памяти (ОЗУ), что увеличит скорость выполнения программы.

Если компилятор исчерпает память быстрого доступа, он будет хранить **register**–переменные обычным способом.

Объявлять регистровыми имеет смысл управляющие переменные цикла или переменные, к которым выполняется доступ в теле цикла (можно минимум две переменные).

```
register int i;
register int sum = 0 ;
```

Современные компиляторы автоматически оптимизируют программных код. Поэтому во многих случаях внесение спецификатора **register** в объявление переменной не ускорит выполнение программы, поскольку обработка этой переменной уже оптимизирована. Но в случае функций с большим количеством переменных это по-прежнему полезно.

ФУНКЦИИ

1. **Массивы в качестве аргументов функции**
2. **Перегрузка функций**
3. **Шаблоны функций**
4. **Аргументы, передаваемые функции по умолчанию**
5. **Указатели на функции**
6. **Рекурсия**

ВВЕДЕНИЕ

В лекции рассматриваются особенности использования функций при составлении программ на языке С++.

При разработке программ обычно ***основной алгоритм*** разбивается на более мелкие части – ***подалгоритмы,*** до тех пор пока не получатся относительно простые для решения задачи. Затем эти подалгоритмы перекладываются на соответствующий язык программирования в виде подпрограмм.

1. Массивы в качестве аргументов функции

При вызове функции с аргументом в виде массива ей передается только адрес первого элемента массива, а не полная его копия.

Способы объявить параметр, который принимает указатель на массив:

1) параметр можно объявить как массив, тип и размер которого совпадает с типом и размером массива, используемого при вызове функции.

```
void  show( int  a[10])
```

Несмотря на то что параметр show объявлен здесь как целочисленный массив, состоящий из 10 элементов, С++-компилятор автоматически преобразует его в указатель на целочисленное значение. Необходимость этого преобразования объясняется тем, что никакой параметр в действительности не может принять массив целиком. А так как будет передан один лишь указатель на массив, то функция должна иметь параметр, способный принять этот указатель.

2) представление параметра-массива в виде безразмерного массива.

```
void  show( int  a[ ])
```

Здесь параметр show объявляется как целочисленный массив неизвестного размера. Поскольку С++ не обеспечивает проверку нарушения границ массива, то реальный размер массива — нерелевантный фактор для подобного параметра(но, безусловно, не для программы в целом). Целочисленный массив при таком способе объявления также автоматически преобразуется С++-компилятором в указатель на целочисленное значение.

3) При передаче массива функции ее параметр можно объявить как указатель - этот вариант чаще всего используется профессиональными программистами.

```
void  show( int  *a)
```

Возможность такого объявления параметра объясняется тем, что любой указатель (подобно массиву) можно индексировать с помощью символов квадратных скобок ([ ]). Таким образом, все три способа объявления параметра-массива приводятся к одинаковому результату, который можно выразить одним словом — *указатель.* Т.о. функции передается адрес массива, реальное содержимое которого она может изменить.

Одномерные массивы

Задача 1. Составить программу, в которой рассчитать значения одномерного массива из 10 элементов по формуле $e^i + i + 5$. Для вывода элементов массива на экран использовать функцию `show()`.

```
void show(double *n);
void show(double *n)
```

```
{for(int i=0;i<10;i++)
cout<<n[i]<<"\t";
   }

...
 double a[10];
 for(int i=0;i<10;i++)
a[i]=exp((double)i)+i+5;
show(a);
```

Задача 2. Составить программу, в которой необходимо ввести с клавиатуры значения элементов массива целых чисел b из 10 элементов, возвести все элементы в квадрат, вывести на экран массив до изменения и после изменения. Все действия выполнить с помощью функций.

```
void show(int *n,int k)
{for(int i=0;i<k;i++)
cout<<n[i]<<"\t";}

void vvd(int *n,int k)
{for(int i=0;i<k;i++)
cin>>n[i];}

void kv(int *n,int k)
{for(int i=0;i<k;i++)
n[i]=n[i]*n[i];}

...
int a[10];
 vvd(a,10);
show(a,10);
kv(a,10);
show(a,10);
```

Многомерные массивы

При передаче массивов в качестве параметров через заголовок функции следует учитывать, что передавать массивы можно только с одной неопределенной границей мерности (самой левой).

Задача 3. Составить программу, в которой необходимо ввести с клавиатуры значения элементов массива целых чисел b [3][3], вывести в виде матрицы, определить сумму всех его элементов. Все действия выполнить с помощью функций.

```
void show(int n[][3])
{
for(int i=0;i<3;i++)
{for(int j=0;j<3;j++)
cout<<n[i][j]<<"\t";
cout<<"\n";}
}

void vvd(int n[][3])
{for(int i=0;i<3;i++)
for(int j=0;j<3;j++)
cin>>n[i][j];}


int sum(int n[][3])
{int s=0;
for(int i=0;i<3;i++)
```

```
for(int j=0;j<3;j++)
s=s+n[i][j];
return s;}

...
int a[3][3];
vvd(a);
show(a);
cout<<"\nsum="<<sum(a);
...
```

Динамические массивы

Задача 4. Динамические массивы как параметры функций.

Требуется составить программу с использованием функции, которая заполняет элементы матрицы размером *mxn* последовательно значениями целых чисел: 0,1, 2, 3.....

Для формирования матрицы в основной программе использованы динамические массивы, так как размеры матрицы заранее не известны.

```
void f(int m,int n, int **a)
{
int k=0;
for(int i=0;i<m;i++)
for(int j=0;j<n;j++)
*(*(a+i)+j)=k++;
}
...
int **b,m1,n1;
cout<<"strok:";
cin>>m1;
cout<<"stolbzov:";
cin>>n1;
int i,j;
b=new int* [m1];
for(i=0;i<m1;i++)
      b[i]=new int[n1];
f(m1,n1,b);
for(i=0;i<m1;i++)
      {
      cout<<"\n";
      for(j=0;j<n1;j++)
      cout<<b[i][j]<<"\t";
      }
for(i=0;i<m1;i++)
delete b[i];
delete b;
```

2. Перегрузка функций

Перегрузкой функций (function overloading), называют наличие нескольких функций с одинаковыми именами и различными параметрами. Это один из способов реализации полиморфизма.

Чтобы перегрузить функцию, достаточно объявить различные ее версии с отличающимися типами и/или числом параметров в одной области видимости. При этом поддерживается принцип полиморфизма "один интерфейс − множество методов".

В данном случае мы имеем два экземпляра перегружаемой функции *rez ()*.

Пример

```
float rez(float a, float b);
int rez(int a, int b);
```

```
float rez(float a, float b)
{return (a-b)/100;}
int rez(int a, int b)
{return (a+b)*100;}
...
int a=16, b=9;
float c=3.5,d=1.2;

float y=rez(c, d); //0.023
int x=rez(a, b);      //2500
      cout<<y<<"   "<<x;
```

3. Шаблон функций

Подготовку перегружаемых функций помогают автоматизировать шаблоны. Шаблон семейства функций определяется один раз, но это определение параметризуется. Для этого используется список параметров шаблона (в качестве параметра на этапе компиляции передается конкретный тип данных).

Шаблон семейства функций состоит из двух частей:

- заголовок шаблона (template <список параметров шаблона>)
- обычного определения функции (заголовок и тело функции), в котором тип возвращаемого значения и/или типы параметров обозначаются именами параметров шаблона, введенных в его заголовке.

Имена параметров шаблона могут использоваться и в теле определения функции для обозначения типов локальных объектов.

Формат простейшей функции-шаблона:

```
template <class type>
заголовок функции
{ тело функции
}
```

где вместо слова type может использоваться произвольное имя.

Пример.

```
template <class T>
T sum(T a,T b){
return a+b;
}
...
cout<<char(sum(30,35));
```

4. Аргументы, передаваемые функции по умолчанию

Аргументы, передаваемые функции *по умолчанию*, используются автоматически, если при вызове функции не задан аргумент. Позволяют упростить обращение к сложным функциям, а также применяются в качестве "сокращенной формы" перегрузки функций.

```
void sum(int a=0, int b=1)
{
cout<<"sum="<<a+b<<"\n";
return;
}

int _tmain()
{
 int a=16, b=9;

 sum();    //1
```

```
sum(5);   //6
sum(5,6); //11
```

Значения аргументов функций, передаваемых по умолчанию, должны быть заданы только однажды и при первом объявлении функции в файле.

При попытке определить новые (или даже те же) передаваемые по умолчанию значения аргументов в определении функции компилятор отобразит сообщение об ошибке и не скомпилирует вашу программу.

Для каждой версии перегруженной функции для передачи по умолчанию можно задавать различные аргументы. Таким образом, разные версии перегруженной функции могут иметь различные значения аргументов, действующие по умолчанию.

Пример "сокращенной формы" перегрузки функций

функция для сложения двух или трех чисел

```
void sum(int a, int b, int c=0);
void sum(int a, int b, int c)
{
cout<<"sum="<<a+b+c<<"\n";
return;
}


 sum(5,6);   //11
 sum(5,6,2); //13
```

вместо

```
void sum(int a, int b, int c);
void sum(int a, int b, int c)
{
cout<<"sum="<<a+b+c<<"\n";
return;
}

void sum(int a, int b);
void sum(int a, int b)
{
cout<<"sum="<<a+b<<"\n";
return;
}
```

5. Указатели на функции

В языке C++ функция не может быть значением переменной, в то же время, аналогично переменным функция физически расположена в памяти и соответственно имеет адрес. Таким образом, ее адрес, присвоенный указателю, является точкой входа в функцию. Указатель на функцию может быть использован для вызова функции вместо ее имени, а также позволяет передавать функцию как обычный параметр в другую функцию. Следовательно с указателем на функцию можно обращаться, как с переменной (передавать его другим функциям, помещать в массивы и т.д).

Указатель на функцию – это такой тип переменной, которой можно присваивать адрес точки входа в функцию, то есть адрес первой исполняемой команды. Эта переменная в дальнейшем может использоваться для вызова функции вместо ее имени. Определение указателя на функцию имеет следующий общий вид:

```
тип_результата (*имя_указателя_на функцию)(список_типов_параметров);
```

В объявлении

```
int (*fun)(int, int *);
```

переменная `fun` является указателем на функцию с двумя параметрами:

типа `int` и указателем на `int`. Сама функция должна возвращать значение типа `int`. Круглые скобки, содержащие имя указателя `fun` и признак указателя *****, обязательны, иначе запись

```
int *fun (int, int *);
```

будет интерпретироваться как объявление функции `fun` возвращающей указатель на `int`. Это следует из того, что приоритет префиксного оператора *ниже, чем приоритет ().

Вызов функции возможен только после инициализации значения указателя `fun` и имеет вид:

```
(*fun)(i,&j);
```

В этом выражении для получения адреса функции, на которую ссылается указатель `fun` используется операция разадресации * . Вызов функции через указатель возможен так же в обычным способом:

```
fun( i,&j);
```

Пример . Использование указателя на функцию в качестве параметра функции

```
Пример. Функция myoperator() способна выполнить любую
операцию с двумя целыми числами, – эта операция задается третьим
параметром (см. также Прата С. Язык программирования С++. ).

int plus(int,int);
int minus (int,int);
int myoperator (int x, int y, int (*f)(int,int));

int main ()
{
int m,n;
m = myoperator(7, 5, plus);
cout<<"m= "<<m;
n = myoperator(20, m, minus);
cout<<" n= "<<n<<endl;
return 0;
}

int plus(int a, int b){
return (a+b);
}

int minus (int a, int b){
return (a-b);
}

int myoperator (int x, int y, int (*f)(int,int))
{
int g;
g = (*f)(x,y);
return (g);
}
```

5. Рекурсия

Рекурсивным называется объект, который частично определяется через самого себя.

Вызов рекурсивной процедуры должен выполняться по условию, которое на каком-то уровне рекурсии станет ложным.

Если условие истинно, то цепочка рекурсивных вызовов продолжается. Когда оно становится ложным, то рекурсивный спуск заканчивается и начинается поочередный рекурсивный возврат из всех вызванных на данный момент копий рекурсивной процедуры.

Число рекурсивных вызовов в каждый конкретный момент времени, называется **текущим уровнем рекурсии.**

Максимальное число рекурсивных вызовов процедуры без возвратов, которое происходит во время выполнения программы, называется **глубиной рекурсии.** Для каждой копии рекурсивной процедуры необходимо выделял дополнительную область памяти.

Рекурсия, как правило, применяется в тех случаях, когда основную задачу можно разбить на подзадачи той же структуры, что и первоначальная задача.

Основным достоинством рекурсивных программ по сравнению с нерекурсивными является наглядность и компактность. Однако некоторые рекурсивные программы выполняются дольше чем итерационные и требуют значительных затрат оперативной памяти.

Это связано с тем, что каждый раз при вызове подпрограммы выделяется новый блок памяти и начинает работать, по сути, новая подпрограмма. Необходимо помнить, что если в рекурсивной подпрограмме существуют локальные данные (константы, типы данных, переменные, подпрограммы), то при каждом следующем вызове такой подпрограммы порождается новый набор локальных данных, значения которых отличаются от данных с теми же именами на предыдущем шаге рекурсии. Если рекурсивная подпрограмма содержит правила изменения глобальных данных, то при каждом вызове подпрограммы произойдет новое уточнение этих данных.

Каждый набор локальных данных и текущие значения фактических параметров помещаются в программный стек в виде экземпляра памяти.

Программным стеком называется определенным образом организованная область оперативной памяти, в которую помещаются локальные данные и фактические параметры на время выполнения рекурсивной подпрограммы и в которой доступен экземпляр памяти, созданный последним. Из-за многократного выделения памяти под локальные данные рекурсивной подпрограммы может произойти переполнение программного стека. Следовательно, особое внимание надо уделять подбору оптимального количества локальных данных, выбору типа соответствующего параметра или переменной и способа передачи данных в подпрограмму. По умолчанию размер стека равен 16 Кбайт. С помощью директивы компилятора *$M* его можно изменить [9, 10].

Особое внимание необходимо уделить корректному выходу из рекурсивной подпрограммы. Рекурсивный вызов подпрограммы должен управляться некоторым условием, которое в определенный момент перестает выполняться. Пока условие истинно, рекурсия повторяется (осуществляется *рекурсивный спуск*), но как только условие становится ложным, начинается последовательный *рекурсивный возврат* из всех созданных копий подпрограммы.

В общем случае любая рекурсивная процедура включает в себя некоторое множество операторов S и один или несколько операторов рекурсивного вызова.

Структура рекурсивной процедуры может принимать три разных формы:

1. Форма с выполнением действий до рекурсивного вызова *(с выполнением действий на рекурсивном спуске).*

```
void Rec()
{
      … S … ;
      if (условие) Rec ();
      return;
 }
```

2. Форма с выполнением действий после рекурсивного вызова *(с выполнением действий на рекурсивном возврате).*

```
void Rec()
{
      if условие Rec ();
      … S … ;
      return;
}
```

3. Форма с выполнением действий как до, так и после рекурсивного вызова *(с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате).*

```
void Rec ()
 {
      … S1 …;
      if (условие) Rec();
      … S2 … ;
      return;
   };
```

Пример. Вычисление факториала числа N!

без рекурсии

```
long  Iter_Fact (int n);  //объявление функции (прототип)
//  идентификатор формального параметра можно опустить
int _tmain()
{
int i;
long Fact1, Fact2, Fact3;

Fact1= Iter_Fact (10);        //вызов функции  можно так
i = 10;
Fact2 = Iter_Fact (i);       //вызов функции  или так
Fact3 = Iter_Fact (5 + 5);   //вызов функции  или так

cout << Fact1<<"  "<< Fact2 <<"  "<< Fact3 << endl;
getch();
}
//определение функции
long  Iter_Fact (int n)
{
long  f = 1;
for (int i = 2; i <= n;  i++)
              f = f * i;
return (f);               // возвращаем результат
}
```

Рекурсивное определение факториала

$$n! = \begin{cases} 1 & n = 0, \\ n \cdot (n-1)! & n > 0. \end{cases}$$

```
//вычисление на рекурсивном возврате
double  Rec_Fact_Up (int);    // прототип функции
double  Rec_Fact_Dn (double, int, int); // прототип функции

void main()
{     int n = 5;
double Fact;
Fact = Rec_Fact_Up (n);      // вычисление факториала
cout << n << " != " <<  Fact << endl;

                                        // вычисление факториала
Fact = Rec_Fact_Dn (1.0, 1, n);
cout <<n<< " ! = " <<  Fact << endl;
getch();

}

double  Rec_Fact_Up (int n)
{     if (n <= 1)
   return 1.0;
else
   return  Rec_Fact_Up(n-1) * n;
```

```
//вычисление на рекурсивном возврате
// Mult=Rec_Fact_Up(n-1) -  рекурсивный вызов;
}     // Mult *n – оператор накопления факториала

double  Rec_Fact_Dn(double Mult, int i, int m)
{
Mult = Mult * i;
if (i == m)
  return Mult;
else
  return  Rec_Fact_Dn (Mult, i+1, m);
      //вычисление на рекурсивном спуске
// Rec_Fact_Dn() -  рекурсивный вызов;
// Mult=Mult *i – оператор накопления факториала
}
```

Пример. Возвести число в степень.

```
double degree(double x,  int n);
double degree(double x,  int n)
{
  if  (n= =0)      return 1;

 else   return x*degree(x, n-1);
   // рекурсивный вызов функции
  //действия выполняются на рекурсивном возврате
}

int _tmain()
{

double x, y;
  int  n;
  cout << " X= ";
  cin >> x;
  cout<<" N = ";
  cin >> n ;
  if (n >= 0)
  {
y = degree(x, n);
cout << x <<" v stepeni  " << n <<" = "<< y << endl;
  }
getch();

}
```

Пример. Вывод массива на экран.

```
printM(int *b,int n){
int i=0;
cout<<b[i]<<" ";
 if (i<n-1) {
   printM(b+1,n-1) ;
 }
}
int _tmain()
{
int a[7]={1,5,7,2,4,1,9};
 printM(a,7);
. . .}
```

Пример. Определить наибольший общий делитель (по алгоритму Евклида)

```
int  NOD_rec  (int,  int);

int _tmain ()
```

```
{ int m, n;

  cin>>m;
  cin>>n;

  cout << NOD_rec  ( m, n) << endl ;

  getch();
  return 0 ;
}

// рекурсивная функция
int  NOD_rec (int m,  int n)
{
   if (!(m % n))
   return n;
   else
       return NOD_rec(n, m %n);
// рекурсивный вызов функции
//действия выполняются на рекурсивном спуске
}
```

Пример. Рекурсивная функция вычисления чисел Фибоначчи.

```
unsigned long fib (unsigned long);

int main ()
{ unsigned long number;
  cout << " Input number";
  cin >> number;
  cout << number<<" chislo Fib = "<< fib(number) << endl;
  getch();
 return 0 ;
}

                              // рекурсивная функция
unsigned long fib (unsigned long n)
{
if (n == 0 || n == 1)
   return n;
else
   return  fib(n - 1) + fib(n - 2);
        // рекурсивный вызов функции
       //действия выполняются на рекурсивном возврате
}
```

Пример. Рекурсивная функция вычисления суммы элементов числовой последовательности

```
double sum (int [],  int);

int main ()
{
  const int nn =100;
  int array[nn], n;
  cout<<" n= ";
```

```
  cin>>n;

  for (int i = 0; i < n; i++)
  {   array[i] = rand() % 20;
  cout << " " << array[i];
  }
cout <<endl;
cout << sum (array, n) << endl ;
  getch();
return 0 ;
}
                                          // рекурсивная функция
double sum (int s[], int n)
{
if (n == 1)
  return s[0];
else
  return sum(s, n-1) + s[n-1];
         // рекурсивный вызов функции
        //действия выполняются на рекурсивном возврате
}
```

Пример. Классика рекурсии - "Ханойские башни".

Есть три стержня A, B, и C. На стержень A надето N дисков, наверху самый маленький, каждый следующий диск больше предыдущего, а внизу самый большой. На другие стержни дисков не надето.

Необходимо перенести диски со стержня A на стержень C, пользуясь стержнем B, как вспомогательным, так, чтобы диски на стержне C располагались в том же порядке, в каком они располагаются на диске A перед перемещением.

При перемещении никогда нельзя класть больший диск на меньший.

История создания головоломки

Эту известную игру придумал французский математик Эдуард Люка, в 1883 году её продавали как забавную игрушку.

Первоначально она называлась «Профессор Клаус (Claus) из Коллеж Ли-Су-Стьян (Li-Sou-Stian)» но обнаружилось, что таинственный профессор из несуществующего колледжа — это анаг-рамма фамилии изобретателя игры — профессора Люка (Lucas) из коллежа Сен-Луи (Saint Louis).

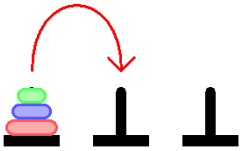
Рекурсивный метод

Для того, чтобы переложить всю пирамиду, надо сначала переложить все, что выше самого большого диска, с первого на вспомогательный стержень, потом переложить это самое большой диск с первого на третий стержень, а потом переложить оставшуюся пирамиду со второго на третий стержень, пользуясь первым стержнем, как вспомогательным.

Всего получается 2^N-1 перекладываний.

```
void Move(int n, char x,char y,char z) {
   if (n==1)   cout<<x<<" -> "<<y<<endl;
else
   if (n>1) {
Move(n-1,x,z,y);
cout<<x<<" -> "<<y<<endl;
Move(n-1,z,y,x);
}
}
. . .
Move(3,'A','B','C');
. . .
**********
```

```
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```



Легенда гласит, что, в Великом храме города Бенарас, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времен монахи этого монастыря провинились перед богом Брамой. Разгневанный, Бог Брама поместил на один из стержней 64 диска из чистого золота, причем так, что каждый меньший диск лежит на большем.

Монахи должны были переместить все диски с одного стержня на другой при условии, что при каждом перемещении можно брать только один диск и больший диск никогда нельзя положить на меньший. Третий стержень предоставляет возможность для временного размещения дисков. Как только все 64 диска будут переложены со стержня, на который Бог Брама сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.

Напишите программу решения этой задачи.
Перемещение N дисков может быть легко представлено в терминах перемещения только N-1 диска (и, следовательно, рекурсивно):
1. Переместить N-1 дисков с колышка 1 на колышек 2, используя колышек 3 как место временного размещения.
2. Переместить последний диск (наибольший) с колышка 1 на колышек 3
3. Переместить N-1 дисков с колышка 2 на колышек 3, используя колышек 1 как место временного размещения.
Используйте рекурсивную функцию с четырьмя параметрами:
1. Количество дисков, которое должно быть перемещено.
2. Колышек, на который эти диски нанизаны первоначально.
3. Колышек, на который эта группа дисков должна быть перемещена.
4. Колышек, используемый как место временного размещения.
Ваша программа должна печатать четкие инструкции, что нужно делать для перемещения дисков с начального колышка на конечный. Например, чтобы передвинуть группу из трех дисков с колышка 1 на колышек 3, ваша программа должна напечатать следующую последовательность перемещений:
1->3 (это обозначает перенос диска с 1-го колышка на 3-ий)
1->2
3->2
1->3
2->1
2->3
1->3

СОРТИРОВКА

1. **Сортировка вставкой (простыми включениями)**
2. **Сортировка выбором (выделением)**
3. **Сортировка обменом (метод «пузырька»)**
4. **Сортировка Шелла;**
5. **Шейкер сортировка**
6. **Быстрая сортировка (QuickSort).**

Элементы сортировки присутствуют почти во всех задачах. Упорядоченные объекты содержатся в телефонных книгах, в ведомостях подоходных налогов, в оглавлениях, в библиотеках, в словарях, на складах, да и почти всюду, где их нужно разыскивать.

Сортировка служит хорошим примером того, что одна и та же цель может достигаться с помощью различных алгоритмов, причем каждый из них имеет свои определенные преимущества и недостатки, которые нужно оценить с точки зрения конкретной ситуации. Многие из алгоритмов в некотором смысле являются оптимальными. Поэтому на примере сортировки мы убеждаемся в необходимости сравнительного анализа алгоритмов. Кроме того, здесь мы увидим, как при помощи усложнения алгоритмов можно добиться значительного увеличения эффективности по сравнению с более простыми и очевидными методами.

Сортировка - процесс перестановки заданного множества нумерованных объектов в определенном порядке. Цель сортировки - облегчить последующий поиск элементов в отсортированном множестве.

Например, последовательность A из n элементов отсортирована в порядке возрастения (неубывания), если $\mathbf{a_1 \le a_2 \le a_n}$.

Например, даны элементы:

$a_1, a_2, a_3, \ldots a_n$;

Сортировка означает перестановку этих элементов в таком порядке:

$a_{k1}, a_{k2}, a_{k3}, \ldots a_{kn}$;

что при заданной функции упорядочения f справедливо отношение:

$f(a_{k1}) \le f(a_{k2}) \le f(a_{k3}) \le \ldots \le f(a_{kn})$;

Обычно функция упорядочения не вычисляется по какому-то специальному правилу, а содержится в каждом элементе в виде явной компоненты (поля). Ее значение называется ключом элемента. Для представления элемента a, особенно хорошо подходит представление данных – структура (массив). Для простоты будем считать функцией упорядочения тождественную функцию, т.е. ключом для сортировки будут являться сами элементы, а порядок сортировки – по возрастанию.

Тип ключа (a в данном случае тип элемента) может быть произвольным, но обязательно у такого типа должно быть задано отношение всеобщего порядка.

Различают две категории методов сортировки:

- сортировка массивов
- сортировка файлов

Сортировку массивов называют внутренней, т.к. все элементы массивов хранятся в быстрой внутренней памяти машины с прямым доступом в произвольном порядке, а сортировку файлов – внешней, т.к. их элементы хранятся в медленной, но большей по объему внешней памяти. Внешняя сортировка выполняется в строго определенной последовательности.

Оценка алгоритма сортировки включает определение

- средней скорости сортировки
- скорости для лучшего и для худшего случая
- изменения скорости сортировки с увеличением упорядоченности массива
- наличия перестановки элементов с одинаковыми значениями.

Скорость сортировки зависит от числа сравнений (C) и необходимых операций обмена (M). Данные параметры являются функциями от числа N сортируемых элементов. Операции обмена занимают большее время, чем операции сравнения. Важными критериями являются требуемый объем памяти и сложность алгоритмов сортировки.

Таким образом, выбирая метод сортировки, руководствуясь критерием экономии памяти, классификацию алгоритмов проводят в соответствии с их эффективностью, т. е. экономией времени или быстродействием.

Самым простым было бы использовать вспомогательный массив, куда последовательно помещать элементы, начиная с минимального по значению. Однако в этом случае требуется двойное количество памяти.

Таким образом, принципы построения алгоритмов сортировки массивов:

- сортировка на месте
- произвольный доступ к элементам

Ниже будут рассматриваться методы сортировки данных в оперативной памяти без привлечения вспомогательного массива.

Выделяют базовые и улучшенные методы сортировки.

Базовые (простые) методы особенно хорошо подходят для разъяснения свойств большинства принципов сортировки. Программы, основанные на этих методах, легки для понимания и коротки. Хотя сложные методы требуют меньшего числа операций, эти операции более сложные, поэтому, при достаточно малых n простые методы работают быстрее, но их не следует использовать при больших n.

Прямые методы сортировки массивов можно разбить на три основных класса в зависимости от лежащего в их основе приема:

сортировка вставкой (простыми включениями)
сортировка выбором (выделением)
сортировка обменом (метод «пузырька»)

Улучшенные (сложные) методы сортировки базируются на тех же принципах, что и прямые, но используют различные варианты ускорения процесса сортировки:

сортировка Шелла;
Шейкер сортировка
быстрая сортировка (QuickSort).

Практически каждый алгоритм сортировки можно разбить на три части:
- сравнение, определяющее упорядоченность пары элементов;
- перестановку, меняющую местами неупорядоченную пару элементов;
- сортирующий алгоритм, сравнивающий и переставляющий элементы, пока они не упорядочатся.

Тестирование различных методов сортировки полезно проводить на массивах с различным способом заполнения, а именно: на случайно заполненных, изначально отсортированных, отсортированных в обратном порядке. Для этого реализуются соответствующие функции заполнения.

Пусть все программы, для тестирования тестировок используют динамические массивы целых чисел и сортировку по возрастанию.

```
void f(int *a,int n){
randomize();
int k=0;
for(int i=0;i<n;i++)
*(a+i)=random(100);
```

```
}

 void show(int *a,int n){
for(int i=0;i<n;i++)
cout<<a[i]<<"\t";
cout<<"\n";
}

int _tmain(){
int *b,n1;
cout<<"strok:";
cin>>n1;
int i,j;
b=new int [n1];
f(b,n1);
show(b,n1);
. . .
delete b;
. . .
}
```

Сортировка вставкой (простыми включениями)

Рассмотрим на примере восьми случайно взятых чисел. Элементы условно разделяются на готовую (отсортированную) последовательность *{a1 ... ai-1}* и входную последовательность *{ai ... an}*. На каждом шаге, начиная с *i = 2* и увеличивая *i* на единицу, берут *i*-й элемент входной последовательности и передают в готовую последовательность, вставляя его на подходящее место.

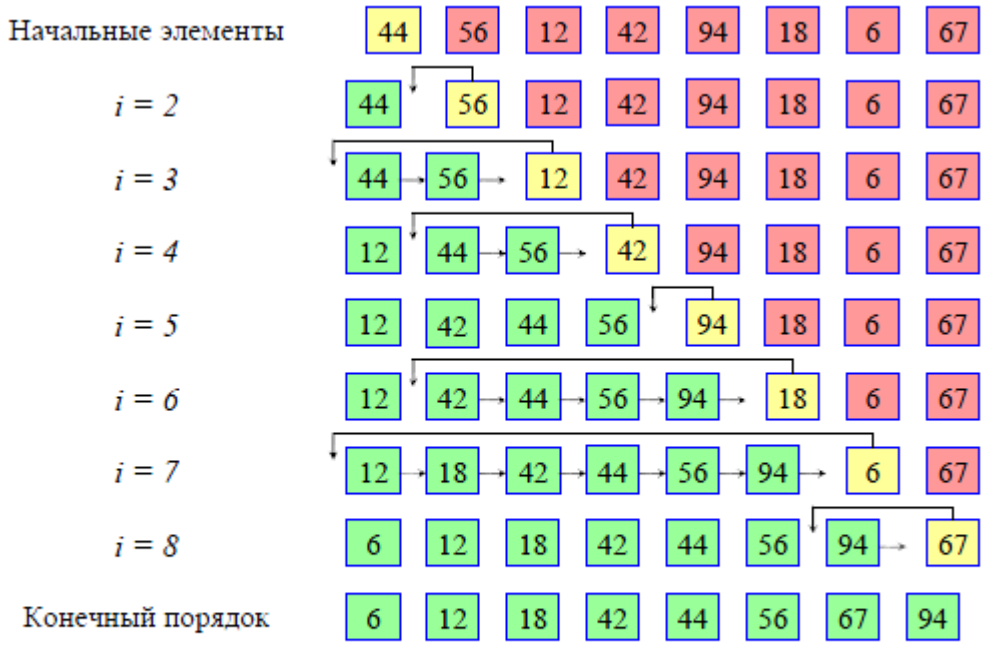


Рис. 1. Пример сортировки простыми включениями: – отсортированная часть, – входная часть, - текущий элемент.

Алгоритм сортировки массива простыми включениями на псевдокоде выглядит следующим образом

Алгоритм сортировки вставкой массива [0…n]:

1. Для всех i от 1 до n выполнить:

1. 1. Взять очередной i-й неотсортированный элемент и сохранить его в рабочей переменной x
1. 2. Найти позицию j в отсортированной (0…i-1) части массива, в которой присутствие взятого элемента не нарушит упорядоченности элементов
1.3. Сдвиг элементов массива от i-1 до j-1 вправо, чтобы освободить найденную позицию вставки
1.4. Вставка взятого элемента в найденную j-ю позицию.

Данный алгоритм можно оптимизировать, скомбинировав шаги 1.2 и 1.3, т.е. при поиске подходящего места вставки чередовать сравнения и пересылки как бы «просеивая» *x*, сравнивая его с очередным элементом *a*, и затем либо вставляя *x*, либо пересылая *aj* направо и продвигаясь налево. Просеивание может закончиться при двух различных условиях:

1. Найден элемент *a* с ключом меньшим, чем ключ *x*.
2. Достигнут левый конец готовой последовательности.

Простейшая программа сортировки вставкой:

```
 void InsertSort (int *a,int size) {
//последовательный перебор не отсортированных эл-тов массива
for (int i=1;i< size;i++) {
int x = a[i]; //взятие очередного элемента
int j = i-1;
while (x<a[j]) { // повторять пока место вставки не найдено
// сдвиг текущего j- го элемента на 1 позицию вправо
a[j+1] = a[j];
j--;
if (j<0) break; //условие выхода при достижении левой границы
}
a[j+1] = x; // вставка взятого элемента
}
}
            . . . InsertSort(b,n1);
```

Анализ сортировки простыми включениями. Число C_i сравнений ключей при i-м просеивании составляет самое большее $i - 1$, самое меньшее 1 и, если предположить, что все перестановки n ключей равновероятны, в среднем равно $i/2$. Число M_i пересылок (присваиваний) равно $C_i + 1$. Общее число сравнений и пересылок есть:

$$C_{min} = n-1; \qquad M_{min} = 2(n-1);$$

$$C_{cp} = \frac{1}{4}(n^2+n-2); \qquad M_{cp} = \frac{1}{4}(n^2+9n-10);$$

$$C_{max} = \frac{1}{2}(n^2+n)-1; \qquad M_{max} = \frac{1}{2}(n^2+3n-4);$$

Наименьшие числа появляются, если элементы с самого начала упорядочены, а наихудший случай встречается, если элементы расположены в обратном порядке. В этом смысле сортировка включениями демонстрирует вполне естественное поведение. Ясно также, что данный алгоритм описывает устойчивую сортировку: он оставляет неизменным порядок элементов с одинаковыми ключами.

Алгоритм сортировки простыми включениями легко можно улучшить, пользуясь тем, что готовая последовательность *a1, ..., ai-1* в которую нужно включить новый эле-

мент, уже упорядочена. Место включения можно найти значительно быстрее, применив **бинарный или двоичный поиск**.

К сожалению, улучшение, которое получается, используя метод бинарного поиска, касается только числа сравнений, а не числа необходимых пересылок. В действительности, поскольку пересылка элементов, т. е. ключей и сопутствующей информации, обычно требует значительно больше времени, чем сравнение двух ключей, то это улучшение ни в коей мере не является решающим: важный показатель *M* по-прежнему остается порядка *n2*. И в самом деле, пересортировка уже рассортированного массива занимает больше времени, чем: при сортировке простыми включениями с последовательным поиском! Этот пример показывает, что «очевидное улучшение» часто оказывается намного менее существенным, чем кажется вначале, и в некоторых случаях (которые действительно встречаются) может на самом деле оказаться ухудшением. В конечном счете, сортировка включениями оказывается не очень подходящим методом для вычислительных машин: включение элемента с последующим сдвигом всего ряда элементов на одну позицию неэкономна. Лучших результатов можно ожидать от метода, при котором пересылки элементов выполняются только для отдельных элементов и на большие расстояния. Эта мысль приводит к сортировке выбором.

Сортировка Шелла

Некоторое усовершенствование сортировки простыми включениями было предложено Д. Л. Шеллом в 1959 г.

Основная идея алгоритма состоит в том, что на начальном этапе реализуется сравнивание и, если требуется, перемещение далеко отстоящих друг от друга элементов. Интервал между сравниваемыми элементами (*step*) постепенно уменьшается до единицы, что приводит к перестановке соседних элементов на последних стадиях сортировки (если это необходимо).

Реализуем метод Шелла следующим образом. Начальный шаг сортировки примем равным n/2, т.е. 1/2 от общей длины массива, и после каждого прохода будем уменьшать его в два раза. Каждый этап сортировки включает в себя проход всего массива и сравнение отстоящих на *step* элементов. Проход с тем же шагом повторяется, если элементы переставлялись. Заметим, что после каждого этапа отстоящие на *step* элементы отсортированы.

Псевдокод

Шаг 0. $i = t$.

Шаг 1. Разобьем массив на списки элементов, отстающих друг от друга на h_i. Таких списков будет h_i.

Шаг 2. Отсортируем элементы каждого списка сортировкой вставками.

Шаг 3. Объединим списки обратно в массив. Уменьшим i. Если iнеотрицательно — вернемся к шагу 1

```
void ShellSort(int *ms, int k)  {
int i, j, n;
int gap; // шаг сортировки
int flg; // флаг окончания этапа сортировки
```

```
for(gap = k/2; gap > 0; gap /= 2)
do
{ flg = 0;
for(i = 0, j = gap; j < k; i++, j++)
if(*(ms+i) > *(ms+j)) // сравниваем отстоящие на gap эл-ты
{ n = *(ms+j);
*(ms+j) = *(ms+i);
*(ms+i) = n;
flg = 1; // есть еще не рассортированные данные
}
} while (flg); // окончание этапа сортировки
}
```

Рассмотрим пример. Рассортировать массив чисел: 41,53,11,37,79,19,7,61.

В строке после массива в круглых скобках указаны индексы сравниваемых элементов и указан номер внешнего цикла.

41 53 11 37 79 19 7 61 – исходный массив.
41 19 11 37 79 53 7 61 – (0,4), (1,5) 1-ый цикл
41 19 7 37 79 53 11 61 – (2,6), (3,7)
7 19 41 37 ~~11~~ ~~53~~ ~~79~~ 61 – (0,2),(1,3),(2,4),(~~3,5~~),(~~4,6~~),(5,7) 2-ой цикл
7 19 11 37 41 53 79 61 – (0,2),(1,3),(2,4),(3,5),(4,6),(5,7)
7 11 19 37 41 53 61 79 – сравнивались соседние.(3-ий цикл).

На первом проходе отдельно группируются и сортируются все элементы, отстоящие друг от друга на четыре позиции. Этот процесс называется 4-сортировкой. После этого элементы вновь объединяются в группы с элементами, отстоящими друг от друга на две позиции, и сортируются заново. Этот процесс называется 2-сортировкой. Наконец, на третьем проходе все элементы сортируются обычной сортировкой, или 1-сортировкой.

Сначала может показаться, что необходимость нескольких проходов сортировки, в каждом из которых участвуют все элементы, больше работы потребует, чем сэкономит. Однако на каждом шаге сортировки либо участвует сравнительно мало элементов, либо они уже довольно хорошо упорядочены и требуют относительно мало перестановок.

Очевидно, что этот метод в результате дает упорядоченный массив, и также совершенно ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая *i*-сортировка объединяет две группы, рассортированные предыдущей *2i*-сортировкой. Приемлема любая последовательность приращений, лишь бы последнее было равно 1, так как в худшем случае вся работа будет выполняться на последнем проходе. Однако менее очевидно, что метод убывающего приращения дает даже лучшие результаты, когда приращения не являются степенями двойки.

Худшее время	зависит от выбранных шагов
Лучшее время	O(n*k) сравнений, O(k) обменов,

	где k - количество шагов
Среднее время	зависит от выбранных шагов
Затраты памяти	O(n) всего, O(1) дополнительно

- первоначально используемая Шеллом последовательность длин промежутков:

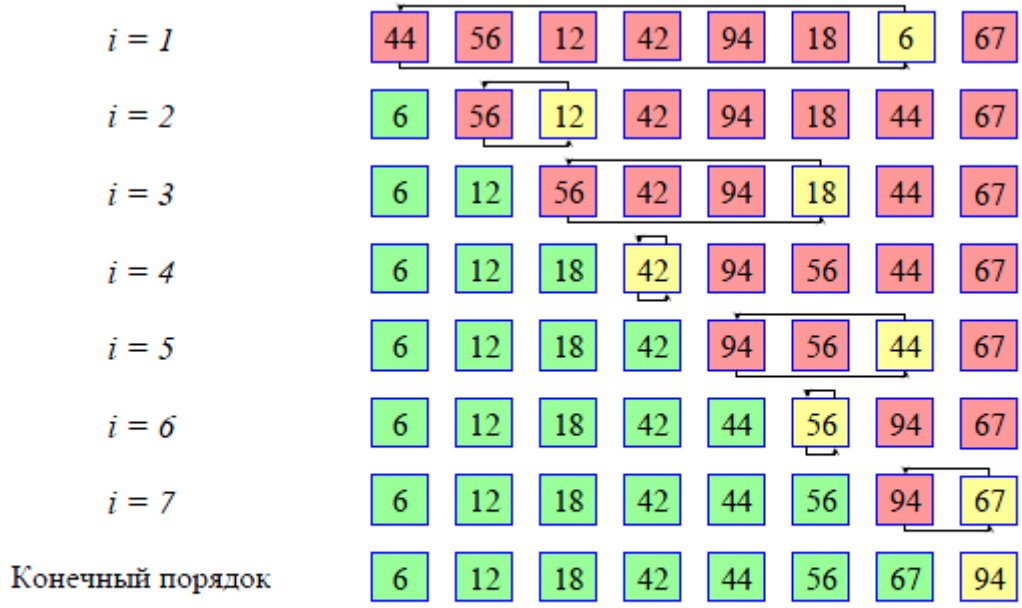
  в худшем случае, сложность алгоритма $O(N^2)$;

- предложенная Хиббардом последовательность: все значения $2^i - 1 \leq N, i \in \mathbb{N}$;

  сложность алгоритма $O(N^{3/2})$;

Сортировка выбором

Из всего массива *[0..n-1]* выбирается элемент с наименьшим ключом (в нашем случае минимальный). Он меняется местами с первым элементом массива *a[0]*. Далее выбирается минимальный элемент из подмассива *[1..n-1]* и меняется местами с *a[1]*, далее минимальный из *[2..n-1]* и меняется местами с *a[2]*, и т.д. Эти операции повторяются до тех пор, пока не останется только один элемент – наибольший.



Пример сортировки выбором: – отсортированная часть, – входная часть, – текущий минимум входной (не отсортированной) части

Этот метод, называемый сортировкой простым выбором, в некотором смысле противоположен сортировке простыми включениями; при сортировке простыми включениями на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готового массива для нахождения места включения; при сортировке простым выбором рассматриваются все элементы входного массива для нахождения элемента с наименьшим ключом, и этот один очередной элемент отправляется в готовую по-

следовательность. Алгоритм и процедура сортировки массива *a[n]* простым выбором **на псевдокоде.**

1.Для всех i от 0 до n-2 выполнить:
1.1. Взять очередной i-й не отсортированный элемент и сохранить его в рабочей переменной min
1.2. Присвоить переменной min_pos значение i;
1.3. Найти минимум в части массива от a[i+1] до a[n-1] и запомнить его позицию в переменной min_pos;
1.4. Поменять местами элементы a[i] и a [min_pos];

```
void SortSelect (int *a,int size) {
// последовательный перебор всех элементов кроме последнего
for (int i=0;i<size-1;i++) {
int min = a[i]; //присвоение переменной минимум текущего элемента
int index_min = i; // запоминаем индекс текущего элемента
// поиск минимума в части массива от i+1 до конца
for (int j=i+1;j< size;j++)
if (a[j] < min)
{
min = a[j]; // запоминаем текущий найденный минимум
index_min = j; // запоминаем его индекс
}
//обмен местами текущего элемента и найденного минимального
a[index_min] = a[i];
a[i] = min;
}
}
```

Анализ сортировки простым выбором. Очевидно, что число C сравнений ключей не зависит от начального порядка ключей. В этом смысле можно сказать, что сортировка простым выбором ведет себя менее естественно, чем сортировка простыми включениями. Мы получаем: $C = \frac{1}{2}(n^2 - n)$

$$M_{min} = 3(n-1)$$

Минимальное число пересылок M равно:
в случае изначально упорядоченных ключей и принимает наибольшее значение:

$$M_{max} = \left[\frac{n^2}{4}\right] + 3(n-1) \qquad M_{cp} = n(\ln n + \gamma)$$

Можно сделать вывод, что обычно алгоритм сортировки простым выбором предпочтительней алгоритма сортировки простыми включениями, хотя в случае, когда ключи заранее рассортированы или почти рассортированы, сортировка простыми включениями все же работает несколько быстрее.

СОРТИРОВКА ПРОСТЫМ ОБМЕНОМ (Пузырьковая)

Классификация методов сортировки не всегда четко определена. Оба представленных ранее метода можно рассматривать как сортировку обменом. Однако в этом разделе

мы остановимся на методе, в котором обмен двух элементов является основной характеристикой процесса. Приведенный ниже алгоритм сортировки простым обменом основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут рассортированы все элементы.

Как и в предыдущих методах простого выбора, мы совершаем повторные проходы по массиву, каждый раз просеивая наименьший элемент оставшегося множества, двигаясь к левому концу массива. Если мы будем рассматривать массив, расположенный вертикально, а не горизонтально и представим себе элементы пузырьками в емкости с водой, обладающими «весами», соответствующими их ключам, то каждый проход по массиву приводит к «всплыванию» пузырька на соответствующий его весу уровень.

1.Для всех i от 1 до n-1 выполнять :

1.1. Слева направо поочередно сравнивать два соседних элемента и если их взаиморасположение не соответствует заданному условию упорядоченности, то менять их местами.

На приведенном рисунке изображен только один шаг (просмотр).

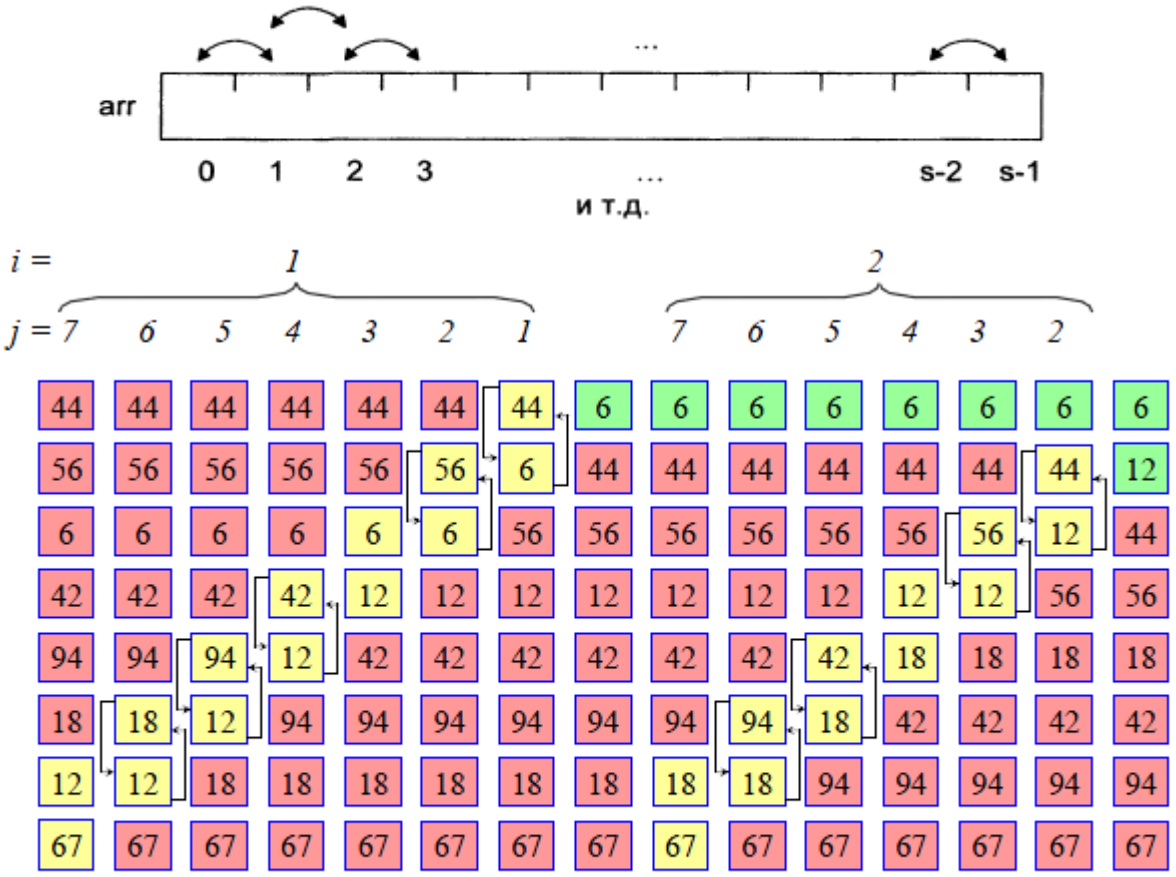


Рис. 3. Пример двух проходов сортировки методом пузырька: ■ – отсортированная часть, ■ – входная часть, ■ – пара сравниваемых элементов.

```
void SortBubble(int *a, int size)  {
 //повторять проходы по массиву n-1 раз
 for(int i=1;i< size;i++)
{
//проход с n – 1-го элемента вверх до i-го
     for (int j= size -1; j >= i; j--)
     if (a[j-1]>a[j])
```

```
     {
// обмен элементов в случае неправильного порядка
     int x = a[j-1]; a[j-1] = a[j];
     a[j] = x;
      }
} }
```

Название метода отражает его суть. ”Легкие”элементы массива ”всплывают” вверх(в начало), а ”тяжелые” опускаются вниз(в конец).

Пузырьковая сортировка проходит по массиву снизу вверх. Каждый элемент массива сравнивается с элементом, который находится непосредственно над ним. И если при их сравнении оказывается, что они удовлетворяют условию их перестановки, то она выполняется. Процесс сравнений и перестановок продолжается до тех пор, пока ”легкий” элемент не “всплывет” вверх. В конце каждого прохода по массиву, верхняя граница сдвигается на один элемент вниз(вправо). Сортировка продолжается анализируя все меньшие массивы.

Внутренний цикл for выполняет проход по подмассиву в направлении обратном внешнему циклу. Если изменить направление этих циклов, то ?

Шейкер сортировка

В этом методе учитывается тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные. Тогда просмотр имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. Если же просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то получим шейкер-сортировку. Ниже приведен пример программы, использующей шейкер-сортировку элементов массива размерностью `size`.

```
void ShakerSort(int *a, int size)
{
cout<<"Shaker Sort Mode:\n";
// объявление переменных левой и правой границ для текущего  участка  массива
// объявление переменной last для хранения индекса последнего обмена

int left = 1, right = size-1, last = right;
do  // повторять до тех пор пока границы не сомкнутся
{
for (int j = right; j >= left; j--)
// движение справа налево
if (a[j-1] > a[j]) // проверка условия обмена пары
{  // обмен элементов в случае неправильного порядка
int temp = a[j-1];
a[j-1] = a[j];
a[j] = temp;
```

```
last = j; // запоминание индекса последнего обмена
}
left = last + 1; //расчет новой левой границы
for (int j = left; j < right+1; j++)
// движение слева направо
if (a[j-1] > a[j])
// проверка условия обмена пары
{ // обмен элементов в случае неправильного порядка
int temp = a[j-1];
a[j-1] = a[j];
a[j] = temp;
last = j; // запоминание индекса последнего обмена
}
right = last - 1; //расчет новой правой границы
} while(left < right); // повторять до тех пор пока границы не сомкнутся
}
```

Этот алгоритм оптимизирует «пузырьковую» сортировку. В случае, когда последние элементы упорядочены, можно заметить что соответствующие проходы никак не влияют на их порядок. Очевидный способ улучшить данный алгоритм – это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то это означает, что алгоритм может закончить работу. Этот процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и место (индекс) последнего обмена. Ведь ясно, что все пары соседних элементов с индексами, меньшими этого индекса k, уже расположены в нужном порядке. Поэтому следующие проходы можно заканчивать на этом индексе, вместо того чтобы двигаться до установленной заранее нижней границы *i*. Однако внимательный программист заметит здесь странную асимметрию: один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только на один шаг на каждом проходе. Например, массив {12 18 42 44 55 67 94 06} будет рассортирован при помощи метода пузырька за один проход, а сортировка массива {94 06 12 18 42 44 55 67} потребует семи проходов. Эта неестественная асимметрия подсказывает третье улучшение: менять направление следующих один за другим проходов. Полученный в результате алгоритм называют шейкер-сортировкой.

Число сравнений в алгоритме простого обмена равно:

$$C = \frac{1}{2}(n^2 - n)$$

Минимальное, среднее и максимальное количества присваиваний равны:

$$M_{\min} = 0, \quad M_{cp} = \frac{1}{4}(n^2 - n), \quad M_{\max} = \frac{1}{2}(n^2 - n)$$

Анализ улучшенных методов, особенно метода шейкер-сортировки, довольно сложен. Наименьшее число сравнений есть $C_{min} = n - 1$. Среднее число сравнений пропорционально $1/2[n^2 — n(k_2 + \ln n)]$. Но оказывается, все предложенные выше усовершенствования никоим образом не влияют на число обменов, они лишь уменьшают число избыточных

повторных проверок. К сожалению, обмен двух элементов – обычно намного более дорогостоящая операция, чем сравнения ключей поэтому эти усовершенствования дают значительно меньший эффект, чем можно было бы ожидать.

Сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором, и действительно, сортировка методом пузырька вряд ли имеет какие-то преимущества, кроме своего легко запоминающегося названия.

Алгоритм шейкер-сортировки выгодно использовать в тех случаях, когда известно, что элементы уже почти упорядочены – редкий случай на практике.

Метод Хора

Алгоритм быстрой сортировки (сортировка с разделением) − это усовершенствованный метод сортировки, основанный на принципе обмена. Он построен на основе идеи разбиения массива на разделы.

Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно производить обмены элементов на больших расстояниях.

Предположим, что даны n элементов с ключами, расположенными в обратном порядке. Их можно рассортировать, выполнив всего $n/2$ обменов, если сначала поменять местами самый левый и самый правый элементы и так постепенно продвигаться с двух концов к середине. Разумеется, это возможно, только если мы знаем, что элементы расположены строго в обратном порядке.

Рассмотрим следующий алгоритм: выберем случайным образом какой-то элемент (назовем его x), просмотрим массив, двигаясь слева направо, пока не найдем элемент $a_i > x$. Затем просмотрим его справа налево, пока не найдем элемент $a_i < x$. Теперь поменяем местами эти два элемента и продолжим процесс «просмотра с обменом», пока два просмотра не встретятся где-то в середине массива.

В результате массив разделится на две части: левую – с ключами меньшими, чем x, и правую – с ключами большими x.

Если, например, в качестве x выбрать средний ключ, равный 42, из массива ключей {44 55 12 42 94 06 18 67}, то для того, чтобы разделить массив, потребуются два обмена: конечные значения индексов $i = 5$ и $j = 3$. Ключи $a_1, ..., a_{i-1}$ меньше или равны ключу $x = 42$, ключи $a_{j+1}, ..., a_n$ больше или равны x.

$a[k] \le x$, *при* $k = 1 \ldots i-1$;
$a[k] \ge x$, *при* $k = j+1 \ldots n$;
$a[k] = x$, *при* $k = j+1, i-1$

Этот алгоритм очень прост и эффективен. В его правильности можно убедиться на основании того, что два первых утверждения являются вариантами оператора цикла с постусловием. Вначале, при $i = 1$ и $j = n$, они, очевидно, истинны, а при выходе с $i > j$ они предполагают, что получен нужный результат.

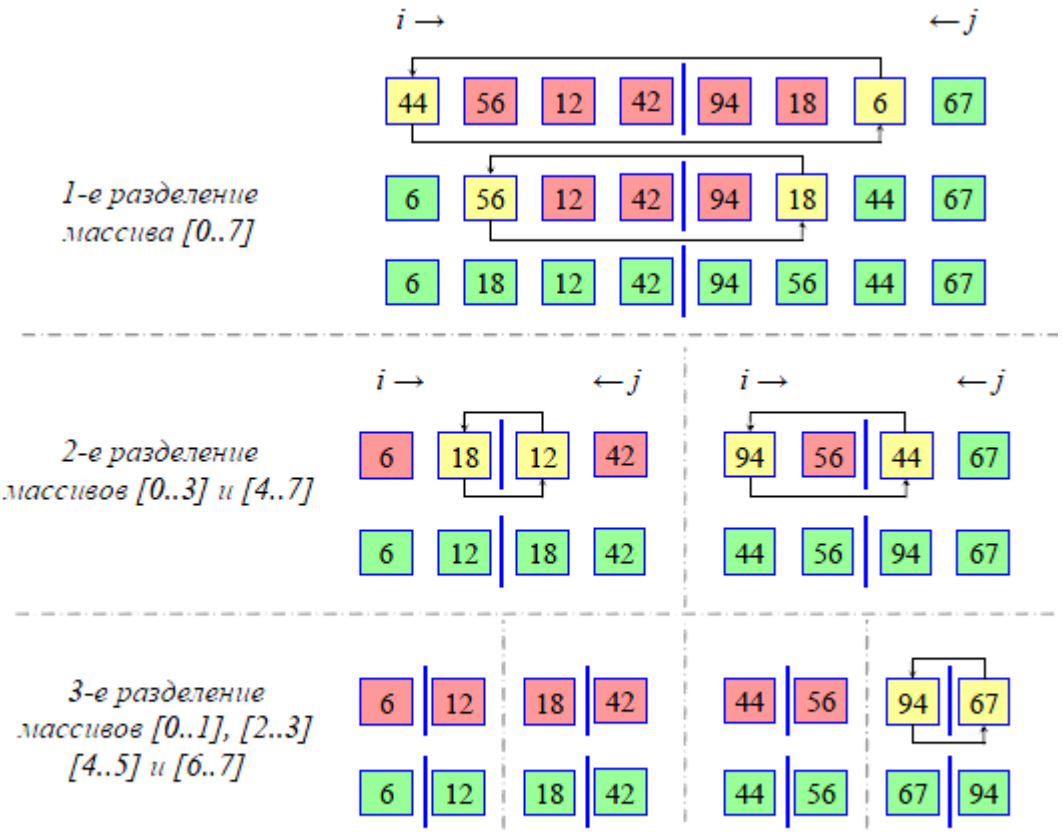


Рис. 4. Пример сортировки разделением: – текущая отсортированная часть, – входная часть, – текущая пара обмениваемых элементов.

Наша цель – не только разделить исходный массив элементов на большие и меньшие, но также рассортировать его. Однако от разделения до сортировки всего лишь один небольшой шаг: разделив массив, нужно сделать то же самое с обеими полученными частями, затем с частями этих частей и т. д. пока каждая часть не будет содержать только один элемент.

Процедура *QuickSort* рекурсивно вызывает сама себя каждый раз для новой части исходного массива, границы которой определяется парой индексов *left* и *right*. Такое использование рекурсии в алгоритмах – очень мощное средство.

```
void QuickSort(int a[], int left, int right)
{
//присвоение параметрам счетчиков текущих левой и правой границ части
//массива
int i = left, j = right;
// сохранение в mid значения среднего элемента
int mid = a[(left +right)/2];
do  // повторять пока проходы слева и справа не перехлестнутся
{
//проход слева до первого попавшегося элемента больше среднего
while (a[i] < mid) i++;
//проход справа до первого попавшегося элемента меньше среднего
while (a[j] > mid) j--;
if (i <= j)  // обмен местами найденных элементов
```

```
{
int temp = a[i];
a[i] = a[j];
a[j] = temp;
i++; j--;          // переход к следующим для каждого прохода значениям
}
} while(i < j); // повторять пока проходы не перехлестнутся
                // рекурсивное повторение алгоритма для разделенных частей массива
if ( left < j ) QuickSort(a,left,j);
if (i < right) QuickSort(a, i,right);
}


. . .
 QuickSort(b,0,n1);
```

Выразить этот же алгоритм можно в виде нерекурсивной процедуры, но это потребует более сложной реализации, т.к. необходимы некоторые дополнительные операции для хранения и обработки информации о границах подмассива.

Основа итеративного решения – ведение списка запросов на разделения, которые еще предстоит выполнить. После каждого шага нужно произвести два очередных разделения, и лишь одно из них можно выполнить непосредственно при следующей итерации, запрос на другое заносится в список. Важно, что запросы из списка выполняются в обратной последовательности. Это предполагает, что первый, занесенный в список запрос, выполняется последним и наоборот; список ведет себя как пульсирующий стек. В нерекурсивной версии быстрой сортировки каждый запрос представлен левым и правым индексами, определяющими границы части, которую впоследствии нужно будет разделить.

Анализ быстрой сортировки. Для того чтобы проанализировать свойства быстрой сортировки, мы должны сначала изучить поведение процесса разбиения.

После выбора границы x процессу разбиения подвергается весь массив. Таким образом, выполняется ровно n сравнений. Число обменов можно оценить при помощи следующего вероятностного рассуждения.

Предположим, что множество данных, которые нужно разделить, состоит из n ключей $1, \ldots, n$, и мы выбрали x в качестве границы. После разделения x будет занимать в массиве позицию x. Число требующихся обменов равно числу элементов в левой части $x–1$, умноженному на вероятность того, что ключ нужно обменять. Ключ обменивается, если он не меньше чем x. Вероятность этого равна $(n – x + 1)/n$.

Ожидаемое число обменов вычисляется при помощи суммирования всех возможных вариантов выбора границы и деления этой суммы на n:

$$M = \frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}(n-x+1) = \frac{n}{6} - \frac{1}{6n}$$

Следовательно, ожидаемое число обменов равно приблизительно $n/6$.

Если предположить, что нам очень везет, и мы всегда выбираем в качестве границы медиану, то каждое разделение разбивает массив на две равные части и число проходов, необходимых для сортировки, равно $\log n$. Тогда общее число сравнений составит $n \log n$, а общее число обменов – $(n/6) \log n$. Разумеется, нельзя ожидать, что мы все время будем попадать на медиану. На самом деле, вероятность этого равна всего лишь $1/n$. Но к удив-

лению, если граница выбирается случайным образом, эффективность быстрой сортировки в среднем хуже оптимальной лишь в *2 ln 2* раз.

Однако быстрая сортировка все же имеет свои «подводные камни». Прежде всего, при небольших значениях *n* ее эффективность невелика, как и у всех усовершенствованных методов. Ее преимущество по сравнению с другими усовершенствованными методами заключается в том, что для сортировки уже разделенных небольших подмассивов легко можно применить какой-либо простой метод. Это особенно ценно, если говорить о рекурсивной версии программы. Тем не менее, остается проблема наихудшего случая. Как тогда ведет себя быстрая сортировка? Ответ, к сожалению, разочаровывает. Здесь проявляется слабость быстрой сортировки, которая в таких случаях становится «медленной сортировкой». Рассмотрим, например, неблагоприятный случай, когда каждый раз в качестве *x* выбирается наибольшее значение в подмассиве. Тогда каждый шаг разбивает сегмент из *n* элементов на левую часть из *n – 1* элементов и правую часть, состоящую из одного элемента. В результате вместо *log n* необходимо *n* разбиений, и скорость работы в наихудшем случае оказывается порядка n^2.

Очевидно, что эффективность алгоритма быстрой сортировки определяется выбором элемента *x*. В нашем примере программа выбирает в качестве *x* элемент, расположенный посередине. Заметим, что почти с тем же успехом можно было бы выбрать либо первый, либо последний элемент: a [left] или a [right]. Но при таком выборе наихудший вариант встретится, когда массив уже предварительно рассортирован; быстрая сортировка в этом случае проявляет определенную «неприязнь» к тривиальной работе и предпочитает беспорядочные массивы. При выборе среднего элемента в качестве границы это странное свойство быстрой сортировки менее заметно, так как уже рассортированный массив становится оптимальным случаем! Действительно, средняя скорость работы здесь оказывается несколько выше, если выбирается средний элемент. Хоар считает, что выбор *x* должен быть «случайным», или в качестве *x* нужно выбирать медиану из небольшого числа ключей (скажем, из 3-ех). Такой осмотрительный выбор почти не влияет на среднюю скорость быстрой сортировки, но значительно улучшает скорость в худшем случае. Как мы видим, быстрая сортировка напоминает азартную игру, где следует заранее рассчитать, сколько можно позволить себе проиграть в случае невезения.

Быстрая сортировка (Quicksort), в варианте с минимальными затратами памяти — сложность алгоритма: $O(n \log n)$ — среднее время, $O(n^2)$ — худший случай; широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине. При использовании $O(n)$ дополнительной памяти, можно сделать сортировку устойчивой.

Выводы по простым методам сортировки

1. В простых методах сортировки массивов время сортировки пропорционально $size^2$.

2. Более точные оценки производительности простых методов сортировки массивов показывают, что наиболее *быстрой* является *сортировка вставками,* а наиболее *медленной - сортировка обменом.*

3. Несмотря на плохое быстродействие, простые алгоритмы сортировки следует применять при малых значениях *size.*

4. Наряду с простыми алгоритмами сортировки массивов существуют сложные алгоритмы сортировки, обеспечивающие время сортировки, пропорциональное не $size^2$, а $size \cdot \log_2(size)$. При больших значениях *size* они обеспечивают существенный выигрыш.

Общая процедура заключается в выборе пограничного значения, называемого *компарандом*, которое разбивает сортируемый массив на две части. Все элементы, значения которых больше пограничного значения, переносятся в один раздел, а все элементы с меньшими значениями - в другой.

Затем это процесс повторяется для каждой из частей и так до тех пор, пока массив не будет отсортирован.

Метод Хоора (Charles Antony Richard Hoare, 1962 г.), называемый также методом быстрой сортировки (Quicksort) основывается на следующем:

находится такой элемент, который разбивает множество на два подмножества так, что в одном все элементы больше, а в другом - меньше делящего. Каждое из подмножеств также разделяется на два, по такому же признаку. Конечным итогом такого разделения станет рассортированное множество. Рассмотрим один из вариантов реализации сортировки Хоора.

В процедуре сортировки сначала выберем срединный элемент. Далее, в слева от срединного элемента выбираются элементы большие, а в правой – меньшие срединного. Найденные элементы меняются местами. Это выполняется пока левая и правая границы не равны. В результате будут получены два подмножества. Если эти подмножества существуют, то выполняем по отдельности их сортировку.

Например, необходимо рассортировать массив: 13,3,23,19,7,53,29,17.

Переставляемые элементы будем подчёркивать, средний − выделим жирным шрифтом, а элемент попавший на своё место и не участвующий в последующих сравнениях − наклонным шрифтом. Индекс i будет задавать номер элемента слева от среднего, а j справа от среднего.

13 3 23 **19** 7 53 29 17
13 3 17 **19** 7 53 29 23
13 3 17 7 19 53 29 23
делим на два подмножества 13 **3** 17 7 19 **53** 29 23
Выделяем подмножество
3 13 17 7 19 23 29 53
13 **17** 7
Выделяем подмножество 13 **7** 17
Результат: 3 7 13 17 19 23 29 53

```
// l – левая r – правая границы сортируемого массива
void hoar(int *ms, int l, int r) // передаем левую/правую границы
{ int i, j, k;
int sr = ms[(l+r)/2]; // срединный элемент
i = l; j = r; // начальные значения границ массива
do
{ while(ms[i] < sr) i++; // ищем слева элемент больше среднего
while(ms[j] > sr) j--; // ищем справа элемент меньше среднего
if(i <= j) // если левая граница не прошла за правую
{ k = ms[i]; // перестановка элементов
ms[i] = ms[j];
```

```
ms[j] = k;
i++; j--; // переходим к следующим элементам
}
} while(i <= j); // пока границы не совпали
// массив разбит на два подмножества
if(i < r) // если есть что-нибудь справа
hoar(ms,i,r); // сортируем правый подмассив
if(j>l) // если есть что-нибудь слева
hoar(ms,l,j); // сортируем левый подмассив
}
```

Еще один вариант программы сортировки Хоора:

```
void hoor(int *a,int l,int r)
{ int i,las;
if(l>=r) return;
swap(a,l,(l+r)/2); // делящий эл-т переносится в a[l] (a[l]<->a[(l+r)/2])
las=l; // позиция последнего элемента большего
// чем делящий
for(i=l+1;i<=r;i++) // деление [l,r] на [l,las-1] и [las+1,r]
if(a[i]<a[l]) swap(a,++las,i);
swap(a,l,las);
hoor(a,l,las-1); // сортировка для [l, las-1]
hoor(a,las+1,r); // сортировка для [las+1, r]
}
void swap(int *a,int i,int j)
{ int tmp; // функция замены i и j элементов в массиве a
tmp=a[i];
a[i]=a[j];
a[j]=tmp;
}
```