Назначение конструктора

В C++ при определении переменных часто их сразу инициализируют. Например,
int x = 5;

String s; // s – объект класса String

Хотим проинициализировать его пустой строкой.
Для структур эта инициализация выполняется так:
String s = {“”, 0};

s.len = 0; s.line[0] = ’\0’;   // ??????????????

Для объектов класса такая инициализация запрещена в силу принципа инкапсуляции

Проблема

Внутри описания класса инициализировать
нельзя по синтаксису структуры,
но и вне класса записать
s.len = 0; s.line[0] = ’\0’;
тоже нельзя, т.к. *член-данные из части private недоступны*.

(Заметим, что если определить их в части *public*, то их можно инициализировать как структуру, т.е.
String s = {“”, 0};)

Выход

Инициализацию должна выполнять *специальная член-функция класса.*

Конструктор!

Определение. Член-функция класса, предназначенная для инициализации член-данных класса при определении объектов *класса, называется конструктором.*

Конструктор всегда имеет имя класса.

Конструктор в классе String

Объявление :

String();

Определение конструктора:

String:: String() { len = 0; line[0] = '\0';} (1)

Определение объектов:

String s1, s2;

.

Конструктор в классе String

Конструктор *всегда вызывается неявно*,
и выполняет инициализацию объектов

Так как конструктор не имеет аргументов, то он называется *конструктором по умолчанию*.

Несколько конструкторов

```
String:: String(const char * s)                    (2)
{ for( len = 0; line[len] != '\0'; len++)
        line[len] = s[len];
}
```

работает конструктор с аргументом (2)

Тогда объекты можно определить таким образом

```
String s1, s2("Иванов"), s3 = String("Петров");
```

работает конструктор по умолчанию (1)

Для объекта s3 конструктор задается явно(но так конструктор используется редко)

Заметим, что в классе должен быть один конструктор по умолчанию и один или несколько с аргументами.

Особенности конструктора, как функции:

1. Главная - конструктор не имеет возвращаемого значения (*даже void*), так как его единственное назначение – инициализировать собственные член-данные объекта;
2. Конструктор имеет имя класса;
3. Конструктор всегда работает **неявно** при определении объектов класса

Недостаток определенного класса String

- это то, что он берет для каждого объекта 259 байтов памяти, хотя фактически использует меньше

```
class String{ char *line; int len;
                public:
                 ....
                };
```

Тогда конструкторы надо определить иначе, т.к. кроме инициализации *значений* член-данных, они должны брать память в динамической области для поля *line*.

Другие конструкторы

В классе объявим 2 конструктора

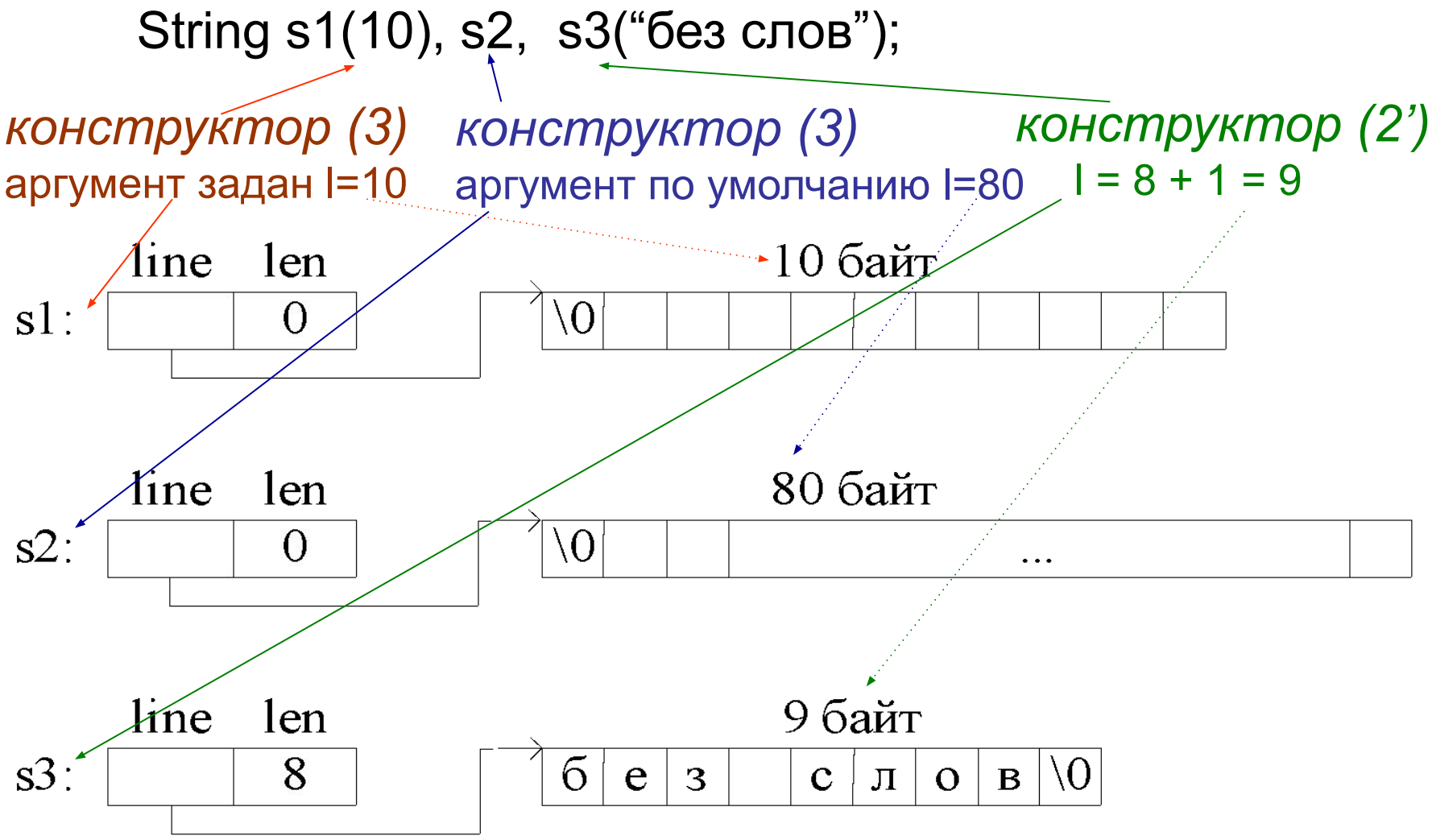
```
String(int l = 80);     //с аргументом по
                                // умолчанию

String (const char *); //с аргументом
// строкой
```

```
String:: String(int l) // l=80 – не повторять! (3)
{line = new char [l]; len=0;
  line[0]='\0';
}

String::String(const char * s)                    (2')
{line = new char [strlen(s)+1];// для нуль-кода
 for( len = 0; line[len] != '\0'; len++)
    line[len] = s[len];
}
```

Пример использования



Замечание

В классе должен быть

или конструктор по умолчанию без аргументов вида (1),

или конструктор с аргументом по умолчанию вида (3)

String ss;

‘Ambiguity between ‘String::String()’ and ‘String::String(int)’ -

‘Двусмысленность между String() и String(int)’

Инициализация значением другой переменной

В C++ кроме инициализации константным значением

```
int x = 5;
...
x++;
...
```

используется и такая инициализация данных

```
int y = x; // инициализация одного данного
   // значением другого
```

В классе String подобная инициализация может привести к *ошибкам*.

```
String s(“паровоз”);
...
String r = s; // определение объекта r и
                    // инициализация его значением                    //
   объекта s
r.Index(4) = ‘х’ ; r.Index(6) = ‘д’;// изменим на парохо**д**

s.Print(); r.Print();
```

Увидим, что выведется ***пароход*** в обоих случаях. Это плохо.

Разберемся, почему это происходит

При определении объекта String s(“паровоз”); работает конструктор,

```
String::String(const char * s)
(2’)
{ line = new char [strlen(s)+1]; // для нуль-кода
for( len = 0; line[len] != ‘\0’; len++)
line[len] = s[len];
}
```

который возьмет память в динамической области 8 байтов

и адрес первого запишет в поле s.line.

Затем цикл for занесет в поле *line слово *паровоз* и одновременно определит len = 7.

При определении объекта r
String r = s; // или String r(s);
компилятор просто выполняет *копирование полей*
r.line = s.line и r.len = s.len

А значит поле *r.line* будет показывать на ту же динамическую область!

И при выполнении операторов
r.Index(4) = ‘x’ ; r.Index(6) = ‘д’;

изменятся оба объекта. ☹

Что неграмотно и недопустимо !

Поэтому для инициализации одного объекта другим надо определить специальный *конструктор копирования*

X :: X(X&); // где X - имя класса

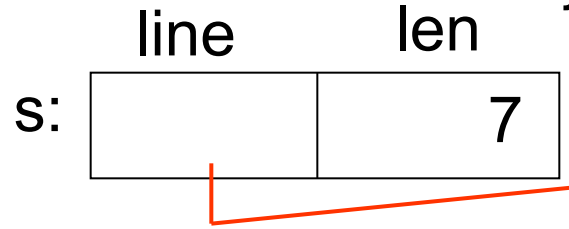
String(String &);

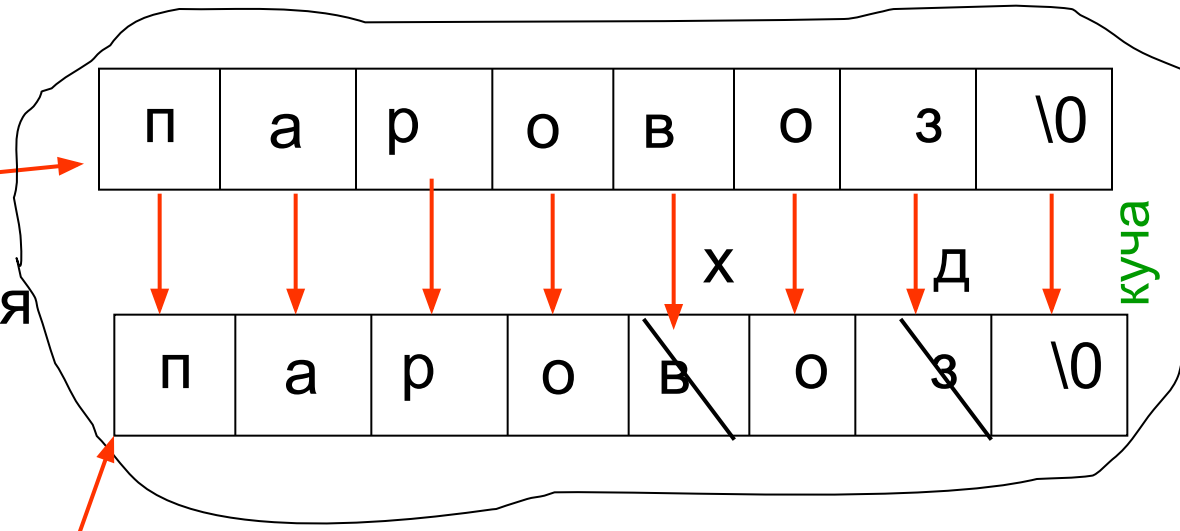
const

```
String::String(String  &s)
{ line = new char[ s.len + 1 ];
  for( len = 0; line[len] != '\0'; len++)
            line[len] = s.line[len];
 }
```

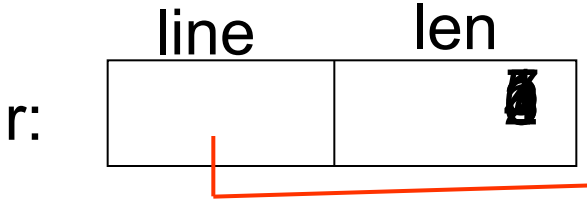
Тогда иници
String r = s;
выполнится

```
{ line = new char[ s.len+1 ];
  line = new char[ s.len + 1 ];
  for( len = 0; line[len] != '\0'; len++)
        line[len] = s.line[len];
}
```





Конструктор копирования возьмет для объекта r



новую динамическую память длиной s.len + 1 байтов.

И цикл for затем запишет из объекта s в поле *r.line* слово *паровоз, в поле r.len длину 7.*

При выполнении операторов

r.Index(4) = 'x' ; r.Index(6) = 'д';

значение s.line теперь не изменится ! ☺

Все верно

s.Print(); // выведет ‘паровоз’

r.Print(); // выведет ‘пароход’

Замечание

Конструктор копирования кроме рассмотренной инициализации работает также

- при передаче *значений* фактических аргументов-объектов в функцию

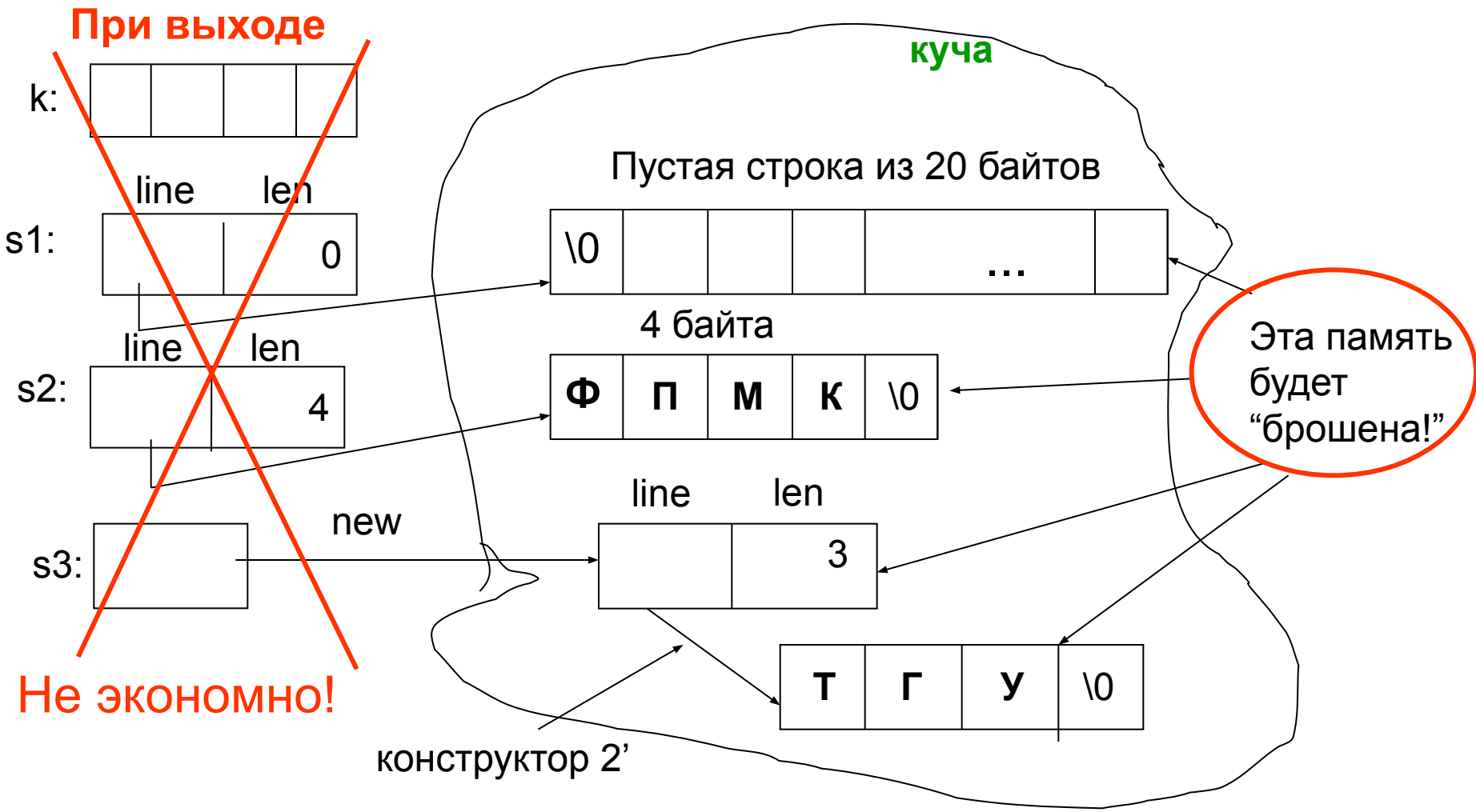
- при *возврате* результата-объекта из функции.

п 3.3. Деструктор

В языке С++ одним из важных моментов является освобождение памяти, занятой переменными, при выходе из функции.

```
void F()
{ int k;
  String s1(20),s2(“ФПМК”),*s3;
  s3= new String (“ТГУ”);
}
```

При выходе из функции *освобождается* память для локальных объектов, т.е. k, s1,s2, s3. Но рассмотрим внимательнее, как это будет реализовано.



Для того, чтобы при выходе из функций динамическая память, которая берется конструкторами объектов, *освобождалась автоматически*, надо задать специальную член-функцию

деструктор.

Деструктор

Определение. ***Деструктор*** - это член функция класса, предназначенная для освобождения динамической памяти, занимаемой член-данными класса, при выходе из функций. Деструктор имеет формат

~ имя_класса(){…}

Пример

Для класса String его можно определить таким образом

~ String() {delete [ ] line;}

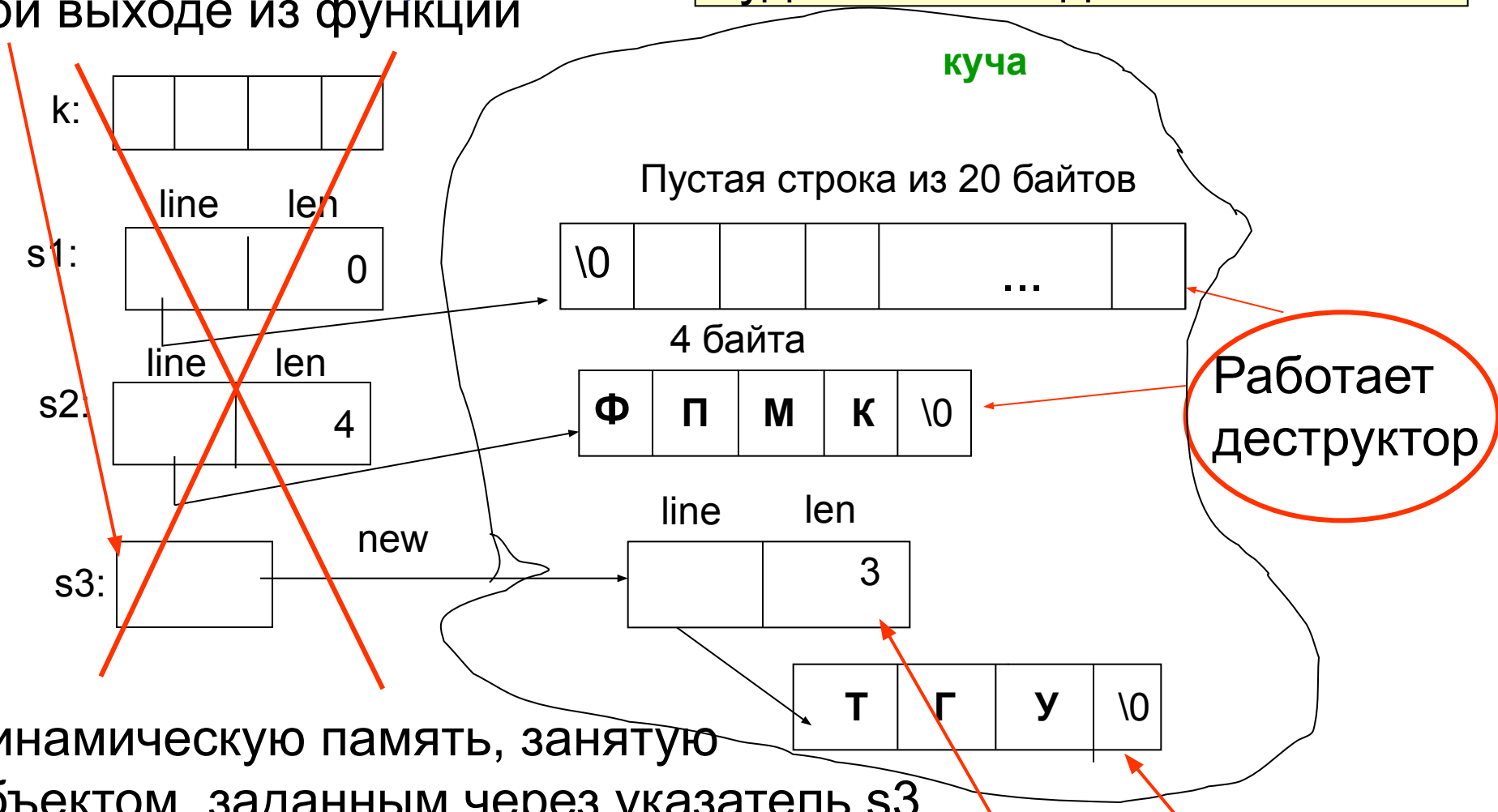
Пример

В этом случае при выходе из области видимости функции F() память для объектов s1, s2, которую брал конструктор для поля *line*, будет освобождена.
Заданный деструктор это будет делать *по умолчанию.*

Работает стандартное освобож
памяти от локальных данных
при выходе из функции

3) Стандартно от ячейки s3
при выходе из функции

Память по операции
delete s3;
будет освобождена в 3 этапа:



Динамическую память, занятую объектом, заданным через указатель s3, надо освобождать явно операцией
delete s3;

1) деструктором
2) операцией delete

Особенности деструктора как функции:

- он не имеет аргументов;
- он не возвращает значения;
- работает неявно для всех объектов при выходе из функций

Замечание. Работу деструктора можно *“увидеть”*, если в деструкторе задать какой-либо вывод.

```
~String()
   { printf(“\nРаботает деструктор класса String”);
     delete [ ] line; }
```

```
class String
 {char *line; int len;
   public:
  String(int l=80); // конструктор по умолчанию
  String(const char *); //конструктор с аргументом
  String(String &);       // конструктор копирования
  ~String()  { delete [] line;} // деструктор
   void Print()  { cout << line;}
   int  Len() { return len;}
   char & Index( int );
   void Fill( const char* );
  };
```

char & Index (int)

```
char & String::  Index (int i)
  { if(i<0 || i>=len)
      cout << ”\n Индекс за пределами “;  exit(0);}
     return line[i]; }
```

Тип возвращаемого значения char & - ссылка возвращает не просто значение символа, а ссылку на ячейку, где он находится.
Это и позволяет выполнить присвоение вида
r.Index (4) = ’x’;

Возвращаемый тип char

Если определить тип возвращаемого значения просто char, то присвоение вида
r.Index (4) = ’x’; (*)
было бы ошибочным, так как функция вернет *значение символа* *и* компилятор будет трактовать оператор (*), как присвоение одного кода символа другому коду, как в данном примере
‘в’=’х’;
что невозможно.

В других операциях в этом случае символ использоваться может, ***кроме присвоения ему нового значения***.