П.7 Дружественность

Пример. Пусть некоторая внешняя функция Show выводит строку в красивом виде - в ‘*’:

```
                const
void Show(√String &s)
{int   I, k = s.n;
 for(i = 0; i < k + 4; i++) cout << ‘*’;
 cout<<“* ” << s.line << “ *”;
 for(i = 0; i < k + 4; i++) cout << ‘*’;
}

String  s(“Hello”);
Show(s);
```

```
*********
* Hello *
*********
```

Очевидно, что задать такую функцию компилятор не позволит, так как будет
нарушена инкапсуляция член-данных ***len*** и ***line***.

Если все-таки необходимо разрешить некоторой не член-функции (внешней функции) использовать член-данные из части private какого-либо класса, ее можно объявить ***дружественной*** этому классу.

Формат объявления

```
class   String{
      friend    void   Show(String &, int, int);
       // в любом месте определения класса
                    char *line;
   …
      };
```

Кто они друзья?

1. внешняя по отношению к классу ***функция***, как в нашем примере;

2. член-функция известного на данный момент ***другого класса***

Например, функция f – член-функция класса A

```
class  A { … тип_возвр_знач  f(аргументы); ...};

class  B { …
friend  тип_возвр_знач  A ::  f(аргументы);
// сама f определяется в классе A
        ...    };
```

Друг 3

3. другой определенный (или объявленный) на данный момент ***класс***

```
class A; // упреждающее объявление
class B{   friend class A;
              ....          };
```

Такое объявление означает, что всем член-функциям класса A разрешается доступ ко всем член-данным класса B,

но не наоборот

Замечание1

Дружественность нужно использовать оптимально, так как она нарушает принцип инкапсуляции.

Замечание 2

Операции можно перегружать и как внешние дружественные классу функции. В этом случае одноместная операция имеет один аргумент - объект класса, а двуместная - 2: объект класса и второй операнд.

Пример. Перегрузка операции +, как внешней дружественной функции

```
class String{
                ...
friend  String  operator + (String &, String &);
                ...
};
```

```
String operator + (String &s, String &t)
{ String z(s.len + t.len + 1);
// определим локальную переменную
// суммарной длины, пустую строку
    strcpy(z.line, s.line);
    strcat(z.line, t.line);
    z.len = strlen(z.line);
    return z;
 }
```

Используется она так же, как и перегруженная в классе.

п8. *Перегрузка операций потокового ввода >> и вывода <<*

Библиотека *iostream* содержит стандартные классы ввода-вывода:

класс istream - потоковый ввод со стандартного устройства stdin (клавиатура),

класс ostream - потоковый вывод на стандартное устройство вывода stdout (монитор).

Рассмотрим их.

ostream

В классе *ostream* определена операция **<<**, перегруженная для форматного вывода базовых типов данных, т.е.

```
class ostream { ...
                public:
                ostream & operator <<(char *);
              ostream & operator <<(char);
                ostream & operator <<(double);
                ostream & operator <<(long int);
                  ...
                };
```

cout - это стандартное имя объекта - потока вывода, т.е. в системе есть описание
ostream cout;
Поэтому операцию
cout << x;
рассматриваем как двуместную: слева первый операнд - имя потока вывода, справа второй операнд – имя переменной вывода.
Так как возвращаемое значение – ссылка & на объект cout, то можно писать цепочки вывода.

Например, пусть задана переменная x

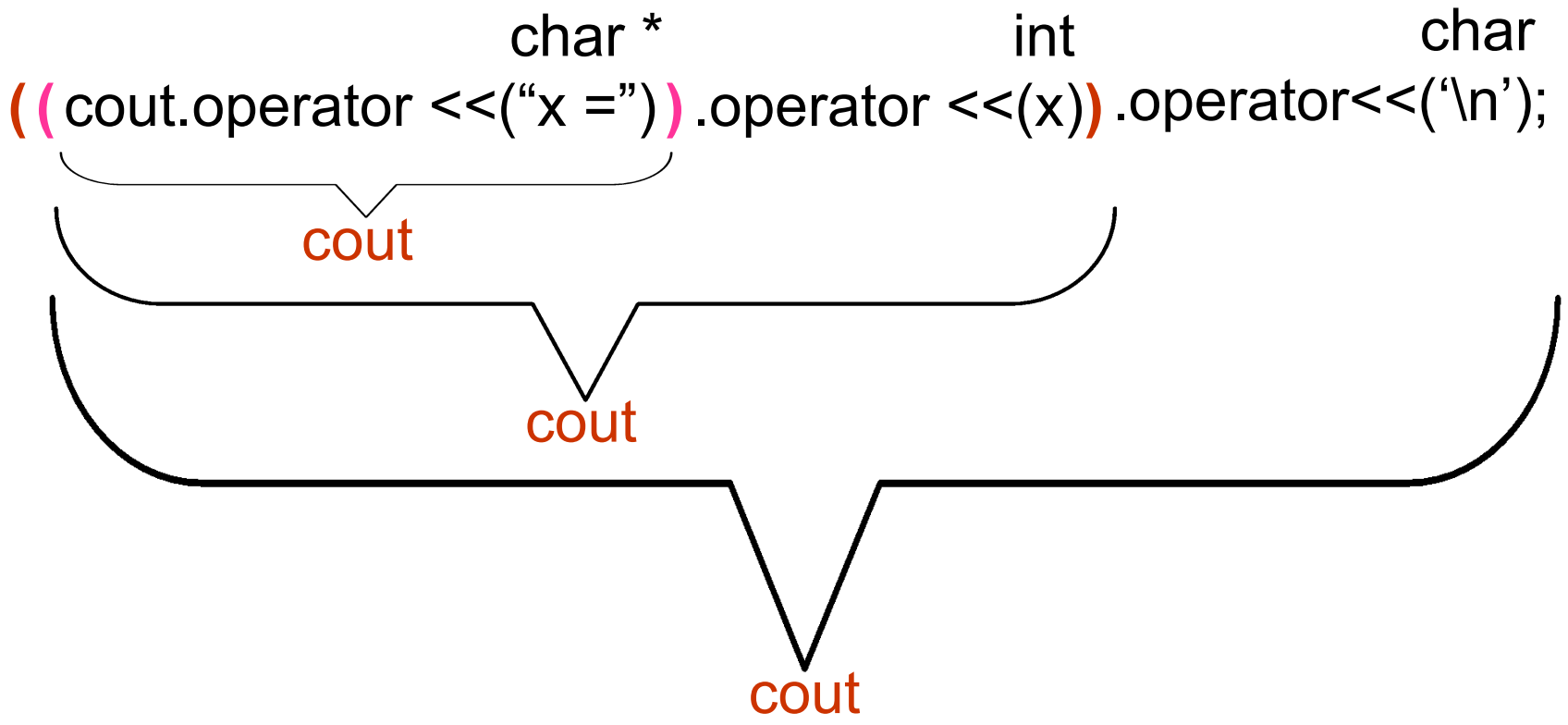
int x = 1185;

Цепочка вывода

cout << ”x = “ << x << ’\n’;

представляет собой последовательное

выполнение операции << с аргументами разного типа:



Это важно

Оператор, определенный как член-функция класса, первым операндом всегда имеет объект класса, т.е. ***this**. Первым операндом операции << является поток вывода, поэтому ее можно перегрузить для абстрактных типов ***только, как дружественную классу***.

<< для класса String

перегрузка может быть определена таким образом

```
class String{
                public:
                 ...
  friend ostream & operator<<(ostream &r, String &s)
                 { r << s.line;
                   return r;
                 }
        };
```

Теперь и для объектов класса String можно применять операцию **<<**:

```
String s("Иванов");
cout << s;
```

istream

В классе istream определена перегруженная операция >> для базовых типов данных

```
class istream{ ...
                    public:
                        istream & operator >>(char *);
                        istream & operator >>(char &);
                        istream & operator >>(long int &);
                        istream & operator >>(double &);
                        istream & operator >> (float &);
 .....
    };
```

Имеется определение стандартного имени **cin**:

istream cin;

Если определить переменную

int x;

то операция

cin **>>** x;

означает, что введенное число со стандартного устройства ввода передается в переменную x.

В этой операции cin - первый операнд, x – второй операнд.

Также можно писать цепочки ввода.

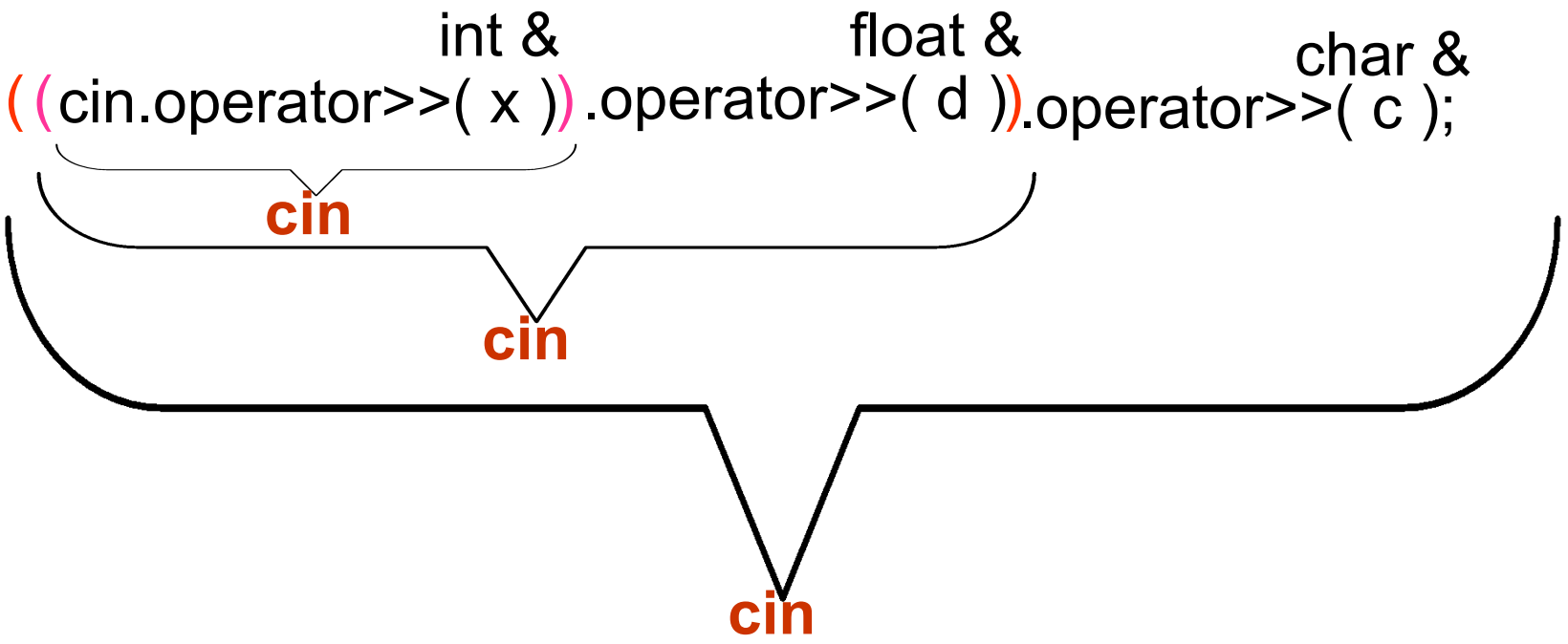
int x; float d; char c;

Цепочка ввода

cin >> x >> d >> c;

это последовательное применение операции

>> с аргументами разного типа:



>> для класса String

```
class String{ ...
                public:
                    ...
  friend istream & operator >>(istream &r, String &s)
          { char buf[80];
            cout<<"\nВведите строку, в конце Enter";
            gets(buf);
             String q(buf); // работает String ( char*)
             s = q;              // работает операция =
             return  r;
            }                    // работает  ~String для q
      };
```

Использование

```
String s1, s2(30);
cin >> s1 >> s2;
```

Замечание

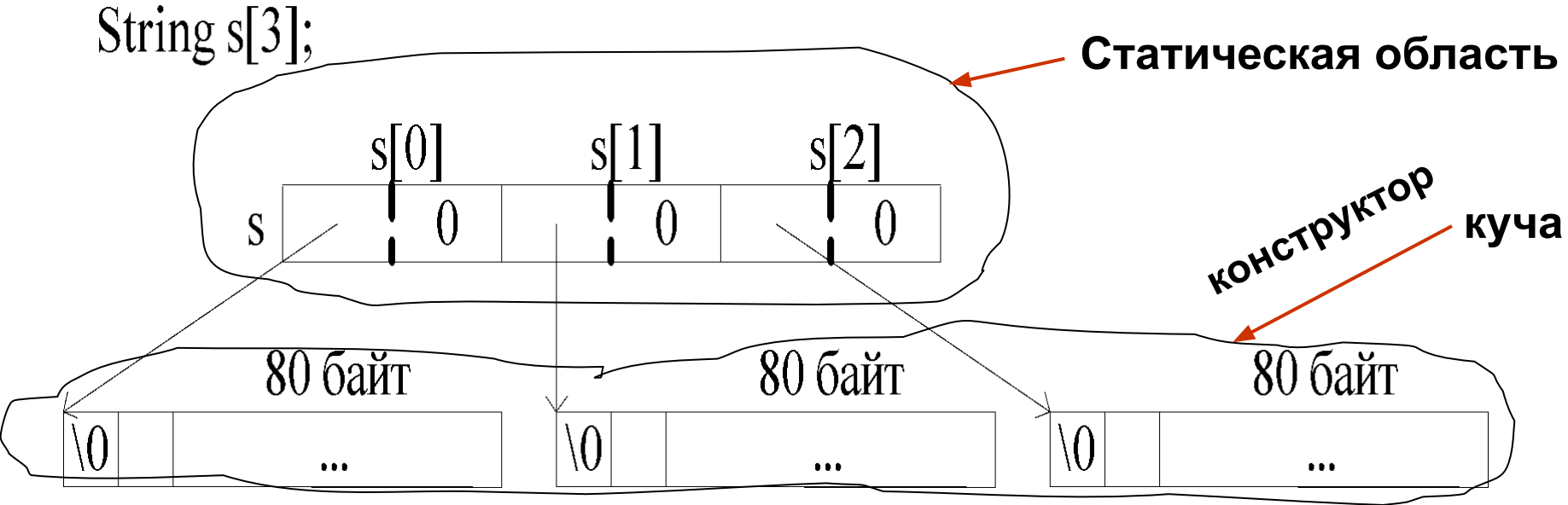
У второго аргумента операции вывода << тип ссылка & желателен (чтобы не выполнялось копирование), но не обязателен, а у операции ввода >> этот аргумент обязательно должен иметь тип

& - ссылка

Иначе
значение фактического аргумента не изменится
(не введется)!

п9. Массивы объектов

Массивы объектов определяются обычным образом:
String s[3];
3 объекта в статической области, каждый захватывает память конструктором по умолчанию по 80 байтов для пустой строки



Объекты, составляющие массив, конструктором с аргументами инициализируются каждый по отдельности,

например

String s1[3] = {String(“Иванов”), String(“Петров”)};

Если аргумент один, как в примере, то запись можно сократить

String s1[3] = {“Иванов”,”Петров”};

обратите внимание:

элемент **s1[2]** инициализируется по умолчанию пустой строкой.

Конструкторы можно комбинировать:

String ss[3]={12,20, “C++”};

Можно определить динамический массив:

String *sp = new String[4];

Массив из 4 объектов в динамической области. Для каждого объекта память по указателю ***line*** также в динамической области берется ***конструктором по умолчанию*** по 80 байтов.

Запомните!

Нельзя явно инициализировать массив объектов, определенных в динамической области.

Для таких случаев и должен быть предусмотрен конструктор по умолчанию.

~~String *sp = new String [5]= {10,30,"Что такое?",~~
~~"Нельзя так инициализировать?!",33};~~

// Да, так нельзя!!!

Использование массивов

String s1[3] = {String(“Иванов”), String(“Петров”)};

String *sp = new String[4]; String s[3];

String ss[3] = {12,20, “C++”};

s1[1].Print(); // вывод ”Петров”

sp[0] = ss[2]; // Работает перегруженная
// операция ‘=’ : sp[0] = ”C++”

s1[1] [0] = ’B’;// ‘Петров’ превратится в ‘Ветров’

станд перегруженная

s[0] = s1[0] + s1[0];

Работают 2 операции String::operator+ и String:: operator=

// вместо пустой строки s[0] получим два ‘Иванов’-ых, то есть “ИвановИванов”

Освобождение памяти

При выходе из функции память для массивов s, s1 и ss будет освобождаться в 2 этапа - сначала деструктором класса String для каждого элемента, затем стандартным деструктором от локальных полей len, line.

Ох уж эта память!

Однако для массива объектов, определенного в динамической области, надо ***явно освободить*** память при выходе из функции операцией
delete [] sp;
В этом случае освобождение происходит в 3 этапа:
- деструктором класса String,
- операцией delete от полей line и len каждого элемента массива
и, наконец,
- стандартным деструктором от поля sp.
Если этот оператор не задать, то будет освобождена память только от ячейки sp.

Не забывайте этого - берегите память!

Решение

```
Complex:: operator String()
      {char r1[25],r2[10];
       sprintf(r1,"%7.2f",re);
       sprintf(r2,"%7.2f",im);
       strcat(r1," + i*");
       strcat(r1,r2);
       String *s=new String(r1);
       return *s;
       }
```