САОД
ООП, классы и объекты в C++

А. Задорожный

2018

Вопросы для повторения

1. Из каких шагов состоит построение программы на C++?
2. Как в тексте программы обнаружить инструкции препроцессора? Приведите примеры инструкций препроцессора.
3. Назовите 4 основных типа данных, определенных в языке C++.
4. Можно ли в C++ написать x = y = 3? Почему?
5. Можно ли вызвать void foo(int & a) следующим образом: foo(x+y); Почему?
6. Проинтерпретируйте объявление второго параметра main.
   ```
   int main(int argc, char* argv[])
   ```
7. Сколько операций умножения в операторе *p = *p * 2?
8. Если массив w проинициализирован значениями 1, 2, 3. Чему равно значение выражения *w?
9. Чему равна величина p + i, где p – указатель на массив целых, а i – целое число?

Содержание

Классы, структуры, объекты*

Классы объявляются по аналогии с C#

```
class Date {
    int m_nYear;
    int m_nMonth;
    int m_nDay;
public:     //Указывается не перед каждым методом
    Date();  //Конструктор, как правило, public
    ~Date() {}
};              //Точка с запятой обязательна!
```

Объявления, как правило, в заголовочном файле (*интерфейс*). Реализация часто в *CPP*-файле

Реализация методов*

Date.cpp

```
#include “Date.h”

Date::Date () {  //Указываем класс
    m_nYear = 1;
     m_nMonth = 1;
     m_nDay = 1;
}
```

Такой конструктор по умолчанию создает объект-даты ***1 янв. 0001*** года.

Объявление и время жизни объектов*

```
#include “Date.h”
 int main(){
      Date d; // Это не ссылка, а объект!
      return 0;
}
```

- ***Локальные объекты создаются*** *при выполнении объявления.*
- ***Локальные объекты исчезают*** *по завершении блока в котором они были объявлены.*

- ***Глобальные объекты*** ***объявляются вне блоков****.*
- ***Глобальные объекты создаются*** *до начала работы программы, а* ***разрушаются*** *после завершения программы.*

Конструкторы инициализации*

```
class Date {
    int m_nYear;
    int m_nMonth;
    int m_nDay;
public:
    Date();
    Date(int year, int mon=1, int day=1);
    ~Date() {}
};
```

Параметры по умолчанию задаются при объявлении!

Конструкторы инициализации

```
Date::Date (int year, int mon, int day) {
  if (year < 1 || year > 10000)
        year = 1;
  if (mon < 1 || mon > 12)
        mon = 1;
  int vDays[12]={31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  if (mon < 1 || mon > 12)
        mon = 1;
  if (day < 1 || day > vDays[mon-1])
        day = 1;
  if (mon == 2)
        if(day == 29 && (year % 4 > 0 || (year % 100 == 0 && year % 400 >0))) {
                    day = 1;
                    mon = 3;
        }
  m_nYear = year;
  m_nMonth = mon;
  m_nDay = day;
}
```

Конструктор обеспечивает правильность создаваемой даты

```
Date d(2006, 10, 12), t(2006, 11), e(2006);
```

Структуры*

- Структуры – те же классы!
- По умолчанию в них действует область видимости public

```
struct Date {
private:
    int m_nYear;
    int m_nMonth;
    int m_nDay;
public:
    Date();
    ~Date() {}    //Конструктор, как правило, public
};                //Точка с запятой обязательна!
```

Структуры в C#

В C# между структурами и классами имеются существенные различия!

Все структуры в C# являются ***Value Type***! В том числе, числовые типы, перечисления и пр.

Все классы – ***Reference Type***

Наследование

```
class B    //Доступом к предку можно управлять
{
public:
      void boo();
};
```

```
class C: B
{
      …
};

C c;
c.boo();
```

```
class C: public B
{
      …
};

C c;
c.boo();
```

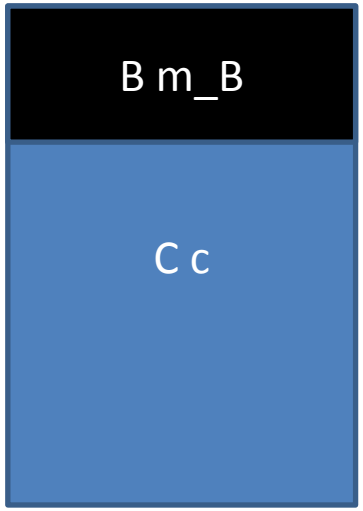
Включение и наследование

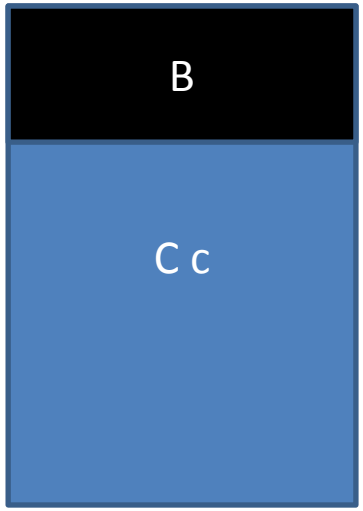
```
class B
{
public:
	void boo();
};

class C
{
public:
	B m_B;
};

C c;
c.m_B.boo();
```

Предки как неименованные члены класса*

class C{ public: B m_B} class C: public B {}

c.m_B.boo() c.boo()





Есть и существенные различия (см. комментарий к слайду).

Шаблоны

Шаблоны – абстракции типов и алгоритмов

Шаблон алгоритма:

```
template <class T> void Swap (T& a, T& b)
{ T c = a; a = b; b = c;}
```

Шаблон класса:

```
template <class T> class …
```
Далее определение класса

Параметры шаблонов, которые не являются типами

Вектора произвольной размерности с типом *double* или *float*, а может быть и *int*.

```
template <class T, int N> class vect{
      int n = N;
   public:
      T data[N];
      int GetLen(){ return n; }
};
```

Оператор определения типа

typedef vect<double, 2> Vector2D;

Теперь можно использовать

Vector2D v;

! Шаблоны, как и определения типа размещаются в заголовочных файлах!

Сам-по себе шаблон в результате компиляции не образует фрагмента кода. Он превращается в нужный метод или алгоритм на этапе компиляции вызывающей части кода!

Промежуточный итог

Узнали особенности классов C++ :

- Объявление объекта == создание.
  Не возникает = new …;

- Объявление отделено от определения;
  В определении (cpp) указываем контекст (<Класс>::);

- Более важна роль конструкторов;
  Конструктор “по умолчанию”, “инициализации”, …;

- Структуры – те же классы.
  По умолчанию public (для совместимости с “C”);

- При наследовании нужно указывать видимость предков.
  По умолчанию видимость private и методы предков не будут видны;

- Шаблоны позволяют определять абстракции алгоритмов и структур данных.
  Располагаются шаблоны в заголовочных файлах. Параметрами шаблонов являются типы данных и конкретные значения.

Контрольные вопросы

1. К чему приведет объявление класса в C++ без использования слова public?
2. Почему в конце объявления класса ставится точка с запятой?
3. Определите время жизни объекта. Сформулируйте правило, по которому возникают и исчезают временные объекты в C++.
4. В чем различие между объектами в C++ и C# (.Net)?
5. Опишите время жизни объектов в C# (.Net)?
6. Что такое “конструктор инициализации”?
7. Опишите правила определения и использования параметров по умолчанию. (Где они указываются и почему, как они расположены в списке параметров, как могут опускаться при вызове)
8. В чем различие в C++ между struct и class?
9. Как будет исправлена дата при вызове Date d(0, 15, 2006), если конструктор реализован как в лекции? Какая дата будет задана при вызове Date d(2007)?
10. В чем различие в наследовании в C# и C++? (по лекции)
11. Где располагаются шаблоны?
12. Что означает в параметре шаблона слово class?

Вызов функции и inline*

Вызов функции требует времени:

1. Вычислить и поместить в стек параметры
2. Запомнить адрес возврата
3. Переключить стек и передать управление на функцию
4. …
5. Переключить стек и вернуть управление в вызывающий код

Частые вызовы коротких функций снижают эффективность выполнения программ.

Функцию можно пометить квалификатором inline. При этом, компилятор не будет генерировать эту функцию, а в местах вызова сгенерирует код, эквивалентный вызову.

```
inline int max(int a, int b){return a>b? a: b;}
```

Переопределение операций
Inline методы*

```
class Date {
        int m_nYear;
        int m_nMonth;
        int m_nDay;
public:
        Date();
        Date(int year, int mon=1, int day=1);
        ~Date() {}
        bool operator < (const Date &d) const {
                return m_nYear < d.m_nYear
        || (m_nYear == d.m_nYear && m_nMonth < d.m_nMonth)
        || (m_nYear == d.m_nYear && m_nMonth == d.m_nMonth          &&
m_nDay < d.m_nDay);
        }
        int get_Year() const {return m_nYear;}
};

cout << (t < d) << " " << (d < e) << endl;
```

“Динамические” объекты

```
char *s = new char[128];
Date *p = new Date();
Date *pv = new Date[5];

delete [] s;
delete p;
delete [] pv;
```

- “Динамические” объекты создаются оператором new;
- За удаление таких объектов отвечает программа!

Указатели на объекты

При работе с динамическими объектами получаем указатели:

```
Date *p = new Date();
```

Можно и как раньше, получить ссылку на объект (*p), и обращаться к доступным методам или данным через точку

```
int y = (*p).get_Year();
```

Но! Принято (и удобнее) использовать специальный синтаксис:

```
int y = p -> get_Year();
```

Глобальные объекты

Правила:

Время жизни объектов ограничено блоком кода ({…})в котором они объявлены.

Глобальные объекты объявляются вне всех блоков.

Это определяет время их жизни.

Но как сделать видимыми во всех модулях (cpp-файлах)?

Для этого, их объявления выносят в заголовочные *h*-файлы!

Глобальные объекты*

Глобальный объект должен включаться в *h*-файл.

Что бы избежать создания его экземпляров во множестве объектных файлов (модулей) в языке введен квалификатор ***extern***

H-файл ----------

extern **Date UnixBase**;

Один из *CPP*-файлов ----------

Date UnixBase(1970,1,1);

В других *CPP*, где включен *h*-файл можно использовать UnixBase!

Статические поля*

Статические поля объявляются в классе с квалификатором ***static****

H-файл ----------

```
class Test
{
	public:
		static int Count;
};
```

В CPP-файле ----------

```
int Test :: Count = 0
```

В других CPP, где включен h-файл можно использовать Test :: Count!

Промежуточный итог

Узнали:

- Назначение и правила inline функций.
  Быстрее вызываются, определяем в h-файлах;

- Переопределение операций в C++;
  <тип >operator <операция> (<параметр>)…

- Динамические объекты и указатели;
  new, delete, оператор ->;

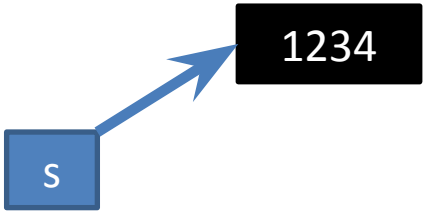
- Как правильно объявлять глобальные переменные и статические поля.
  Объявление в h с extern и размещение в cpp.
  Объявление static в классе и размещение в cpp;

Контрольные вопросы

1. Что означает запись o.**operator** = (t)? Как ее можно эквивалентно записать иначе?
2. *Переопределите операцию == для Date.*
3. Что такое и зачем нужны ***inline*** функции и методы?
4. Как объявить ***inline*** метод в классе?
5. Что такое “динамические” объекты? Как они создаются? Уничтожаются?
6. Что означает запись: **o->foo();**? Что такое **o**? **foo**?
7. Опишите объявление и использование глобальных переменных в C++.
   1. Как их объявить, что бы можно было использовать в разных модулях?
   2. Как добиться того, что бы был создан только 1 уникальный экземпляр глобальной переменной?
8. Опишите правила объявления статических полей класса в C++.

Виды конструкторов* Класс Str.

Класс для понимания роли конструкторов и деструкторов.




```
class Str {
    char* m_pszText;
public:
    Str ()    //Конструктор “по умолчанию” (default)*
        {m_pszText = new char[1]{0};}

    ~Str () { delete []m_pszText;}
};
```

Здесь деструктор обязателен!

Доработка Str*

Оператор *преобразования типа* (*Str=>char**)

operator const char *()**const** {**return** m_pszText;}

После слова *operator* следует определение типа.
В данном случае это *const char ** - константный указатель на символ (*z*-строка).

Теперь объекты *Str* можно применять везде, где допускается использовать *z*-строку!

Конструктор инициализации для Str*

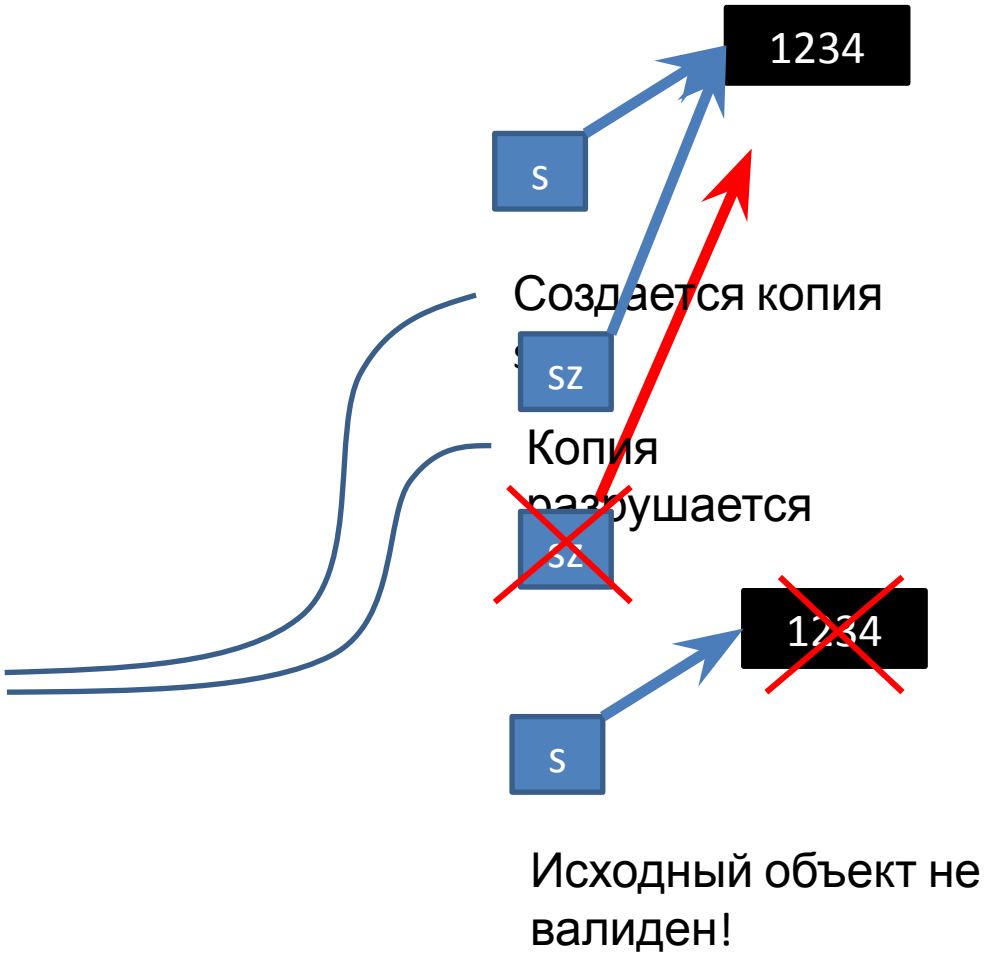
```
Str (const char * p)
{
    if(p) {
        m_pszText = new char [strlen(p) + 1];
        strcpy_s (m_pszText, strlen(p) + 1, p);
    }
    else
        m_pszText = new char[1]{0};
}
```

- Теперь можно:
  Str s = “abc”;

Конструкторы с непустым списком параметров – ***конструкторы инициализации***. Таких конструкторов может быть много.

Варианты использования Str
Неправильное копирование*

```
void Test1 (Str sz)
{}

Str Test2 () {
      Str sz= “123”;
      return sz;
}

int main(){
       Str s = “1234”;
       Test1(s);
       Str t = Test2();
}
```



Конструктор копирования*

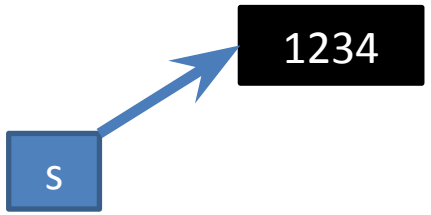
```
Str (const Str &s) //Константная ссылка на объект
{
      m_pszText = new char
[strlen(s.m_pszText) + 1];
      strcpy_s (m_pszText,
                  strlen(s.m_pszText) + 1,
                  s.m_pszText);
}
```

Конструктор, у которого есть единственный параметр – константная ссылка на объект того же типа, называется ***конструктором копирования***.

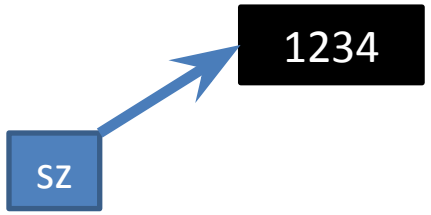
Он, если задан, будет вызываться во всех случаях создания копии объекта.

Создание копии*

Теперь есть правила Копирования



В результате получаем:



“Правильные” копии можно создавать и использовать без ограничений!

Конструктор копирования
Ловушка

Если нет конструктора копирования, то копирование заключается в копировании памяти (*dummy constructor*).

При этом, все переменные типа *int, double* … **автоматически** приобретут в копии то же значение, что и в оригинале!

Но, …!

если вы объявили собственный конструктор копирования, то это правило не действует! При копировании будут выполнены **ТОЛЬКО ТЕ ДЕЙСТВИЯ, КОТОРЫЕ УКАЗАНЫ В КОНСТРУКТОРЕ КОПИИ**!

Конструктор копирования
Ловушка. Пример.

```
class A{
        public:
                int Val;
                A(int v=0) {Val=v;}
                operator int&() const {return Val;}
};
A a(1), b(a);
cout<<a<<b<<endl;  // Увидим 11
```

Добавим конструктор копии:

```
                A(const A &a) {}
```

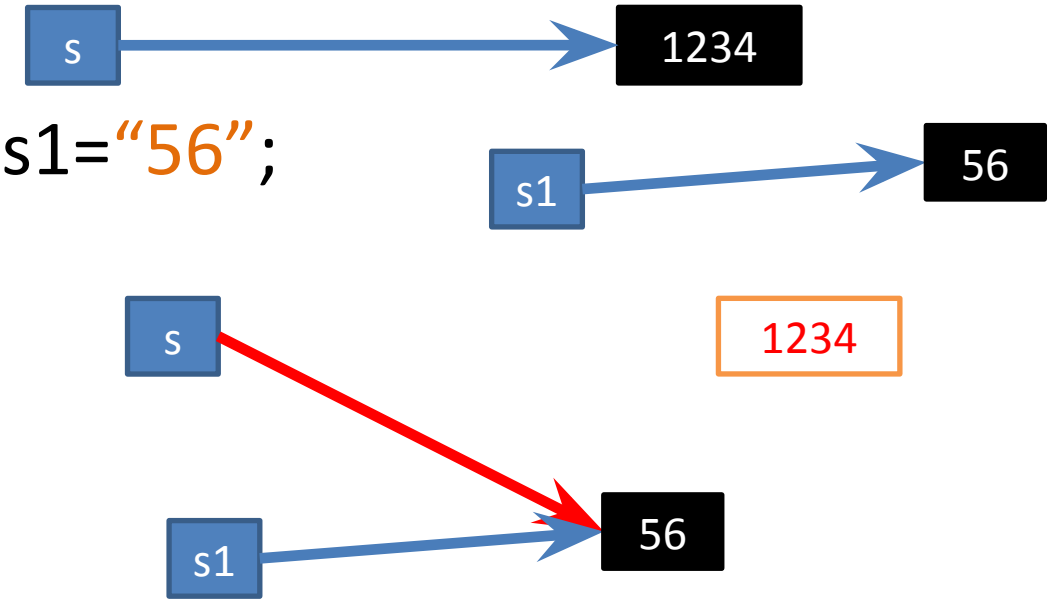
```
A a(1), b(a);
cout<<a<<b<<endl;  // Теперь увидим 1<какой-то мусор>
```

Во втором случае память не копировалась. Нужно обязательно в конструкторе копии

A(const A &a): Val(a.Val) {} или **A(const A &a) {Val = a.Val;}**

Варианты использования Str
Неправильное присваивание*

```
int main(){
    Str s = “1234”, s1=“56”;
    s = s1;
}
```



В результате присваивания память **s1** была скопирована в **s**:

- Блок “1234” остался висеть;
- Оба объекта “владеют” одним блоком.

Операция присваивания*

```
const Str& operator = (const Str &s) //Константная ссылка на объект
{
	if(&s == this) return *this;
	delete [] m_pszText;	//Освобождаем текущие данные
		//Дальше просто копирование
	m_pszText = new char [strlen(s.m_pszText) + 1];
	strcpy_s (m_pszText,
			strlen(s.m_pszText) + 1,
			s.m_pszText);
	return *this;
}
```

Теперь объекты класса Str можно без проблем присваивать друг другу.

Еще о присваивании*

*Имеется конструктор инициализации Str (**const char*** sz). Можем инициализировать объекты z-строкой/*

Это удобно!

Можно ли присвоить уже созданному объекту z-строку (s = “911”)?

Да, можно! При этом, компилятор:

1. Создаст временный объект типа Str с начальным значением, равным z-строке;
2. Выполнить присваивание (есть оператор присваивания);
3. и разрушит временный объект.

С объектами ВСЕ ПРОСТО!

Когда создаются копии объектов, а когда нет? Каково время жизни объектов? Что происходит при присваивании? …

На многие вопросы можно ответить следуя правилу:

“***Объекты классов ведут себя так же как и переменные встроенных типов***”.

Время жизни определяется теми же правилами, что и для встроенных типов. Передача параметров, возникновение копий, определяется теми же правилами и т.п.

Промежуточный итог

Узнали:

- Виды конструкторов и зачем они нужны.
  Конструктор копии. Нужен, если в классе используются динамические поля;

- Как переопределить операцию преобразования типа;
  operator <тип> const()…

- Разобрались, что происходит при присваивании;
  по умолчанию копируется память одного объекта в другой. Компилятор ищет возможность выполнить операцию;

- Узнали как понять, что происходит с объектом.
  - “А что происходило бы с переменной встроенного типа?”;

Контрольные вопросы

1. Перечислите виды конструкторов применяемых в C++.
2. Почему для Date не нужен деструктор и конструктор копии, а для Str нужны?
3. Почему в конструктор копии передают ссылку на объект, а не копию объекта?
4. Зачем при передаче ссылок и указателей применяется слово **const**?
5. Приведите примеры, когда копии объектов возникают без явного вызова конструктора.
6. Что в C++ методе означает слово this? Чем оно отличается от this в C#?
7. Что произойдет, если выполнить следующий код:
   Str s = “abcd”;
   s = “1234”;

Имеется соответствующий конструктор инициализации и оператор присваивания объектов Str.

Ссылки и объекты в C#*

В C# Reference Type (*Value Types похожи на C++*) это:

- **Всегда ссылка**, но !
- Ссылка в .Net – *это самостоятельная переменная*!
- При копировании копируется ссылка, а не объект! Копирование объектов в .Net довольно сложная задача!

Ссылки и объекты в C# Сравнение понятий

C++	C# (.Net)
Ссылка не может не ссылаться на объект (это не самостоятельная переменная, а просто другое имя объекта) После объявления ссылки нельзя изменить объект, на который она указывает.	Ссылка является самостоятельной сущностью. Есть область видимости и область существования. Может никуда не ссылаться или ссылаться на разные объекты в течение времени жизни.
При передаче параметра по ссылке (&) передается адрес объекта, на который сослались при вызове метода. *Но работаем со ссылкой как с обычной переменной.*	При передачи в качестве параметра ссылка копируется! Что бы передать ссылку на ссылку используется квалификатор ref.
О наличии ссылок на объект узнать в программе нельзя!	За наличием ссылок на объект следит сборщик мусора (Garbage Collector)

Строки стандартной библиотеки

```
#include <iostream>
#include <string> // Появился тип string.

using namespace std;

string s;
cin >> s;     // Читает слово
cout << s.length() << endl;
cout << s[0] << endl;
cout << (s + s) << endl;
…
```

Строки стандартной библиотеки и консольный ввод

getline(cin, s); **//Читает всю строку!** ~~**cin>>s**~~

```
while (!cin.eof())        // Закончится по Ctrl+z
{
      cout <<s.size() << ' ' << s << endl;
      cin >> s;
}
cout << "end of input" << endl;
cin.clear();       // Очистит Ctrl+z
cin >> s;          // Теперь можно снова читать cin
```

Более полно будет рассмотрено в дальнейшем, при изучении ***stl***.

Контрольные вопросы
Строки и ввод с консоли

1. Опишите, чем понятие ссылки в .Net отличается от ссылки в C++.
2. Какой класс стандартной библиотеки используется для работы со строками?
3. Как прочесть слово с консоли?
4. Как прочесть с консоли строку целиком?
5. Как читать с консоли в цикле? Как завершить ввод?
6. Как продолжить ввод с консоли после ввода конца файла?