САОД (stl)

А. Задорожный

2017

Контрольные вопросы

1. Для каких целей применяются объекты, учитывающие ссылки?

2. Можно ли сказать, что в .Net применяются объекты, учитывающие ссылки?

3. Что позволяют описывать шаблоны в C++?

4. Для чего нужен алгоритм Бойера-Мура?

Рассмотрены

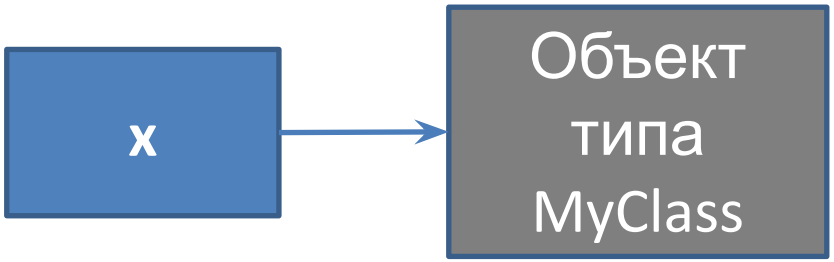
- Value и Reference типы, ссылки в .Net
- STL
  - Потоки
  - Строки
  - Общие свойства контейнеров
  - Последовательные контейнеры:
    - Список
    - Динамический массив
    - Очереди и стеки
  - Ассоциативные контейнеры:
    - Множества
    - Словари (map)
- Примеры применения контейнеров
- Умные указатели

Value и Reference типы в .Net

- **Типы-значения**: struct, enum. В частности, int, double, bool и char.

- Остальные типы ссылочные.

MyClass x = **new** MyClass();

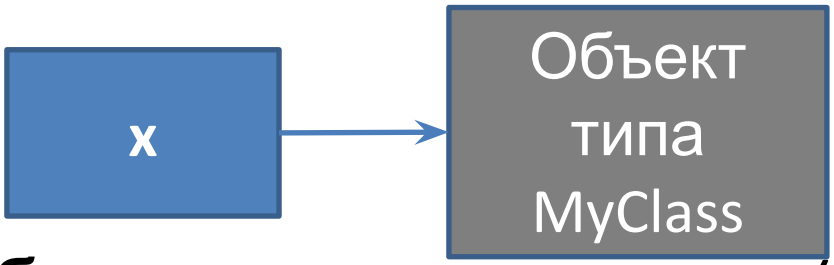


Различия при копировании и присваивании.

Value и Reference типы в .Net

- Остальные типы ссылочные.

MyClass x = **new** MyClass();



Объект не имеет имени (как и объекты в C++, создаваемые **new**).

Имя у ссылки!

Value и Reference типы в .Net

```
MyStruct s1 = new MyStruct { Name = "123"}, s2 = s1;
Console.WriteLine("{0} {1}", s1.Name, s2.Name);
s2.Name = "456";
Console.WriteLine("{0} {1}", s1.Name, s2.Name);
```

```
123 123
123 456
```

```
MyClass c1 = new MyClass { Name = "123"}, c2 = c1;
Console.WriteLine("{0} {1}", c1.Name, c2.Name);
c2.Name = "456";
Console.WriteLine("{0} {1}", c1.Name, c2.Name);
```

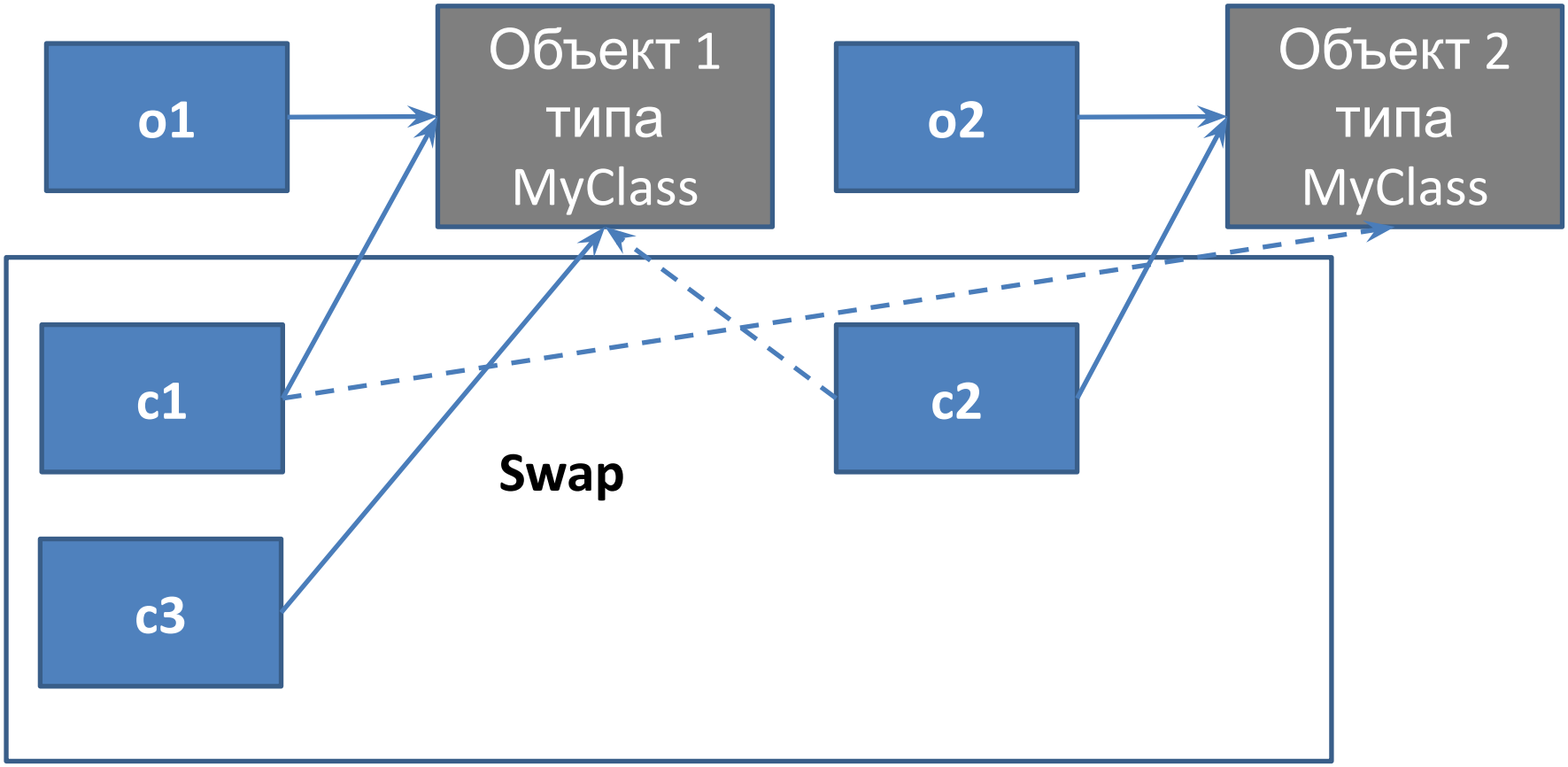
```
123 123
456 456
```

Ссылки и передача параметров в .Net

```
void Swap(MyClass c1, MyClass c2)
{
    MyClass c3 = c1;
    c1 = c2;
    c2 = c3;
}
```

Хотя в метод переданы ссылки, обмена значениями не произойдет!

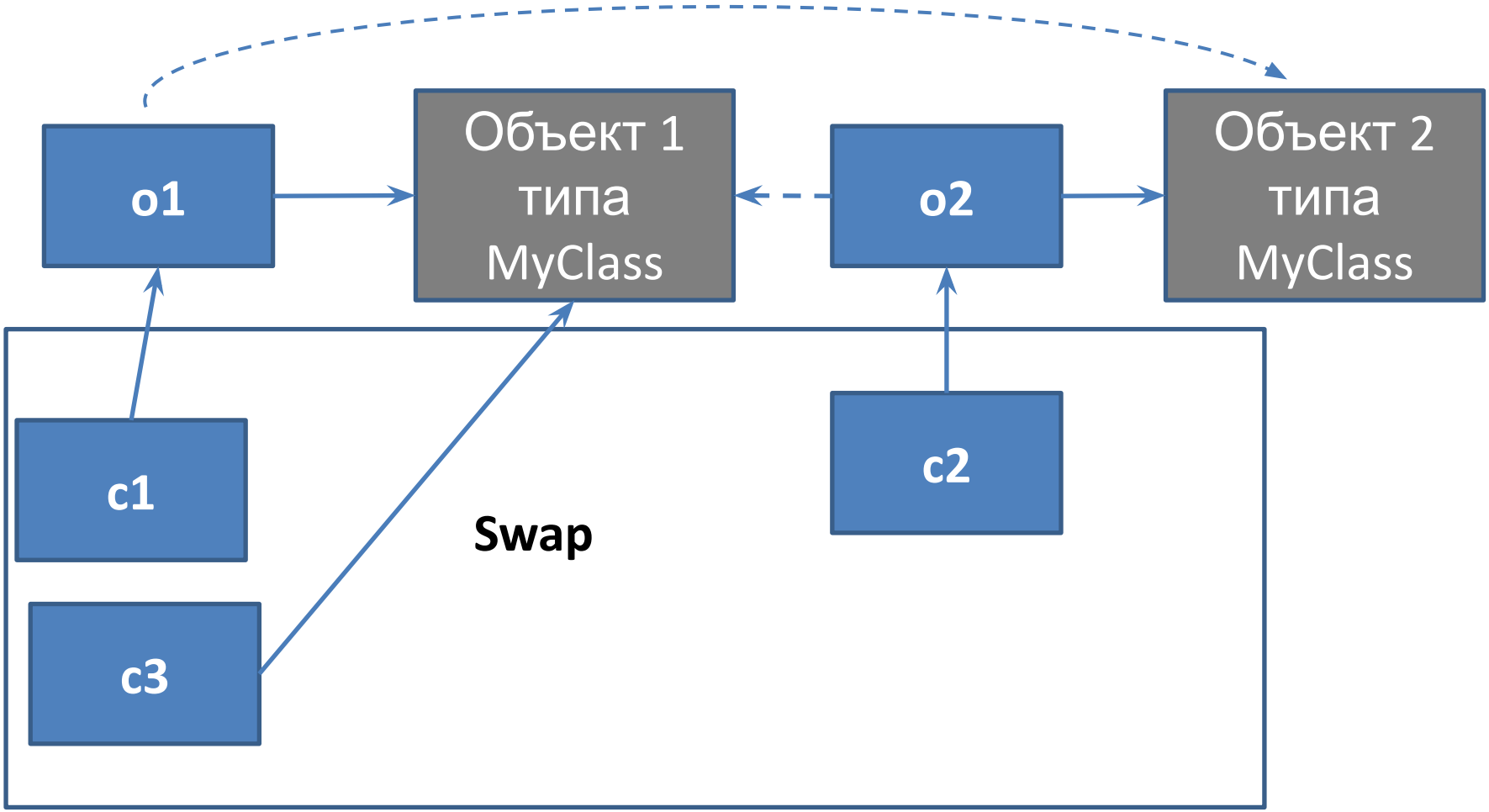
Ссылки и передача параметров в .Net



Ссылки и передача параметров в .Net

```
void Swap(ref MyClass c1, ref MyClass c2)
{
    MyClass c3 = c1;
    c1 = c2;
    c2 = c3;
}
```

Ссылки и передача параметров в .Net



Ссылочные типы и строки .Net

Интересной особенностью обладает тип string.

String в .Net – неизменяемый ссылочный тип!

Т.е., если нужно изменить строку в .Net, то необходимо построить новую строку с нужным значением.

String s = “123”;
s += “456”

После операции += создана новая строка, на которую и указывает ссылка s. Исходная строка оказалась неиспользуемой!

Упражнения

1. Предложите проверку, которая покажет, является ли объект экземпляром ссылочного типа или экземпляром типа-значения.

2. Как убедиться, что String – ссылочный тип?

STL – стандартная библиотека (шаблонов)

STL - **S**tandard **T**emplate **L**ibrary

Набор абстрактных типов данных и алгоритмов.

Основное содержание в заголовочных файлах.

Для использования модуля библиотеки нужно подключить соответствующий заголовок.

Файлы не имеют расширения.

Потоки (STL)

- Обобщенный способ организации ввода/вывода.
  - Потоки ввода (istream),
  - Потоки вывода (ostream),
  - И потоки ввода-вывода (iostream).

Моды потоков

- Потоки открываются в бинарной и текстовой моде
- В текстовой моде можно управлять форматированием. Например, представлением целых чисел.

```
cout << hex << i << endl;   //Шестнадцатиричный вывод
dec( cout );          // десятеричный по умолчанию
cout << i << endl;
oct( cout );          // восьмеричный по умолчанию
cout << i << endl;
cout << dec << i << endl; // десятеричный 1 раз
```

Специальные потоки

- потоки **cout**, **cin**, **cerr**, **clog** связаны только со стандартным вводом выводом и применимы только в консольных приложениях.

- Остальные потоки не имеют ограничений. Для работы с файловыми потоками нужно использовать заголовок

```
#include <fstream>
```

Файловые потоки. Пример

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
	char fName[128];
	cout << "File name ? : "; cin.getline(fName, 127);
	cout << fName << endl;

	ifstream ifs;
	ifs.open(fName);	// (fName, ios_base::binary);
	if (ifs.is_open())
	{
		char tmp[1024];
		ifs >> tmp;
		cout << tmp << endl;
		ifs.close();
	}
	else
	 cout << fName << " - does not exist" << endl;
      return 0;
}
```

Строки.

Стандартная библиотека предоставляет класс работы со строками – заголовок <string>.

Теперь вместо:

```
char fName[128];
cout << "File name ? : "; cin.getline(fName, 127);
cout << fName << endl;
```

можно использовать:

```
string fName;
cout << "File name ? : "; cin >> fName;
cout << fName << endl;
```

Строки. Минимальный набор.

- **Конструкторы**: string(), string (const char *p) и string(const string &s).
- **Операции**: length(), [], =, +, +=, <, >, …
- string **substr(**int Offset, int Length)
- int **find**(const string &s, int Offset) - *возвращает индекс ближайшего появления подстроки начиная с Offset и **-1** если нет.*

Всего несколько десятков методов.

Контейнеры в STL

Важная часть STL – шаблоны контейнеров.

Контейнеры – класс для хранения объектов других классов. Реализуют все базовые структуры данных:

deque (queue, stack**),** - очередь с двумя концами
list, - связный список
vector (*priority_queue*) – динамический массив
hash_map, hash_set, *hash_multimap, hash_multiset*
map, set, *multimap, multiset*

Общие свойства контейнеров

- Контейнеры STL можно параметризовать любыми типами, для которых определены операции =, == и <.
- Создание копии при включении в контейнер.
- Универсальный способ для доступа к элементам любого контейнера – итераторы.

Итераторы

Итераторы - типы, позволяющие двигаться по контейнеру

- <TYPE NAME>::iterator i = obj.begin();
- <TYPE NAME>::iterator i = obj.end();

Метафора итератора – указатель.

```
for(<TYPE NAME>::iterator i = obj.begin();
                i != obj.end();              i++)
{
// Делаем что хотели с элементами,
//используя *i;
}
```

Начиная с C++11 можно:

```
for (auto &v : obj)
{
// Делаем что хотели с элементами,
//используя v;
}
```

Итераторы и .Net

- Как правило, использование итераторов позволяет более эффективно пройти по всему контейнеру, чем другие способы;

- Аналогией в .Net является применение оператора foreach

```
foreach(var p in Lst)
{
    //… обработка элемента контейнера p
}
```

Последовательные контейнеры. Список.

```
typedef list<string> LSTSTR;
 //Создание
LSTSTR lst1, lst2(5, “abc”);
LSTSTR lst3(lst2), lst4(lst2.begin(), --lst2.end());

//Проверка на пустоту
cout << lst1.empty() << endl; //true

//Количество элементов
cout << lst2.size() << endl; //5
```

Список

```
                                    //Добавление элементов
lst1.push_back(“2”); lst1.push_front(“1”); // {1,2}
lst1.insert(--lst1.end(), “a”);          // list is {1,a,2}
cout << lst1.size() << endl;             // 3
                                    //Изменить элемент
*lst1.begin() = “3”;                 // {3,a,2}
                                    //Получить первый/последний
cout << lst1.front() << endl;            //3
cout << lst1.back() << endl;             //2
                                    //Присваивание
lst2 = lst1;
                                    //Удаление элементов
lst1.remove(“a”);                    // {3,2}
lst1.erase(lst1.begin())             // {2}
lst1.erase(lst1.end())               // empty
                          //Отсортировать список можно методом sort.
lst2.sort()                  // {2, 3, a}
```

Последовательные контейнеры. Динамический массив.

```
typedef vector<int> VINT;
                                        //Создание
VINT v1, v2(100);
VINT v3(v2.begin(), --v2.end());
                                        //Присваивание
v3 = v1;
                                        //Доступ к элементам
v2[i] = 10;

v2.push_back(11);          // добавляет элемент в конец массива
cout<< v2.size()<<endl;    //  101

                                        // Можно и v2.insert(x, v2.begin());
```

Отсортировать массив можно функцией STL sort.

```
                                        //sort vector
sort(v.begin(), v.end());
```

Последовательные контейнеры. Динамический массив.

В дополнение к списку vector (как и string) имеет 2 характеристики размера:

- size() – количество элементов;
- capacity () – количество элементов, который может включать без расширения памяти;

Очевидно, **size()<=capacity()**!

Последовательные контейнеры. Очереди и стеки.

<deque> - *Double ended queue*, двусторонняя очередь (сокращенно *Дек*).

Позволяет помещать и получать объекты в порядке поступления как в начало, так и в конец.

Основные операции:

push_back(<…>), push_front (<…>)

<…> pop_back(), <…> pop_front ()

Ясно, что ограничивая функционал дека можно прийти к очереди (*queue*) и стеку (*stack*).

Ассоциативные контейнеры. Множество.

```
typedef set<string> SETSTR;
                                        //Создание
SETSTR s, s2;
                                        //Пустой
cout << s.empty() << endl;          //true
                                //Добавление элементов
s.insert (“abc”); s.insert (“123”);      // s.size() == 2
s2 = s;                             // s2 == s
                                        //Поиск элемента
SETSTR::iterator i = s.find(“abc”);   //i != s.end(), *i == “abc”
i = s.find(“efd”);                      //i == s.end()
                                //Удаление элементов
s.erase (“abc”); s.erase(s.begin());   // s is empty
```

Ассоциативные контейнеры. Таблицы.

```
typedef map<string, int> STR2INT;
                                        //Создание
STR2INT m;
                                        //Пустой
cout << m.empty() << endl;          //true

                                    //Добавить и изменить
m.insert(STR2INT::value_type("a",1));    // m.size() == 1
m["b"] = 2;                         // m.size() == 2
m["a"] = 3;                         // m.size() == 2

                                        //Поиск
STR2INT::iterator i = m.find("a");    // i != m.end(),
                                    // (*i).first == "a"
                                    // (*i).second == 3
m["b"] == 2;
```

Таблицы

```
STR2INT::iterator i = m.find(“ab”);    //i == m.end()

//Содержат пары!
for (STR2INT::iterator i= m.begin(); i != m.end(); i++)
    cout << i->first << “\t” << i->second << endl;

//Удаление элементов
m.erase(“a”); m.erase(m.begin()); // m is empty
```

Пары

pair – удобная абстракция <*ключ, значение*>

```
pair<int, string> pr;  //ключом является int, значение – string.
pr.first = 1;      pr.second = “123”;
```

Пары можно копировать, присваивать, сравнивать (сравнивается ключ и значение!)*

Прочие шаблоны

- Иногда полезно иметь контейнер для хранения нескольких элементов с одинаковым ключом.
- **multiset** и **multimap**.
- Определены эти multi-контейнеры в тех же заголовочных файлах <set> и <map>.

Методы контейнеров практически совпадают с set и map, за исключением того, что для multimap не определен оператор [] – выбора элемента по ключу.

- STL имеет набор стандартных алгоритмов. Они описаны в заголовочном файле <**algorithm**>. (алгоритмы операций с контейнерами для их заполнения, копирования, поиска элементов или пар элементов, сортировки, изменения порядка на обратный, замены одних элементов на другие и пр.)

В частности, метод sort, который применялся для вектора, расположен именно в <**algorithm**>.

Аналоги контейнеров в .Net

- List<> - динамический массив, аналог vector
- Dictionary<,> - словарь, таблица, аналог map

- Аналог set в расширениях, например HashSet<>

Зачем столько контейнеров?

1. Сложность изменения списка O(1), а чтения элемента с заданным номером O(N);

2. Сложность изменения массива O(N), а сложность чтения элемента с заданным номером O(1);

3. Сложность изменения и чтения ассоциативных контейнеров O(Ln(N)).

Пример применения контейнеров

Задача. Посчитать количество различных слов, которые поступают с консоли в программу.

Предположим, что все они в верхнем регистре, т.е. “Hello” и “HELLO” будут поступать как “HELLO”.

```
typedef set<string> SETSTR;
SETSTR c;          //Объявили контейнер
string s;
cin>>s;            //Читаем слово

while (s != “”){          //пока слово не пусто
        c.insert(s);
         cin>>s;          //Читаем следующее слово
}
cout << c.size() << endl;
```

Пример применения set

Если нужно вывести все различные слова, то

```
for (SETSTR:: iterator i = c.begin(); i != c.end(); i++)
    cout << *i << endl;
```

Примеры применения контейнеров.

Задача. Найти N самых частых слов в тексте.

Задачу можно решить в 2 этапа:

1. Посчитать сколько раз каждое слово встречалось в тексте и
2. Найти N наиболее частых.

Примеры применения контейнеров.

```
typedef map<string, int> STR2INT;

STR2INT m;        //Объявили контейнер
string s
cin>>s;           //Читаем слово*

while (s != “”){  //пока слово не пусто
      m[s]++; // вставить, если не было и увеличить счетчик
       cin>>s;       //Читаем следующее слово
}

cout << m.size() << endl; //количество различных слов
```

Примеры применения контейнеров

```
int N=16;
set<pair<int, string>> mostFriquent;

for (STR2INT:: iterator i = m.begin(); i != m.end(); i++)
{
      mostFriquent.insert(pair(i->second, i->first));
      if(mostFriquent.size() > N)
            mostFriquent.erase(mostFriquent.begin());
}

for (set<pair<int, string>>:: iterator i = mostFriquent.begin();
                        i != mostFriquent.end(); i++)
cout << i->second << “\t” << i->first << endl;
```

Контрольные вопросы

1. Почему контейнеры называются абстрактными типами данных?
2. Что такое последовательные и ассоциативные контейнеры? Приведите примеры из STL.
3. Что такое итератор в STL?
4. Какие операции должны быть определены для объектов, вставляемых в контейнеры STL?
5. Если в программе используется глобальный контейнер, а в него помещается локальный объект, не нарушится ли контейнер, когда локальный объект будет разрушен? Почему?
6. Дайте рекомендации, для каких целей хорошо использовать каждый из контейнеров.

Умные указатели (smart pointers)

Одна из распространенных ошибок в C++ связана с *new* ... Нет соответствующего *delete*

```
CTest * p = new CTest;
```

Smart pointers противостоят этой проблеме!

В STL это **shared_ptr<>**. Реализованы они в модуле **<memory>**.

```
shared_ptr<CTest> pt(new CTest);
```

Умные указатели (smart pointers)

```
shared_ptr<CTest> pt(new CTest);
```

pt – *это* ***объект*** *типа shared_ptr<CTest> , проинициализированный указателем на созданный объект класса CTest*.

Его можно использовать как указатель на динамически созданный объект!

```
cout << pt->val << endl;
```

Когда исчезнет последний умный указатель на динамически созданный объект, сам объект будет разрушен!

Умные указатели
Основные операции

shared_ptr<CTest> pt(new CTest);

Конструкторы копирования и инициализации,

= - *присваивание,*

->, * - доступ к членам CTest и разименование;
Имеются и операции сравнения ==, >= , …

bool – *преобразование к булевскому. 1, если инициализирован и 0, если нет!*

int use_count() – сколько объектов ссылается на управляемый объект

Умные указатели демонстрация

```
shared_ptr<CTest> pt(new CTest);
cout << (bool)pt << endl;                  // 1
cout << pt.use_count() << endl;                 // 1
cout << (*pt).val << pt->val << endl;  // 00

shared_ptr<CTest> ps;
cout << (bool)ps << endl;                  // 0
cout << pt.use_count() << endl;                 // 0
cout << (*ps).val << ps->val << endl; // ошибка

ps = pt;

cout << (bool)ps << endl;                  // 1
cout << pt.use_count() << endl;                 // 2
cout << (*pt).val << pt->val << endl;  // 00
```

Умные указатели
Без new

Для *shared_ptr* реализована и “фабрика класса”, функция, которая позволяет создавать объекты этого типа и вообще отказаться от использования оператора ***new***.

shared_ptr<CTest> sp = **make_shared <CTest>**(CTest());

Работает чуть более эффективно и исключает утечки памяти даже в случае ошибок.

Рекомендуется именно так!

Умные указатели
Заключительные замечания

Умные указатели реализуют подсчет ссылок на объект владения.

Появились в такой форме в C++11. Стали официальным стандартом. Все еще подвижны и развиваются.

Терминология. Умные указатели реализуют идиому **R**esource **A**cquisition **I**s **I**nitialization (**RAII**) – получение ресурса – это и его инициализация.

Контрольные вопросы

1. Что означает термин “умные указатели” (smart pointers)? Зачем они введены в STL?

2. Как называется в stl класс умных указателей? Следует ли создавать объекты умных указателей динамически?

3. Назовите основные операции над умными указателями?