Шаблоны классов

```
//list.h
template <typename T>
class List{
public:
    class ListElement{
    public:
        T& operator * ();
        const T& operator * () const;
        ListElement& next ();
        const ListElement& next () const;
    };
    ListElement& head ();
    const ListElement& head () const;
    void addHead (const T& val);
};
```

Реализация шаблонов классов

```
//list.h !!!

template <typename T>
void List<T>::addHead (const T& val){
        //…
}

template <typename T>
List<T>::ListElement& List<T>::ListElement::next (){
        //…
}
```

Специализация шаблонов

```
template <>
List<void*>::ListElementList& <void*>::ListElement::next (){
        //...
}

template <typename T>
List<void*>::ListElementList& List<T*>::ListElement::next (){
        //...
}
```

Использование шаблонов классов

```
int main (){
        List<Complex> list;
        list.addNode (1); //list.addNode (Complex (1, 0));
        List<Complex>::ListElement el;
        el = list.head ();
        List<double> dlist;
        el = dlist.head (); //ошибка
        return 0;
}
```

Метаклассы

Инстанциирование шаблона – создание класса (функции, методы) при подстановке параметров.

Инстанциирование класса – создание объекта путем вызова конструктора с конкретными параметрами.

Шаблонный класс является метаклассом: класс, экземпляры которого являются также классами.

Инстанциирование шаблонов

Шаблон класса с подставленными параметрами является классом.

Отдельный класс генерируется для каждого набора параметров и при любом обращении к нему.
Класс генерируется на стадии компиляции.

```
typedef List <Complex> ComplexList; /*достаточно для генерации класса*/
```

Один о тот же шаблон с разными параметрами – это разные типы

```
List <char> *list = new List <unsigned char>; //ошибка
```

Компиляция шаблонов

- Шаблоны не компилируются
- Инстанциирование шаблонов происходит на стадии компиляции
- Инстанциирование шаблонов – это генерация классов с подставленными параметрами
- Все определение шаблона должно находиться в заголовочном файле.

Standard Template Library

- Контейнеры (basic_string, vector, list, map …)
- Итераторы
- Потоки (basic_istream, basic_ostream)
- Алгоритмы (sort, count, find, for_each …)
- Численные методы (valarray …)

Списки

- Реализует функциональность списка, очереди и стека
- Совместим с другими контейнерами (через итераторы)

```
template <typename T>
class list{
public:
        class iterator{/*...*/};
        class const_iterator{/*...*/};
        class reverse_iterator{/*...*/};
        class const_reverse_iterator{/*...*/};
//...
};
```

Использование списка

```
#include <list>
using namespace std;
int main (void){
        list <char> cList;
        cList.push_front ('a');
        cList.push_back ('c');
        cList.insert (++(cList.begin ()), 'b');
        list<char>::const_iterator it;
        for (it = cList.begin (); it != cList.end (); ++it)
                cout << *it;
        cout << endl;
        return 0;
}
```

```
abc
```

Операции со списком без использования итераторов

- Конструирование, копирование
- Деструктор
- Доступ к крайним элементам (front, back, push_front, pop_front) – работа в режиме двунаправленной очереди
- Получение количества элементов (size () const)
- Очистка (clear ())

Итераторы

Итератор – объект который ведет себя как указатель и позволяет посредством своих методов обходить все элементы контейнера. Может использоваться, как идентификатор элемента.

В STL от итератора обладает следующими методами:

- T& operator *
- T* operator -> ()
- iterator operator ++ ()
- iterator operator -- ()

Операции со списком с использованием итераторов

- iterator begin (); //итератор первого элемента
- const_iterator begin () const;
- iterator end (); //итератор барьера
- reverse_iterator rbegin ();

//всего 8 методов

- insert (iterator where, T elem); //вставка перед where

- erase (iterator where);

- erase (iterator begin, iterator end);

Совместимость с другими контейнерами

```
template <typenate T>
class list{
//...
public:
        template class <InputIter>
        insert (iterator where, InputIter first, InputIter last);
//...
};
```

Массивы

- Реализует функциональность массива, расширяемого с конца, а также стека

- Совместим с другими контейнерами (через итераторы)

```
template <typename T>
class vector{
public:
        class iterator{/*...*/};
        class const_iterator{/*...*/};
        class reverse_iterator{/*...*/};
        class const_reverse_iterator{/*...*/};
//...
};
```

Отличия от списка

- Не определены методы работы с началом (push_front)
- Определен operator []
- Для итераторов определены операторы
operator +
operator += //…
- Для добавления в конец массива используется стратегия резервирования места.

Стратегия резервирования места в массиве

- Вектор всегда резервирует памяти под большее число элементов, чем в нем есть

- Если происходит вставка в конец, и есть резервное место, то память не реалоцируется.

- Если происходит вставка в конец и нет резервного места, то память выделяется с запасом.

- Существует возможность вручную управлять этим механизмом с помощью следующих методов:

```
void reserve (size_type new size);
size_type capacity () const;
```

Строки

- Поддерживаются все основные операции со строками
- Символы могут иметь произвольный тип
- Совместимы с C-строками (char *)

```
template <typename T>
class basic_string{
//...
};
typedef basic_string <char> string;
typedef basic_string <wchar_t> wstring;
```

Пример использования строк

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

int main (void){
        string str = “Hello, Vasya!”;
        string oldName = Vasya;
        str.replace (str.find (oldName), oldName.length (), “Petya”);
        cout << str << endl;
        return 0;
}
```

```
Hello, Petya!
```

Операции со строками

- Сравнение (лексикографическое – operator >, < ==…)
- Конкатенация (operator +)
- Замена, поиск.
- Присваивание, конструирование (в том числе из С-сроки)
- Преобразование в С-строку (c_str () const).
- Поиск, замена и удаление части строки
- Определение длины (length () const, size () const)

Поиск, замена и удаление

```
size_type find (string &) const;
//string::npos – подстрока не найдена

insert (size_type, string &);

replace (size_type, size_type, string&);

erase (size_type pos = 0, size_type num = string::npos);
```

Технология copy-on-write

```
class String{
private:
        struct StrContainer{
                char * str;
                int size;
                int refCnt;
        } * payload;
        void clone ();
public:
        String& operator = (const String& arg){
                payload->refCnt--;
                if (payload->refCnt == 0) delete payload;
                payload = arg->payload;
                payload->refCnt++;
        }
};
```