

Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

18 октября 2017 г.

Оглавление

7	Контейнеры в C++	3
7.1	LinkedList	3
7.2	Библиотека стандартных шаблонов C++	4

Лекция 7

Контейнеры в C++

7.1 LinkedList

Продолжим работу над кодом из предыдущей лекции:

Код 7.1: LinkedList

```
1 #include <iostream>
2 #include <string>
3 #include "LinkedList.hpp"
4
5 template<typename DataType>
6 struct tNode
7 {
8     DataType data;
9     tNode* next = nullptr;
10 };
11
12 template<typename DataType>
13 tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data)
14 {
15     tNode<DataType> *p = new tNode<DataType>;
16     p->data = data;
17     p->next = p_begin;
18     return p;
19 }
20
21 int main()
22 {
23     tNode<std::string> *p_begin = nullptr;
24
25     p_begin = insert_node(p_begin, std::string("Hello"));
26     p_begin = insert_node(p_begin, std::string("World"));
27     return 0;
28 }
```

Код 7.2: Header-файл для нашей программы

```
1 // Added module
2
3 template<typename DataType>
4 struct tNode
5 {
6     DataType data;
7     tNode* next;
8 };
9
10 template<typename DataType>
11 struct List {
12     tNode<DataType> *begin;
13 }
14
15 template<typename DataType>
16 void list_init(List<T>);
17
18 template<typename DataType>
19 void list_insert(List<T>, const T &value);
20
21 template<typename DataType>
22 void list_print(List<T>);
23
24 template<typename DataType>
25 void list_destroy(List<T>);
26
27 template<typename DataType>
28 tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data);
```

Код заполнен не до конца, я не смог поспевать за Т.Ф. (если у кого будут исправления, прошу написать [мне](#)).

7.2 Библиотека стандартных шаблонов C++

Библиотека стандартных шаблонов C++ называется **STL**. Она состоит из нескольких частей и типов, но выдерживает следующую структуру:

1. Контейнеры – некоторый объект, содержащий данные.
2. Итераторы – позволяют *итерироваться по элементам контейнера*.
3. Алгоритмы – название само говорит о предназначении. Примечательно, что они не имеют привязки к конкретному типу контейнера.

Таблица (для наглядности)

Контейнеры	Итераторы	Алгоритмы
Типы контейнеров: 1. vector<T> (похож на list в Питоне) 2. list<T> (это LinkedList из Питона) 3. set<T> – красное-черное двоичное дерево поиска ($O(\log N)$) - добавление/удаление/поиск. 4. unordered_set<T> – хэш-таблица $O(1)$ – добавление/удаление/поиск 5. map<T, V> – ассоциативный массив, состоящий из пар $\langle key, value \rangle$. При этом key типа T , $value$ типа V	Типы итераторов: 1. <u>input output</u> – доступ к элементу содержимого контейнера по итератору 2. <u>bidirectional</u> $--i$ – если контейнер 'знает', что он двунаправленный 3. <u>random access</u> $i = i + 100$ – итератор произвольного доступа (а не только по $+1$) 4. <u>forward</u> $++i$ – обычный шаг на один вперед	1. см. ниже

Итерирование

Попробуем проитерироваться по контейнеру и применить к каждому элементу функцию **foobar**. Это можно сделать несколькими способами:

Код 7.3: Пример итерирования

```

1. 1 vector<int> A = {10, 3, 17, 28};
   2
   3 for (vector<int>::iterator i = A.begin(); i != A.end(); ++i)
   4 {
   5     foobar(*i);
   6 }
```

Здесь итератор **i** пробегает по всем значениям массива. Как он ведет учет текущей позиции, нас не волнует – в этом и есть прелесть подобной структуры. Начало итерирования – начало массива, конец итерирования – конец массива. Есть и защита: если попытаться выйти за `A.end()` (например, на единицу), то при попытке разыменовать такой адрес выйдет ошибка.

2. То же самое итерирование можно было провести с помощью **foreach** (из библиотеки `<algorithm>`)

Код 7.4: Применение алгоритмов для итерирования

```
1 #include <algorithm>
2
3 int main()
4 {
5     vector<int> A = {10, 9, 15, 7, 20};
6     for_each(A.begin(), A.end(), foobar);
7 }
```

Обратите внимание: функции **for_each** уходит *указатель* на функцию **foobar** (она ведь тоже хранится в памяти, как и переменные).

В стандарте C++ 11 использование контейнеров упрощено:

Код 7.5: Контейнеры в C++11

```
1 #include <algorithm>
2
3 int main()
4 {
5     vector<int> A = {10, 9, 15, 7, 20};
6     //for_each(A.begin(), A.end(), foobar);
7     for (auto x : A) {
8         x = x*2
9         std::cout << x;
10    }
11 }
```

Единственный минус: чтобы понять, что происходит в итераторе, приходится копать код вверх, до описания функции **foobar**.

В примере 7.5 используется **auto** – автоматический выбор типа. При таком подходе компилятор сам подбирает подходящий тип для переменной.