Списки, итераторы и лямбда функторы.

Шокуров Антон В.
shokurov.anton.v@yandex.ru
http://машинноезрение.рф

6 октября 2018 г.

Версия: 0.10

Аннотация

В данной заметке будут рассмотрены ещё объектов из библиотеки `stl`. Данный объект обобщает процесс итерации по элементам массива. Также он обладает дополнительной возможность: эффективное добавление/удаление элементов.

Цель. Создание списков. Обход элементов итератором. Ламбда функции, функторы.

Предварительный вариант!

1 Списковые объекты

Каждый тип объекта предназначен для решения определенного типа задачи. Вектор обеспечивает случайный доступ к элементам. В тоже время операции вставки и удаления выполняются крайне неэффективно. Последние операции эффективно выполняются объектами из семейства список. Но, у последних объектов как раз случайный доступ выполняется неэффективно (за что то нужно расплачиваться). У данного объектов есть и другие недостатки, который мы обсудим в следующей заметке.

1.1 Список

Возвращаясь к семейству векторных объектов. В частности, объекты `vector`/`array` обеспечивают через индекс быстрый случайный доступ к любому элементу. Например, если мы хотим изменить значение элемента с индексом 3 в целочисленном векторе a,

```
1 std::vector<int> a;
2 // Идет наполнение вектора.
```

то достаточно сделать следующее:

```
1 a[3] = 5;
```

Таким образом данная операция делается достаточно просто и эффективно. Но, вставка/удаление элемент из середины вектора будет выполнена крайне неэффективно: при каждой такой операции будут перекопированы множество элементов (от данного к концу).

Создание По аналогии с векторами список элементов заданного типа создается так:

```
1 #include <list>
2 ...
3 std::list<int> b;
```

Элементы в список можно добавлять таким же образом как и в вектор, т.е.

```
1 b.push_back( 5 );
```

Добавит число 5 в концу списка.

Цикл по элементам Допустим нам нужно выполнить некоей действие для всех элементов списка. Для векторов было справедливо:

```
1 int i;
2 for( i = 0; i < b.size(); i++)
3   printf("%d ", b[i]);//b.at(i)
4 printf("\n")
```

У списков нет возможности получить доступа к элементу с заданным индексом. Если данную операцию реализовать, то она будет крайне неэффективной (будем пробегать все элементы до заданного).

Так же напомню, что по массивам можно организовать проход по элементам следующим образом:

```
1 int *c;
2 c = (int*)malloc( n * sizeof(int) );
3 int *i;
4 for( i = c; i < c + n; c++)
```

```
5     printf("%d ", *c);
6  printf("\n")
```

В связи со списками появляется новый тип сущности: итератор. Это объект, который позволяет пробежаться по всем элементам списка (не обращаясь к ним по индексу). Последнее делается так:

```
1  std::vector<int>::iterator it;
2  for( it = b.begin(); it != b.end(); it++)
3           printf("%d ", *it);
4           printf("\n");
```

В первой строчке создается объект (итератор), который как раз и используется для обхода всех элементов списка.

В цикле `for` производится инициализация данной переменной начальным значением. Для этого используется метод `begin` объекта список.

Операция перехода к следующему элементу достигается за счет операции ++, т.е. прибавления к объекту итератор 1. Иначе говоря, можно было бы написать

```
1  it = it + 1;
```

Вообще в общем случае,

```
1  it + n
```

соответствует переходу к элементу на n позиций вперед.

Условие завершение цикла заключается в проверки того достигли ли мы конца. Концевому элементу соответствует значение возвращаемое методом `end`.

Доступ к значению самого элемента получается по аналогии с Сишными указателями – за счет разыменования, т.е. использования оператора звездочка.

Самым важным является то, то для вектором сработает тот же самый код. Таким образом, использование итераторов позволяет обобщить проход по всем элементам.

Не по всем элементам Если нужно обойти не все элементы, а например начиная с третьего, до предпоследнего, то пишем:

```
1  std::vector<int>::iterator it;
2  for( it = b.begin() + 2; it != b.end() - 1; it++)
3           printf("%d ", *it);
4           printf("\n");
```

Следует иметь ввиду, что сама операция перехода к заданному элементу может быть и не столь эффективна. Она фактически может оказаться эквивалентной по-элементному проходу. Так, для вектором данная операция будет эффективной (просто передвигается указатель), а для списков не эффективной (придется пробежаться по цепочке элементов).

По всем элементам Если же нужно обойти все элементы (с первого по последний), то можно цикл записать ещё короче:

```
1 for( int a : b)
2         printf("%d ", a);
```

Здесь переменная a (пишется до двоеточия) используется для обхода всех элементов b (пишется после двоеточия). У переменной a Также указан тип данных. Тип int может использоваться для считывания элементов, например, для их печати.

Для того чтобы иметь возможность поменять значение элемента, нужно поменять тип данных на ссылочный. Тогда:

```
1 for(int &a : b)
2         a = a * a;
```

Данный код возведет все элементы вектора в квадрат.

Вставка элемента

1.2 Алгоритмы

В стандартной STL библиотеки имеется большое количество функций для обработки элементов объектов: поиск, по-элементная операция, сортировка, фильтрация и другие. Они в основном оперируют как раз итератором.

Поиск элемента Можно самостоятельно написать код:

```
1 int need = 7;//Значение которое ищем.
2 std::vector<int>::iterator it;
3 for( it = b.begin(); it != b.end(); it++)
4         if( *it == need )
5                 break;
```

Найдя значение итератора можно выполнить какую либо операцию. Например, добавит к элементу значение 100:

```
1 *it = 100 + *it;
```

или вставить значение:

```
1 b.insert( it, 77);
```

Но суть не в этом.

Суть в том, что вместо того, чтобы писать собственную функцию поиска элемента, можно было бы воспользоваться готовой функцией `find`:

```
1 #include <algorithm>
2 ...
3 it = std::find( b.begin(), b.end(), need);
```

Последнее позволило сократить код.

Указывается начальным и последний элемент.

Такие функции существую для разных важных действий.

Произвольный предикат Помимо писка конкретного элемента можно при поиске применить произвольный предикат. Предикат это функция, которая возвращает значение истинности.

Например, если нужно найти первое число делящееся на 3, то можно воспользоваться следующим предикатом.

```
1 bool div3( int a )
2 {
3   return a%3 == 0;
4 }
```

Тогда, необходимо вызвать функцию `find_if` передав в качестве последнего аргумента имя функции.

```
1 #include <algorithm>
2 ...
3 it = std::find_if( b.begin(), b.end(), div3);
```

Подсчет количества Допустим, нужно посчитать количество элементов равных заданному.

Так, если нужно посчитать количество элементов равных 2, то можно написать код:

```
1 // Подсчитаем количества элементов равных 2.
2 int k = std::count( b.begin(), b.end(), 2);
3 // k будет содержать искомое количество.
```

По аналогии с `find_if` можно написать код, который подсчитывает количество элементов для которых истинен предикат:

```
1 // Подсчитаем количества элементов делящихся на 3.
2 int k = std::count_if( b.begin(), b.end(), div3);
3 // k будет содержать искомое количество.
```

Выполнить действие для всех элементов Полным аналогом классического цикла `for` можно считать алгоритм `for_each`.

Ламбда функция Более того, можно воспользоваться ламбда функцией:

```
1 // Подсчитаем количества элементов делящихся на 3.
2 int k = std::count_if( b.begin(), b.end(), [](int a)
3 {
4   return a%3 == 0;
5 });
6 // k будет содержать искомое количество.
```

В более расширенном варианте можно использовать и внешние переменные, т.е. переменные не ходящие в состав аргументов. Например:

```
1 int e;
2 scanf("%d", &e);
3 // Подсчитаем количество элементов меньших e:
4 int k = std::count_if( b.begin(), b.end(), [e](int a)
5 {
6   return a < e;
7 });
8 // k будет содержать искомое количество.
```

Замечу, что переменная e перечислена в квадратных скобках. При таком способе, переменная доступна только на чтение. При попытке записать в неё значение будет выдана ошибка.

При необходимости возможности модификации внешних переменных необходимо воспользоваться модификатором ссылка (&):

```
1 int sum = 0;
2 // Подсчитаем сумму элементов меньших e:
3 std::for_each( b.begin(), b.end(), [&sum, e](int a)
4 {
5   if( a < e )
6     sum += a;
7 });
8 // sum будет содержать искомую сумму.
```

Сортировка Существуют ряд функций, который преобразуют массив в целом. В ним относится сортировка. Сам объект (например `list`) может иметь более эффективную реализацию. Но можно сортировать и с функцией sort.