Односвязный список на C++

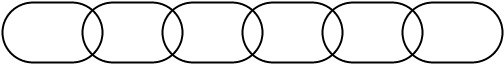
Введение

В стандартную библиотеку C++ входит немало [основных структур данных](http://itnotesblog.ru/note.php?id=164). Среди которых вы найдете списки, стеки, очереди, множества и другие. Но чтобы эффективно пользоваться ими, вы должны хорошо представлять особенности их работы. Речь пойдет об одной из базовых структур: односвязном списке.

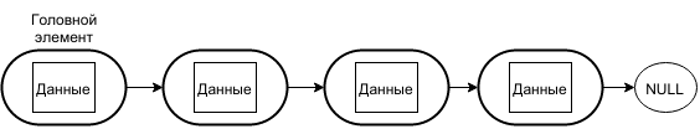
Реклама

Односвязные списки в теории

Чтобы понять, как строится односвязный список, представьте себе цепь. У цепи есть начало и конец, а также звенья, которые последовательно соединяются друг с другом. Вы легко можете пройти от начала цепи до ее конца, переходя последовательно от одного звена к другому:



Именно так и работают односвязные списки. Начало списка называют головным элементом, а звенья - узлами. Конец списка определяется с помощью специального узла (NULL). При этом на каждый узел "вешают" значение, чтобы структура была полезной:



Преимущество односвязных списков заключается в том, что вставка и удаление узлов относительно легко осуществляется в любом месте списка. Однако структура списка ограничивает доступ к его узлам по индексу. Список нельзя индексировать, как массив. Чтобы попасть на некоторый узел односвязного списка, необходимо последовательно пройти весь путь от головного элемента до нужного узла.

Реклама

Интерфейс класса односвязного списка

Односвязный список предназначен для хранения упорядоченного набора однотипных элементов. Чтобы не определять для каждого типа данных свой список, определим шаблонный класс:

**template**< **typename** T >

**class** List {

**private**:

// Объявление структуры узла для использования в классе Iterator

**struct** Node;

**public**:

// Класс итератора односвязного списка

**class** Iterator {

// Пока что опускаем

...

};

**public**:

List();

~List();

// Добавление узла в список

**void** append( **const** T& t );

// Удаление последнего добавленного узла из списка

**void** remove();

// Получить головной элемент списка

T head() **const**;

// Получить итератор на начало списка

Iterator begin() **const**;

// Получить итератор на конец списка

Iterator end() **const**;

// Получить размер списка

size\_t size() **const**;

**private**:

// Структура узла односвязного списка

**struct** Node {

Node() : m\_next( NULL ) { }

Node( **const** T& t ) : m\_t( t ), m\_next( NULL ) { }

// Значение узла

T m\_t;

// Указатель на следующий узел

Node\* m\_next;

};

// Голова односвязного списка

Node\* m\_head;

};

Наш класс односвязного списка будет поддерживать добавление узла в его начало, удаление последнего добавленного узла и получение значения головного элемента. Кроме того, мы реализуем обход списка с помощью итераторов, а также добавим функцию-член для расчета длины списка.

Узел определяется с помощью структуры Node, которая содержит два поля: m\_t - значение, привязанное к узлу, и m\_next - указатель на следующий узел.

Изначально список пуст, поэтому головной элемент указывает на NULL:

**template**< **typename** T >

List< T >::List() : m\_head( NULL ) {

}

Добавление элемента в односвязный список

Добавление узла в односвязный список осуществляется в самое начало. Добавленный узел сам станет новым головным элементом списка:

**template**< **typename** T >

**void** List< T >::append( **const** T &t ) {

// Создаем новый узел для значения

// Не забудем проверить, что память удалось выделить

**if**( Node\* node = **new** Node( t ) ) {

// Новый узел привязывается к старому головному элементу

node->m\_next = m\_head;

// Новый узел сам становится головным элементом

m\_head = node;

}

}

Удаление элемента из односвязного списка

При удалении узла из односвязного списка усекается его головной элемент. Если в списке еще остаются узлы, то новым головным элементом становится узел, следующий за головным элементом, иначе остается NULL:

**template**< **typename** T >

**void** List< T >::remove() {

**if**( m\_head ) { // Если список не пуст:

// Сохраняем указатель на второй узел, который станет новым головным элементом

Node\* newHead = m\_head->m\_next;

// Освобождаем память, выделенную для удаляемого головного элемента

**delete** m\_head;

// Назначаем новый головной элемент

m\_head = newHead;

} // Иначе могли бы возбудить исключение

}

Список является динамической структурой. При добавлении новых узлов выделяется память из кучи, которую нужно освободить, чтобы не создавать утечек памяти. Имея функцию удаления элементов, реализовать деструктор односвязного списка совсем не сложно:

**template**< **typename** T >

List< T >::~List() {

**while**( m\_head ) {

remove();

}

}

Итератор односвязного списка

Функциональность для добавления/удаления узлов в список лучше не смешивать с задачей его обхода. Для этого существуют итераторы. Создадим итератор односвязного списка в стиле C++. Вернемся к его определению, которое ранее пропустили:

**class** Iterator {

**public**:

Iterator( Node\* node ) : m\_node( node ) { }

// Проверка на равенство

**bool** **operator**==( **const** Iterator& other ) **const** {

**if**( **this** == &other ) {

**return** **true**;

}

**return** m\_node == other.m\_node;

}

// Проверка на неравенство

**bool** **operator**!=( **const** Iterator& other ) **const** {

**return** !**operator**==( other );

}

// Получение значения текущего узла

T **operator**\*() **const** {

**if**( m\_node ) {

**return** m\_node->m\_t;

} // Иначе достигнут конец списка. Уместно возбудить исключение

**return** T();

}

// Переход к следующему узлу

**void** **operator**++() {

**if**( m\_node ) {

m\_node = m\_node->m\_next;

} // Иначе достигнут конец списка. Уместно возбудить исключение

}

**private**:

Node\* m\_node;

};

В качестве итераторов для начала и конца списка вернем следующее:

**template**< **typename** T >

**typename** List< T >::Iterator List< T >::begin() **const** {

// Итератор пойдет от головного элемента...

**return** Iterator( m\_head );

}

**template**< **typename** T >

**typename** List< T >::Iterator List< T >::end() **const** {

// ... и до упора, т.е. NULL

**return** Iterator( NULL );

}

Теперь список легко можно обойти от начала до конца. Но не забудем про функцию для вычисления его длины. Воспользуемся механизмом итераторов, чтобы упростить задачу:

**template**< **typename** T >

size\_t List< T >::size() **const** {

size\_t s = 0;

**for**( Iterator it = begin(); it != end(); ++it ) {

++s;

}

/\*

Но можно и без итераторов

for( Node\* n = m\_head; n != NULL; n = n->m\_next ) {

++s;

}

\*/

**return** s;

}

Заключение

Учитывайте, что рассмотренный пример является лишь краткой демонстрацией принципов работы с односвязным списком в C++. В реальных программах практически всегда лучше использовать стандартную библиотечную реализацию списка и любой другой структуры данных. Тогда вам не только потребуется делать меньше работы, но вы получите в распоряжение хорошо отлаженную и оптимизированную версию, которой сможете смело пользоваться.