Односвязный список — конспект темы

Как устроен односвязный список

Односвязный или **однонаправленный** список — базовая структура данных, которая состоит из элементов одного типа. Их логически связывают между собой указатели. Каждый элемент списка указывает на следующий, а последний — на `nullptr`.

Хранятся элементы списка, как правило, в динамической памяти.

Передвигаться по его элементам односвязного списка можно только в прямом направлении.

Односвязный список предоставляет следующие операции:

- вставка элемента в начало или конец списка,
- вставка элемента после некоторого элемента списка,
- удаление элемента следующего за данным элементом списка,
- проверка списка на пустоту,
- определение количества элементов в списке.

Плюсы односвязного списка:

- Вставка и удаление элемента выполняются за константное время, то есть не зависят от количества элементов в списке;
- Размер списка ограничен лишь объёмом доступной памяти.

Минусы односвязного списка:

- Узнать адрес элемента по его порядковому номеру — операция линейной сложности. Чтобы определить адрес N-го элемента списка, нужно последовательно перебрать все N-1 элементов, начиная с начала списка;
- Неэффективное расходование памяти: помимо данных, каждый элемент списка хранит указатель на следующий элемент. Кроме того, при каждом

создании объекта в динамической памяти пара десятков байт расходуется на поддержание структуры кучи;

- Соседние элементы списка могут располагаться в памяти непоследовательно, что снижает эффективность работы кэш-памяти.

Элемент списка называется **узлом**.

Покоряем итераторы

Итератор — специальный объект, который играет роль указателя на элемент контейнера.

Итераторы и полуинтервалы — абстракции. Итераторы разных контейнеров похожи. Они предоставляют:

- операции `*` и `->` для доступа к элементам,
- операции `++` и `--` для обхода элементов в прямом и обратном направлении.

Чтобы класс или структура с точки зрения стандартных алгоритмов считались итератором, внутри должны быть объявлены вложенные типы.

- `iterator_category`. Задаёт категорию итератора. Так как односвязный список поддерживает перебор элементов только в прямом направлении, его категория итератора должна быть равна `std::forward_iterator_tag`.
- `value_type`. Задаёт тип элементов, доступ к которым предоставляет итератор. В случае односвязного списка этот тип совпадает с параметром `Type` шаблона `SingleLinkedList`.
- `difference_type`. Задаёт тип для хранения смещения между двумя итераторами. Для `SingleLinkedList` можно использовать тип `std::ptrdiff_t`. Это целое число со знаком, разрядность которого совпадает с разрядностью указателя на целевой платформе.
- `pointer`. Задаёт тип указателя, возвращаемого оператором `->`. Для неконстантного итератора `SingleLinkedList` это `Type*`, а для константного — `const Type*`.
- `reference`. Задаёт тип ссылки, которую возвращает оператор разыменования `*`. В зависимости от константности итератора `SingleLinkedList` это либо `Type&`, либо `const Type&`.

Итератор категории `std::forward_iterator_tag` соответствует категории однонаправленного итератора. В нём должны быть объявлены следующие операции.

- `==` и `!=`. Итераторы одного и того же списка равны, если указывают на одну и ту же позицию. Константные и неконстантные итераторы одного и того же списка можно сравнивать друг с другом;
- `++`. Перемещает итератор на следующую позицию в списке;
- `*` и `->` для доступа к элементу списка. Возвращают ссылку и указатель на значение, хранящееся в списке, а не на весь узел списка, задаваемый вложенной структурой `Node`. Итераторы должны скрывать внутреннее устройство контейнера от внешнего мира.

Ключевое слово `friend` позволяет объявить класс или структуру дружественными. Класс разрешает дружественным классам, структурам и функциям доступ к своей приватной области. Пользуйтесь этим осторожно, так как изменения в приватной части класса могут повлиять на работу его друзей.

Сравнение, копирование и присваивание

Содержимое стандартных контейнеров, таких как `vector`, можно сравнивать, используя операции `==`, `!=`, `>=`, `<=`, `>`, `<`, когда эти операции определены для их элементов.

Конструктор копирования важно сделать безопасным к возникновению исключений. Для этого используйте идиому copy-and-swap:

- Создайте внутри конструктора копирования временный односвязный список и последовательно скопируйте внутрь него элементы исходного списка. Если на этом этапе будет выброшено исключение, деструктор временного списка освободит память от его элементов.
- Когда временный список будет содержать копию исходного списка, останется использовать метод `swap` и обменять состояние текущего экземпляра класса и временного списка.

Операцию присваивания, последнюю из «Правила трёх» легко реализовать, применив идиому copy-and-swap.

- Проверьте, не выполняется присваивание списка самому себе, сравнив адреса левого и правого аргументов операции присваивания.
- Сконструируйте временную копию правого аргумента. Выбрасывание исключения на этом этапе никак не повлияет на состояние текущего объекта.
- Используйте метод `swap`, чтобы обменять содержимое временной копии и текущего объекта. Операция `swap` исключений не выбрасывает и выполняется быстро. Временная копия будет содержать предыдущее значение левого аргумента, а текущий экземпляр — копию правого аргумента операции присваивания.
- При выходе из оператора `=` временный объект будет разрушен, освободив память от предыдущего содержимого списка.

Вставка и удаление в произвольной позиции

Чтобы вставить элемент в начало списка, нужно уметь адресовать позицию предшествующую первому элементу. Вставка после позиции, предшествующей первому элементу, и есть вставка в начало списка.

Чтобы получить итератор, используют методы `before_begin` и `cbefore_begin`.

Создав новый узел со вставляемым значением, нужно сослаться на него с предшествующего узла. С нового узла — сослаться на узел, на который раньше ссылался предшествующий.

После вставки увеличьте размер списка и венрите итератор, ссылающийся на вновь созданный узел.

При вставке нужно выделить динамическую память и скопировать значение. Это может привести к выбрасыванию исключения. Чтобы избежать этого, сначала создайте узел в динамической памяти, а потом обновите связи между узлами и размер списка.

При удалении первого элемента указатель на начало списка начинает указывать на прежний второй узел, а узел, хранивший первый элемент, удаляется. Операция не требует динамического выделения памяти, поэтому не выбрасывает исключений.

Чтобы удалить элемент в произвольной позиции односвязного списка, нужен доступ к узлу, который предшествует удаляемому элементу. После удаления

указатель с предшествующего узла будет ссылаться на узел, следующий за удаляемым.

Работаем с массивами

Структура данных, где все элементы имеют одинаковый размер и хранятся в непрерывной области памяти один за другим, называется **массивом**. Доступ к элементу массива осуществляется по его индексу.

Самый простой способ создать массив — объявить переменную и указать после её имени размер массива в квадратных скобках. Этот массив располагается в автоматической или статической области памяти, в зависимости от места своего объявления:

```
// Массив из 10 int-ов со статическим временем жизни
int global_array[10];

int main() {
    // Массив из 5 int-ов с автоматическим временем жизни
    int numbers[5];

    numbers[2] = 3;
}
```

Размер массива в статической или автоматической области памяти должен быть константой времени компиляции. Компилятору нужно знать размер кадра стека функции, которая использует массивы.

Увеличить или уменьшить размер такого массива нельзя — сколько в нём было элементов при объявлении, столько останется до окончания его времени жизни. Операций копирования и присваивания для массивов тоже нет.

Когда работаете с массивами переменного размера в C++, выбирайте контейнер `vector`. Для хранения небольших массивов фиксированного размера в области стека можно использовать контейнер `std::array`

Для создания массива в динамической памяти служит специальная форма оператора `new` — `new Тип[размер]`.

В отличие от массива на стеке или в статической области памяти, размер массива в куче может быть произвольным значением, а не константой.

Оператор `new[]` возвращает указатель на первый элемент созданного массива:

```
size_t size;
cin >> size;

// Создаёт в куче массив из size элементов
int* numbers = new int[size];
```

Закончив работу с массивом в динамической памяти, удалите его, используя оператор `delete[]`. Этот оператор вызывает деструкторы у всех элементов массива и освобождает память.

Обратиться к элементам массива можно, используя оператор `[]`:

```
int* my_array = new int[10];

// Следующие две строки делают одно и то же
my_array[0] = 1;
*my_array = 1;

my_array[3] = my_array[2];

delete[] my_array;
```

Создаём RAII-обёртку над массивом в динамической памяти

Часто перегружают две версии оператора `[]` — константную и неконстантную.

Константная версия возвращает константную ссылку на элемент. Благодаря этому можно защитить содержимое контейнера от модификации — модифицировать значение константы нельзя.

Бывают исключения: в контейнере `map` есть только неконстантная версия оператора `[]`. Если элемента с указанным ключом нет, она вставляет ключ со значением по умолчанию.