Тема 2.2 Списковые структуры: стек, очередь, односвязный и двухсвязный списки, упорядоченный список (словарь), кольцо

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый **динамическим распределением памяти**, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

- она не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.:

Каждой динамической структуре данных сопоставляется статическая переменная типа указатель (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Необходимость в динамических структурах данных обычно возникает в следующих случаях.

- Используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.
- В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
- Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Динамические структуры, по определению, характеризуются отсутствием физической смежности элементов структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется **связным**.

Достоинства связного представления данных – в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются следующие:

- на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива – с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются указатели, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

- *информационного поля или поля данных*, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой - вектором, массивом, другой динамической структурой и т.п.;
- *поле связок*, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя "видимым" делается только содержимое информационного поля, а поле связок используется только программистом-разработчиком.

Достоинства связного представления данных - в возможности обеспечения значительной изменчивости структур;

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.
- Вместе с тем связное представление не лишено и недостатков, основные из которых:
- на поля связок расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива - с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

- однонаправленные (односвязные) списки. В односвязном списке каждый элемент информации содержит ссылку на следующий элемент списка. Каждый элемент данных обычно

представляет собой структуру, которая состоит из информационных полей и указателя связи. Существует два основных способа построения односвязного списка. Первый способ — помещать новые элементы в конец списка. Второй — вставлять элементы в определенные позиции списка, например, в порядке возрастания.

- двунаправленные (двусвязные) списки. Двусвязный список состоит из элементов данных, каждый из которых содержит ссылки как на следующий, так и на предыдущий элементы. Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Вероятно, наиболее важное из них состоит в том, что перемещение по списку возможно в обоих направлениях. Это упрощает работу со списком, в частности, вставку и удаление. Помимо этого, пользователь может просматривать список в любом направлении. Еще одно преимущество имеет значение только при некоторых сбоях. Поскольку весь список можно пройти не только по прямым, но и по обратным ссылкам, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.
- циклические списки. Основное отличие циклического списка состоит в том, что в списке нет пустых указателей. Последний элемент списка содержит указатель, связывающий его с первым элементом. Для полного обхода такого списка достаточно иметь указатель только на текущий элемент.
- Стек. структура данных, в которой доступ к элементам организован по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю. Добавление элемента, называемое также проталкиванием (push), возможно только в вершину стека (добавленный элемент становится первым сверху). Удаление элемента, называемое также выталкиванием (pop), тоже возможно только из вершины стека, при этом второй сверху элемент становится верхним.
- Дек. Деком (англ. deque – аббревиатура от double-ended queue, двухсторонняя очередь) называется структура данных, в которую можно удалять и добавлять элементы как в начало, так и в конец. Дек хранится в памяти так же, как и очередь.
- Очередь. Структура данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.
- бинарные деревья. древовидная структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками

Стек

Стеком называется структура данных, в которой данные, записанные первыми, извлекаются последними (FILO: First In - Last Out). Например, если мы записали в стек числа 1, 2, 3, то при последующем извлечении получим 3, 2, 1.

Удобно представить стек в виде узкого колодца или рюкзака, в который мы можем класть предмет только наверх и забирать только верхний предмет.

Мы будем реализовывать стек на одномерном массиве, а указателем на вершину стека (первый свободный элемент в массиве) в таком случае будет целочисленная переменная — индекс свободного элемента. Дня стека определены две операции push(x) — записать в стек элемент (в нашем случае — число) и pop() — извлечь из стека элемент.

Размер стека определим в виде константы:

```
#define MAXN 1000
Сам стек будем описывать в виде структуры,
typedef struct
{
   int sp;
   int val[MAXN];
```

```
} stack;
```

Мы можем создавать стеки, просто написав, например, stack a, Ь;.

При передаче параметров функции нам нужно будет указывать, с каким конкретно стеком мы хотим работать, а чтобы данные не копировались, будем передавать их по указателю.

```
void push(stack *s, int x)
{
    s->val [s->sp++] = x;
}
int pop(stack *s)
{
    return s->val[—-s->sp];
}
```

В функции push мы записываем добавляемый элемент в первую свободную позицию, а затем увеличиваем ее номер (постинкремент). В функции pop мы уменьшаем указатель на вершину стека (п ре декремент), а затем возвращаем значение из последней занятой ячейки.

Для стеков, созданных в статической памяти (так, как мы создавапи их в примере), вызовы функций будут выглядеть как push(&a, x); x = pop(&a); где x — число, а a — стек. Перед этим необходимо инициализировать указатель на вершину стека нулем (s.sp = 0).

Если нам будут необходимы какие-то дополнительные функции работы со стеком, (например, определение пуст ли стек или количества элементов в нем), то всю необходимую информацию мы можем найти в поле sp. Напомним, что sp — текущее количество элементов в стеке.

При планировании размера сгека надо учитывать не общее количество элементов, а максимальное количество одновременно находящихся в стеке элементов (хотя часто эти значения совпадают).

Очередь

Очередь — это информационная структура, в которой для добавления элементов доступен только один конец, называемый *хвостом*, а для удаления — другой, называемый *головой*. В англоязычной литературе для обозначения очередей довольно часто используется аббревиатура FIFO (first-in-first-out — первый вошёл — первым вышел).

Очередь мы будем реализовывать на одномерном массиве, аналогично стеку. Тут нам придется немного отойти от аналогий с реальностью для повышения производительности. Если реализовывать очередь в программе как очередь в магазине, где люди постепенно двигаются к кассе, то извлечение элемента из очереди будет иметь сложность O(N), где N - количество элементов в очереди), т.к. все элементы будет необходимо передвинуть. Гораздо удобнее в нашей ситуации перемещать «кассу», т.е. изменять указатель на начало очереди.

Однако в этом случае возникает другая проблема: если в предыдущем случае размер очереди ограничивался количеством одновременно находящихся в ней элементов, то здесь нам необходимо будет создавать очередь с размером, равным общему количеству элементов, которые в ней побывают.

Эту проблему мы решим, закольцевав очередь. Можно представить круг, где помечены две позиции — с какой уходить, и на какую становиться новому элементу. Для этого в реализации мы будем брать оба указателя по модулю MAXN.

Очередь также реализуем в виде структуры:

```
typedef                                struct                                {

int qh, qt;
     int val[MAXN];
} queue;
```

Теперь мы можем создавать очереди, просто написав queue a, b;

Для очереди существует две операции: извлечь элемент из головы (head) очереди (deq) и добавить элемент в хвост (tail) очереди (enq).

```
void enq (queue *q, int x)
{
    q->val[(q->qt++)7,MAXN] = x;
}
int deq(queue *q)
{
    return q->val[(q->qh++)7,MAXN] ;
}
```

Как и в прошлый раз, необходимо передавать в функции указатель, т.е. их вызов должен выглядеть как: enq(&a, x); x = deq(&a); где x — число, а a — очередь. Так же, как и в случае со стеком, мы должны предварительно инициачизировать оба указателя очереди нулями: a.qt = 0; a.qh = 0;

Признаком того, что очередь пуста или переполнилась, следует считать равенство полей qt и qh. Количество элементов в очереди определяется так: qlen = (qt-qh)%MAXN.

Очереди часто используются в качестве буферов и во многих устройствах реачизованы аппаратно.

Списки

Рассмотрим еще одну динамическую структуру данных, называемую связным списком. Каждый элемент списка представляет собой структуру, одно поле которой содержит информацию (ключ), а другое — ссылку на следующий элемент. Существуют также двусвязные списки, в которых хранится ссылка не только на следующий элемент, но и на предыдущий. Начинается список с указателя на элемент списка. В целом его вид можно представить следующим образом:

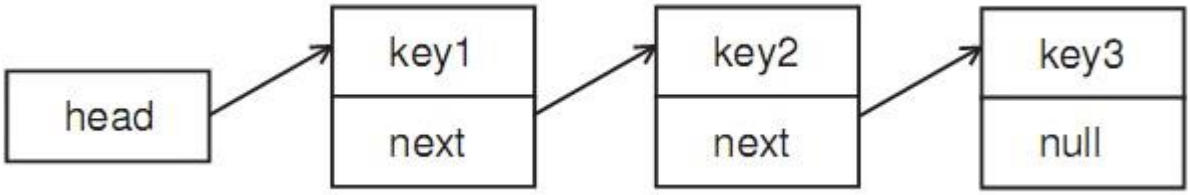

Рисунок 1. Общий вид списка

Один элемент списка будем описывать с помощью следующей структуры:

```
typedef struct
{
    int key;
    struct _list *next;
}                                                    list;
```

Чтобы работать со списком, необходимы указатели на элемент, которые заводятся следующим образом: list *head=NULL, *temp; Добавление происходит в начало списка. Это реализуется следующим фрагментом кода:

```
temp = (list*)malloc(sizeof(list));
temp->key = newkey;
temp->next = head;
head = temp;
```

Первая строка выделяет память под новый элемент, вторая записывает новое значение ключа, третья присваивает полю next вновь созданного элемента указаетль на текущее начало списка, а четвертая установливает указатель на начало списка на вновь созданный элемент. Порядок действий показан на рисунке:

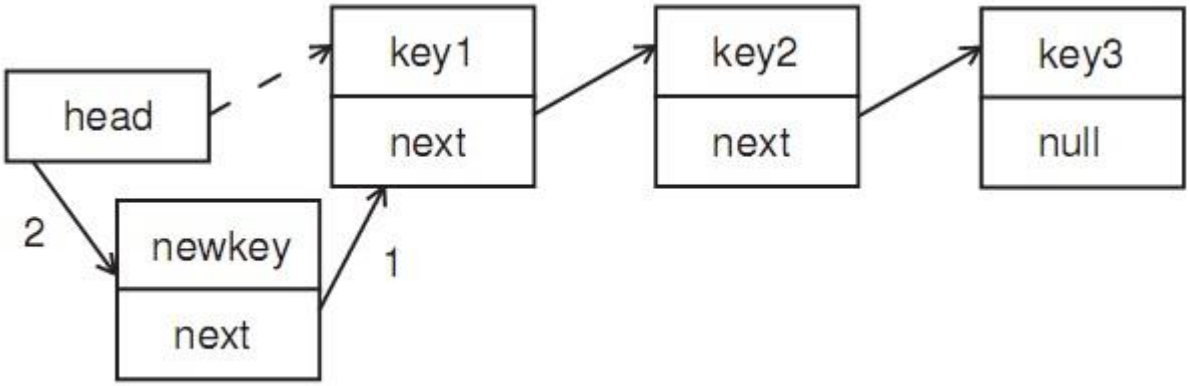

Рисунок 2. Добавление элемента в начало списка

Аналогично осуществляется вставка после элемента, на который имеется ссылка, достаточно заменить head в нашем коде на его поле next.

Для поиска элемента по ключу необходимо пройти весь список. Напишем функцию, которая возвращает указатель на элемент по его значению ключа или NULL, если элемента с таким ключом не существует.

```
list* find(list* head, int key)
{
    list *now=head;
    while (now != NULL)
    {
        if (key == now->key)
            break;
            now = now->next;
    }
    return now;
}
```

Однако, удаление элемента невозможно реализовать только зная ссылку на него (т.к. необходимо, чтобы список остался связным, а удаление элемента создаст в нем «дырку»). Напишем отдельную функцию удаления элемента с заданным ключом и возврщующую ссылку на новый список (без этого элемента).

```
list* del(list *head, int key)
{
    list *now=head, *prev;
    if (key == head->key)
    {
        head = head->next;
        free(now);
    }
    else
    {
        prev = now;
        now = now->next;
        if (key == now->key)
        {
            prev->next = now->next;
            free(now);
        }
    }
    return head;
}
```

Здесь мы отдельно рассматриваем случай, когда необходимо удалить первый элемент списка, т.к. он не имеет предыдущего. В противном случае мы делаем удаление согласно рисунку:

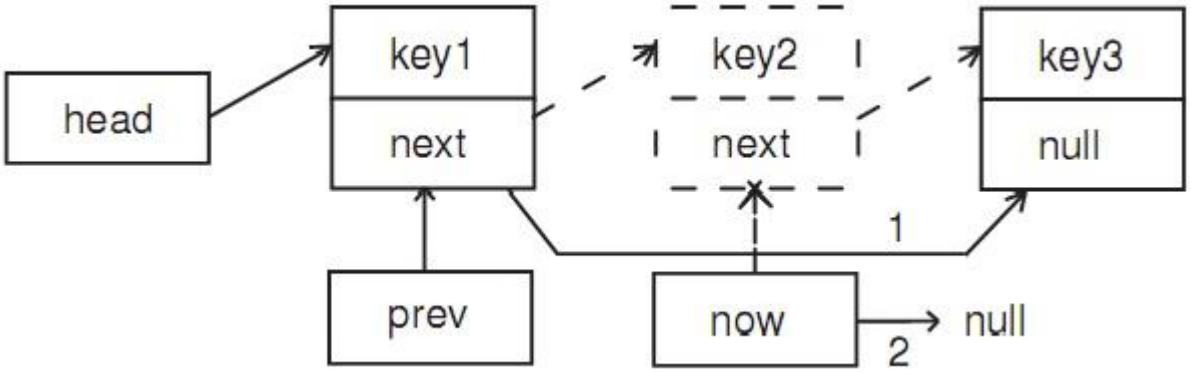

Рисунок 3. Удаление элемента из списка

Сначала осуществляем присваивание поля next для предыдущего элемента (действие 1), а затем удаление текущего элемента (действие 2).

Над списком можно определить множество различных функций (например, объединение двух списков, удаление элементов, обладающих каким-либо признаком и т.д.), но они все базируются на изложенных выше идеях.

Контейнеры стандартной библиотеки c++

Для управления наборами объектов в стандартной библиотеке C++ определены контейнеры.

Контейнер представляет коллекцию объектов определенного типа и позволяет и управлять доступом к этим элементам.

В C++ есть два типа контейнеров: ***ассоциативные и последовательные.***

Последовательный контейнер (sequential container) хранит элементы последовательно, элементы располагаются друг рядом с другом. Однако меняется их порядок в зависимости от конкретного контейнера. .

Типы последовательных контейнеров:

- **array**: коллекция фиксированного размера. Поддерживает произвольный доступ к любому элементу в контейнере. Добавлять или удалять элементы из контейнера нельзя.
- **vector**: коллекция переменного размера. Поддерживает произвольный доступ к любому элементу в контейнере. Обеспечивает добавление и удаление элементов из любого места контейнера.
- **deque**: двусторонняя очередь. Поддерживает произвольный доступ к любому элементу в контейнере. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
- **list**: двухсвязный список Поддерживает только последовательный двухнаправленный доступ к элементам. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.
- **forward_list**: односвязный список. Поддерживает только однонаправленный последовательный доступ к элементам. Обеспечивает удаление и добавление элементов в начале и в конце контейнера.

Таким образом, стандартная библиотека C++ по умолчанию содержит ряд контейнеров, которые представляют определенные структуры данных. Все они имеют как некоторые общие, так и специфические возможности. За исключением класса array все они поддерживают добавление и удаление элементов. Основное различие между ними состоит в том, как они обеспечивают

добавление и удаление элементов, а также доступ к элементам в контейнере. И в зависимости от ситуации и потребностей можно использовать тот или иной тип контейнеров.

Например, если надо иметь возможность произвольный элемент контейнера, то применяется array или vector (с list или forwarded_list может потребоваться пробегаться по списку, чтобы найти нужный элемент). Если же надо иметь возможность добавлять или удалить элементы в середине контейнера, то можно применять list или forwarded_list, что с вектором сложнее сделать. Однако наиболее часто используется вектор, как более гибкий тип данных. Другие типы контейнеров применяются гораздо реже.

Адаптеры контейнеров

Кроме последовательных контейнеров есть так называемые **адаптеры контейнеров** (container adaptor). Технически они не являются контейнерами, а инкапсулируют один из вышеописанных контейнеров (например, вектор) и позвляют работать с этими контейнерами определенным образом.

Это следующие типы

- **std::stack<>**: представляет структуру данных "стек"
- **std::queue<>**: представляет структуру данных "очередь"
- **std::priority_queue<>**: также представляет очередь, но при этому ее элементы имеют приоритеты

Ассоциативные контейнеры

Ассоциативные контейнеры (associative containers) представляют такие контейнеры, где с каждым элементом ассоциирован некоторый ключ, и этот ключ применяется для доступа к элементу в контейнере.

В C++ ассоциативные контейнеры представлены множествами (set) и картами/словарями (map).

Вектор

Вектор представляет контейнер, который содержит коллекцию объектов одного типа. Для работы с векторами необходимо включить заголовок:

```
#include <vector>
```

Определим простейший вектор:

```
std::vector<int> numbers;
```

В угловых скобках указывается тип, объекты которого будут храниться в векторе. То есть вектор numbers хранит объекты типа int. Однако такой вектор пуст. Он не содержит никаких элементов.

Но мы можем инициализировать вектор одним из следующих способов:

```
std::vector<int> v1;              // пустой вектор
std::vector<int> v2(v1);          // вектор v2 - копия вектора v1
std::vector<int> v3 = v1;         // вектор v3 - копия вектора v1
std::vector<int> v4(5);           // вектор v4 состоит из 5 чисел, каждое число равно 0
std::vector<int> v5(5, 2);        // вектор v5 состоит из 5 чисел, каждое число равно 2
std::vector<int> v6{1, 2, 4, 5};  // вектор v6 состоит из чисел 1, 2, 4, 5
std::vector<int> v7 = {1, 2, 3, 5}; // вектор v7 состоит из чисел 1, 2, 3, 5
```

Важно понимать отличие в данном случае круглых скобок от фигурных:

```
std::vector<int> v1(5);       // вектор состоит из 5 чисел, каждое число в векторе равно 0
std::vector<int> v2{5};       // вектор состоит из одного числа, которое равно 5
std::vector<int> v3(5, 2);    // вектор состоит из 5 чисел, каждое число равно 2
std::vector<int> v4{5, 2};    // вектор состоит из двух чисел 5 и 2
```

При этом можно хранить в векторе элементы только одного типа, который указан в угловых скобках. Значения других типов в вектор сохранить нельзя, как например, в следующем случае:

```
std::vector<int> v{5, "hi"};
```

Обращение к элементам и их перебор

Для обращения к элементам вектора можно использовать разные способы:

- **[index]**: получение элемента по индексу (также как и в массивах), индексация начинается с нуля
- **at(index)**: функция возвращает элемент по индексу
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент

Выполним перебор вектора и получим некоторые его элементы:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers {1, 2, 3, 4, 5};

    int first = numbers.front();    // 1
    int last = numbers.back();      // 5
    int second = numbers[1];        // 2
    std::cout << "first: " << first << std::endl;
    std::cout << "second: " << second << std::endl;
    std::cout << "last: " << last << std::endl;

    numbers[0] = 6;  // изменяем значение
    for(int n : numbers)
        std::cout << n << "\t"; // 6  2  3  4  5

    std::cout << std::endl;
}
```

При этом следует учитывать, что индексация не добавляет элементов. Например, если вектор содержит 5 элементов, то мы не можем обратиться к шестому элементу:

```
std::vector<int> numbers {1, 2, 3, 4, 5};
numbers[5] = 9;
```

При таком обращении результат неопределен. Некоторые комиляторы могут генерировать ошибку, некоторые продолжат работать, но даже в этом случае такое обращение будет ошибочно, и оно в любом случае не добавит в вектор шестой элемент.

Чтобы избежать подобных ситуаций, можно использовать функцию **at()**, которая хотя также возвращает элемент по индексу, но при попытке обращения по недопустимому индексу будет генерировать исключение **out_of_range**:

```
#include <iostream>
#include <vector>
#include <stdexcept>

int main()
{
    std::vector<int> numbers { 1, 2, 3, 4, 5};
    try
    {
        int n = numbers.at(8);
    }
    catch (std::out_of_range e)
    {
        std::cout << "Incorrect index" << std::endl;
    }
}
```

Итераторы

Итераторы обеспечивают доступ к элементам контейнера и представляют реализацию распространенного паттерна объектно-ориентированного программирования "Iterator". С помощью итераторов очень удобно перебирать элементы.

В C++ итераторы реализуют общий интерфейс для различных типов контейнеров, что позволяет использовать единой подход для обращения к элементам разных типов контейнеров.

Стоит отметить, что итераторы имеют только контейнеры, адаптеры контейнеров — типы **std::stack**, **std::queue** и **std::priority_queue** итераторов не имеют.

Итератор описывается типом **iterator**. Для каждого контейнера конкретный тип итератора будет отличаться. Так, итератор для контейнера `list<int>` представляет тип **list<int>::iterator**, а итератор контейнера `vector<int>` представляет тип **vector<int>::iterator** и так далее. Однако общий функционад, который применяется для доступа к элементам, будет аналогичен.

Для получения итераторов контейнеры в C++ обладают такими функциями, как **begin()** и **end()**.

Функция **begin()** возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов).

Функция **end()** возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера.

Если контейнер пуст, то итераторы, возвращаемые обоими методами begin и end совпадают. Если итератор begin не равен итератору end, то между ними есть как минимум один элемент.

Обе этих функции возвращают итератор для конкретного типа контейнера:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers{ 1,2,3,4 };
    std::vector<int>::iterator iter = numbers.begin();  // получаем итератор
}

```

В данном случае создается вектор - контейнер типа vector, который содержит значения типа int. И этот контейнер инициализируется набором {1, 2, 3, 4}. И через метод `begin()` можно получить итератор для этого контейнера. Причем этот итератор будет указывать на первый элемент контейнера.

С итераторами можно проводить следующие операции:

- ***iter**: получение элемента, на который указывает итератор
- **++iter**: перемещение итератора вперед для обращения к следующему элементу

- **--iter**: перемещение итератора назад для обращения к предыдущему элементу. Итераторы контейнера **forward_list** не поддерживают операцию декремента.
- **iter1 == iter2**: два итератора равны, если они указывают на один и тот же элемент
- **iter1 != iter2**: два итератора не равны, если они указывают на разные элементы
- **iter + n**: возвращает итератор, который смещен от итератора iter на n позиций вперед
- **iter - n**: возвращает итератор, который смещен от итератора iter на n позиций назад
- **iter += n**: перемещает итератор на n позиций вперед
- **iter -= n**: перемещает итератор на n позиций назад
- **iter1 - iter2**: возвращает количество позиций между итераторами iter1 и iter2
- **>, >=, <, <=**: операции сравнения. Один итератор больше другого, если указывает на элемент, который ближе к концу

Стоит отметить, что итераторы не всех контейнеров поддерживают все эти операции.

Итераторы для типов **std::forward_list**, **std::unordered_set** и **std::unordered_map** не поддерживают операции --, -= и -. (поскольку std::forward_list - однонаправленный список, где каждый элемент хранит указатель только на следующий элемент)

Итераторы для типа **std::list** поддерживают операции инкремента и декремента, но не поддерживаются операции +=, -=, + и -. Те же ограничения имеют итераторы контейнеров **std::map** и std::set.

Операции +=, -=, +, -, <, <=, >, >= и <=> поддерживаются только итераторами произвольного доступа (итераторы контейнеров **std::vector**, **array** и **deque**)

Получение и изменение элемента контейнера

Поскольку итератор по сути представляет указатель на определенный элемент, то через этот указатель мы можем получить текущий элемент итератора и изменить его значение:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers{ 1,2,3,4 };
    auto iter { numbers.begin() };  // получаем итератор

    // получаем элемент, на который указывает итератор
    std::cout << *iter << std::endl;    // 1
    // изменяем элемент
    *iter = 125;
    // проверяем изменение элемента
    std::cout << numbers[0] << std::endl;    // 125
}
```

После получения итератора он будет указывать на первый элемент контейнера. То есть при выражение `*iter` возвратит первый элемент вектора.

Прибавляя или отнимая определенное число, можно переместить итератор вперед или назад на определенное количество элементов:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers{ 10, 20, 30, 40 };
    auto iter { numbers.begin() };  // получаем итератор

    // переходим на 1 элемент вперед ко 2-му элементу
    ++iter;
    std::cout << *iter << std::endl;    // 20

    // переходим на 2 элемента вперед к 4-му элементу
    iter +=2;
    std::cout << *iter << std::endl;    // 40

    // переходим назад на 3 элемента к 1-му элементу
    iter = iter - 3;
    std::cout << *iter << std::endl;    // 10
}
```

Опять же повторю, что стоит учитывать, что не все операции поддерживаются итераторами всех контейнеров.

Перебор контейнера

Например, используем итераторы для перебора элементов вектора:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers{ 10, 20, 30, 40 };
    auto iter { numbers.begin() };  // получаем итератор

    while(iter!=numbers.end())    // пока не дойдем до конца
    {
        std::cout << *iter << std::endl;// получаем элементы через итератор
        ++iter;              // перемещаемся вперед на один элемент
    }

    // аналогичный пример с циклом for
    for(auto start{numbers.begin()}; start !=numbers.end(); start++ )
    {
        std::cout << *start << std::endl;
    }
}
```

При работе с контейнерами следует учитывать, что добавление или удаление элементов в контейнере может привести к тому, что все текущие итераторы для данного контейнера, а также ссылки и указатели на его элементы станут недопустимыми. Поэтому при добавлении или удалении элементов в контейнере в общем случае следует перестать использовать текущие итераторы для этого контейнера.

Константные итераторы

Если контейнер представляет константу, то для обращения к элементам этого контейнера можно использовать только константный итератор (тип **const_iterator**). Такой итератор позволяет считывать элементы, но не изменять их:

```
const vector<int> numbers{1, 2, 3, 4, 5};
for(auto iter {numbers.begin()}; iter != numbers.end(); ++iter)
{
    std::cout << *iter << std::endl;
    // так нельзя сделать
    //*iter = (*iter) * (*iter);
}
```

В данном случае итератор `iter` будет представлять тип `std::vector<int>::const_iterator`.

Для получения константного итератора также можно использовать функции **cbegin()** и **cend**. При этом даже если контейнер не представляет константу, но для его перебора используется константный итератор, то опять же нельзя изменять значения элементов этого контейнера:

```
#include <iostream>
#include <vector>
```

```
3
4  int main()
5  {
6      std::vector<int> numbers { 1, 2, 3, 4, 5 };
7      for (auto iter {numbers.cbegin()}; iter != numbers.cend(); ++iter)
8      {
9          std::cout << *iter << std::endl;
10         // так нельзя сделать, так как итератор константный
11         //*iter = (*iter) * (*iter);
12     }
13 }
```

Стоит отметить, что для типов **std::set** (множество) и **std::map** (словарь) доступны только константные итераторы.

Реверсивные итераторы

Реверсивные итераторы позволяют перебирать элементы контейнера в обратном направлении. Для получения реверсивного итератора применяются функции **rbegin()** и **rend()**, а сам итератор представляет тип **reverse_iterator**:

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> numbers { 1, 2, 3, 4, 5 };
7      for (auto iter {numbers.rbegin()}; iter != numbers.rend(); ++iter)
8      {
9          std::cout << *iter << "\t";
10     }
11     std::cout << std::endl;
12 }
```

В данном случае итератор будет представлять тип `std::vector<int>::reverse_iterator`. Консольный вывод программы:

```
5    4    3    2    1
```

Если надо обеспечить защиту от изменения значений контейнера, то можно использовать константный реверсивный итератор, который представлен типом **const_reverse_iterator** и который можно получить с помощью функций **crbegin()** и **crend()**:

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> numbers { 1, 2, 3, 4, 5 };
7      for (auto iter {numbers.crbegin()}; iter != numbers.crend(); ++iter)
```

```
    {
        std::cout << *iter << std::endl;
        // так нельзя сделать, так как итератор константный
        //*iter = (*iter) * (*iter);
    }
}
```

Итераторы для массивов

Для массивов в C++ также имеется поддержка итераторов. Для этого в стандартной библиотеке C++ определены функции **std::begin()** (возвращает итератор на начало массива) и **std::end()** (возвращает итератор на конец массива):

```
int data[]{4, 5, 6, 7, 8};
// получаем итератор на начало массива
auto iter = std::begin(data);
// получаем итератор на конец массива
auto end = std::end(data);
```

Как и контейнеры, массив можно перебрать с помощью итераторов:

```
#include <iostream>

int main()
{
    int data[]{4, 5, 6, 7, 8};
    // перебор массива с помощью итераторов
    for(auto iter {std::begin(data)}; iter != std::end(data); iter++)
    {
        std::cout << *iter << std::endl;
    }
}
```

Но перебор массива вполне можно сделать и другими способами - через индексы, обычные указатели. Но итераторы на массивы могут быть полезны при манипуляции с контейнерами. Например, функция **insert()**, которая есть у ряда контейнеров, позволяет добавить в контейнер какую-то часть другого контейнера. Для выделения добавляемой части могут применяться итераторы. И таким образом, с помощью итераторов можно добавить в контейнер, например, в вектор какую-то часть контейнера:

```
#include <iostream>
#include <vector>

int main()
{
    int data[]{4, 5, 6, 7, 8};
    std::vector<int> numbers { 1, 2, 3, 4};
    // добавляем в конец вектора numbers из массива data элементы со 2-го по
```

```
предпоследний (включительно)
    numbers.insert(numbers.end(), std::begin(data) + 1, std::end(data)-1);
    for (auto iter {numbers.begin()}; iter != numbers.end(); ++iter)
    {
        std::cout << *iter << "\t";
    }
    std::cout << std::endl;
}
```

Здесь строка

```
numbers.insert(numbers.end(), std::begin(data) + 1, std::end(data)-1);
```

Добавляет в вектор numbers, начиная с позиции, на которую указывает итератор `numbers.end()` (то есть в самый конец вектора), диапазон элементов массива data. Начало этого диапазона задается выражением `std::begin(data) + 1` (то есть со 2-го элемента), а конуц - выражением `std::end(data)-1` (то есть по предпоследний элемент включительно).

Добавление элементов в вектор

Для добавления элементов в вектор применяется функция **push_back()**, в которую передается добавляемый элемент:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> numbers;     // пустой вектор
    numbers.push_back(5);
    numbers.push_back(3);
    numbers.push_back(10);
    for(int n : numbers)
        cout << n << "\t";      // 5    3   10

    std::cout << std::endl;
}
```

Векторы являются динамическими структурами в отличие от массивов, где мы скованы его заданым размером. Поэтому мы можем динамически добавлять в вектор новые данные.

Функция **emplace_back()** выполняет аналогичную задачу - добавляет элемент в конец контейнера:

```
std::vector<int> numbers{ 1, 2, 3, 4, 5 };
numbers.emplace_back(8);    // numbers = { 1, 2, 3, 4, 5, 8 };
```

Добавление элементов на определенную позицию

Ряд функций позволяет добавлять элементы на определенную позицию.

- **emplace(pos, value)**: вставляет элемент value на позицию, на которую указывает итератор pos
- **insert(pos, value)**: вставляет элемент value на позицию, на которую указывает итератор pos, аналогично функции emplace
- **insert(pos, n, value)**: вставляет n элементов value начиная с позиции, на которую указывает итератор pos
- **insert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end
- **insert(pos, values)**: вставляет список значений начиная с позиции, на которую указывает итератор pos

Функция emplace:

```
std::vector<int> numbers{ 1, 2, 3, 4, 5 };
auto iter = numbers.cbegin();   // константный итератор указывает на первый элемент
numbers.emplace(iter + 2, 8); // добавляем после второго элемента  numbers = { 1, 2, 8, 3, 4, 5};
```

Функция insert:

```
std::vector<int> numbers1{ 1, 2, 3, 4, 5 };
auto iter1 = numbers1.cbegin(); // константный итератор указывает на первый элемент
numbers1.insert(iter1 + 2, 8); // добавляем после второго элемента
//numbers1 = { 1, 2, 8, 3, 4, 5};

std::vector<int> numbers2 { 1, 2, 3, 4, 5 };
auto iter2 = numbers2.cbegin(); // константный итератор указывает на первый элемент
numbers2.insert(iter2 + 1, 3, 4); // добавляем после первого элемента три четверки
//numbers2 = { 1, 4, 4, 4, 2, 3, 4, 5};

std::vector<int> values { 10, 20, 30, 40, 50 };
std::vector<int> numbers3 { 1, 2, 3, 4, 5 };
auto iter3 = numbers3.cbegin(); // константный итератор указывает на первый элемент
// добавляем после первого элемента три первых элемента из вектора values
numbers3.insert(iter3 + 1, values.begin(), values.begin() + 3);
//numbers3 = { 1, 10, 20, 30, 2, 3, 4, 5};

std::vector<int> numbers4 { 1, 2, 3, 4, 5 };
auto iter4 = numbers4.cend();   // константный итератор указывает на позицию за последним элементом
// добавляем в конец вектора numbers4 элементы из списка { 21, 22, 23 }
numbers4.insert(iter4, { 21, 22, 23 });
//numbers4 = { 1, 2, 3, 4, 5, 21, 22, 23};


```

Удаление элементов

Если необходимо удалить все элементы вектора, то можно использовать функцию **clear**:

```
std::vector<int> v { 1,2,3,4 };
v.clear();
```

Функция **pop_back()** удаляет последний элемент вектора:

```
std::vector<int> v { 1,2,3,4 };
v.pop_back();       // v = { 1,2,3 }
```

Если нужно удалить элемент из середины или начала контейнера, применяется функция **std::erase()**, которая имеет следующие формы:

- **erase(p)**: удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент
- **erase(begin, end)**: удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

Применение функции:

```
std::vector<int> numbers1 { 1, 2, 3, 4, 5, 6 };
auto iter = numbers1.cbegin(); // указатель на первый элемент
numbers1.erase(iter + 2);   // удаляем третий элемент
// numbers1 = { 1, 2, 4, 5, 6 }

std::vector<int> numbers2 = { 1, 2, 3, 4, 5, 6 };
auto begin = numbers2.cbegin(); // указатель на первый элемент
auto end = numbers2.cend();     // указатель на последний элемент
numbers2.erase(begin + 2, end - 1); // удаляем с третьего элемента до последнего
// numbers2 = {1, 2, 6}


```

Также начиная со стандарта C++20 в язык была добавлена функция **std::erase()**. Она не является частью типа vector. В качестве первого параметра она принимает вектор, а в качестве второго - элемент, который надо удалить:

```
std::vector<int> numbers3 { 1, 2, 3, 1, 5, 6 };
std::erase(numbers3, 1); // numbers3 = { 2, 3, 4, 5, 6 }
```

В данном случае удаляем из вектора numbers3 все вхождения числа 1.

Размер вектора

С помощью функции **size()** можно узнать размер вектора, а с помощью функции **empty()** проверить, путой ли вектор:

```
#include <iostream>
#include <vector>

int main()
{

```

```
    std::vector<int> numbers{1, 2, 3};
    if(numbers.empty())
        std::cout << "Vector is empty" << std::endl;
    else
        std::cout << "Vector has size " << numbers.size() << std::endl;
}

```

С помощью функции **resize()** можно изменить размер вектора. Эта функция имеет две формы:

- **resize(n)**: оставляет в векторе n первых элементов. Если вектор содержит больше элементов, то его размер усекается до n элементов. Если размер вектора меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию
- **resize(n, value)**: также оставляет в векторе n первых элементов. Если размер вектора меньше n, то добавляются недостающие элементы со значением value

Применение функции:

```
std::vector<int> numbers1 { 1, 2, 3, 4, 5, 6 };
numbers1.resize(4); // оставляем первые четыре элемента - numbers1 = {1, 2, 3, 4}

numbers1.resize(6, 8);   // numbers1 = {1, 2, 3, 4, 8, 8}
```

Важно учитывать, что применение функции resize может сделать некорректными все итераторы, указатели и ссылки на элементы.

Изменение элементов вектора

Функция **assign()** позволяет заменить все элементы вектора определенным набором:

```
std::vector<std::string> langs = { "Java", "JavaScript", "C"};
langs.assign(4, "C++"); // langs = {"C++", "C++", "C++", "C++"}
```

В данном случае элементы вектора заменяются набором из четырех строк "C++".

Также можно передать непосредственно набор значений, который заменит значения вектора:

```
std::vector<std::string> langs{ "Java", "JavaScript", "C"};
langs.assign({ "C++", "C#", "C"}); // langs = { "C++", "C#", "C"}
```

Еще одна функция - **swap()** обменивает значения двух контейнеров:

```
std::vector<std::string> clangs { "C++", "C#", "Java" };
std::vector<std::string> ilangs { "JavaScript", "Python", "PHP"};
clangs.swap(ilangs);    // clangs = { "JavaScript", "Python", "PHP"};
for(std::string lang : clangs)
{
    std::cout << lang <<  std::endl;
}
```

Сравнение векторов

Векторы можно сравнивать - они поддерживают все операции сравнения: <, >, <=, >=, ==, !=. Сравнение контейнеров осуществляется на основании сравнения пар элементов на тех же позициях. Векторы равны, если они содержат одинаковые элементы на тех же позициях. Иначе они не равны:

```
std::vector<int> v1 {1, 2, 3};
std::vector<int> v2 {1, 2, 3};
std::vector<int> v3 {3, 2, 1};

bool v1v2 = v1 == v2;   // true
bool v1v3 = v1 != v3;   // true
bool v2v3 = v2 == v3;   // false
```

Контейнер **array** из одноименного модуля `<array>` представляет аналог массива. Он также имеет фиксированный размер.

Определение и инициализация

Для создания объекта **array** в угловых скобках после названия типа необходимо передать его тип и размер:

```
#include <array>

int main()
{
    std::array<int, 5> numbers;       // состоит из 5 чисел int
}
```

В данном случае определен объект array из 5 чисел типа int. По умолчанию все элементы контейнера имеют неопределенные значения.

Чтобы инициализировать контейнер определенными значениями, можно использовать инициализатор - в фигурных скобках передать значения элементам контейнера:

```
std::array<int, 5> numbers {};    // состоит из 5 нулей
```

В данном случае пустой инициализатор инициализирует все элементы контейнера numbers нулями. Также можно указать конкретные значения для элементов:

```
std::array<int, 5> numbers {2, 3, 4, 5, 6};
```

Фиксированный размер накладывает ограничение на инициализацию: количество передаваемых контейнеру элементов не должно превышать его размер. Можно передать меньше значений, которые будут переданы первым элементам контейнера, а остальные элементы получат значения по умолчанию (например, для целочисленных типов это число 0):

```
std::array<int, 5> numbers {2, 3, 4};      // {2, 3, 4, 0, 0}
```

Однако если при инициализации мы предадим большее количество элементов, нежели размер контейнера, то мы столкнемся с ошибкой.

Стоит отметить, что начиная со стандарта C++17 при инициализации можно не указывать тип и количество элементов - компилятор выводит это автоматически исходя из списка инициализации:

```
std::array numbers {2, 3, 4, 5, 6};
```

Однако в этом случае в списке инициализации в фигурных скобках должно быть как минимум одно значение.

Доступ к элементам

Для доступа к элементам контейнера array можно применять тот же синтаксис, что при работе с массивами - в квадратных скобках указывать индекс элемента, к которому идет обращение:

```
#include <array>
#include <iostream>

int main()
{
    std::array<int, 5> numbers {2, 3, 4, 5, 6};
    // получаем значение элемента
    int n = numbers[2];
    std::cout << "n = " << n << std::endl; // n = 4
    // меняем значение элемента
    numbers[2] = 12;
    std::cout << "numbers[2] = " << numbers[2] << std::endl; // numbers[2] = 12
}
```

Перебор контейнера

С помощью стандартных циклов можно перебрать контейнер array:

```
#include <iostream>
#include <array>
#include <string>

int main()
{
    const unsigned n = 5;
    std::array<std::string, n> people { "Tom", "Alice", "Kate", "Bob", "Sam" };

    // обращение через индексы
    for(int i{}; i < n; i++)
    {
        std::cout << people[i] << std::endl;
    }
    std::cout << std::endl;
    // перебор последовательности
```

```
    for (auto person : people)
    {
        std::cout << person << std::endl;
    }
}
```

Основные функции array

В контейнер array нельзя добавлять новые элементы, так же как и удалять уже имеющиеся. Основные функции типа array, которые мы можем использовать:

- **size()**: возвращает размер контейнера
- **at(index)**: возвращает элемент по индексу index
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент
- **fill(n)**: присваивает всем элементам контейнера значение n

Применение методов:

```
#include <iostream>
#include <array>
#include <string>

int main()
{
    std::array<std::string, 3> people { "Tom", "Bob", "Sam" };
    std::string second = people.at(1);    // Bob
    std::string first = people.front();  // Tom
    std::string last = people.back();    // Sam

    std::cout << second << std::endl;   // Bob
    std::cout << first << std::endl;    // Tom
    std::cout << last << std::endl;     // Sam

    // присваиваем всем элементам "Undefined"
    people.fill("Undefined");  // people = { "Undefined", "Undefined", "Undefined" }
    // проверяем
    for (int i{}; i< people.size(); i++)
    {
        std::cout << people[i] << std::endl;
    }
}
```

Несмотря на то, что объекты array похожи на обычные массивы, тип array более гибок.

Например, мы не можем присваивать одному массиву напрямую значения второго массива. В то же время объекту array мы можем передавать данные другого объекта array:

```
std::array<int, 5> numbers1 { 1, 2, 3, 4, 5 };
```

```
std::array<int, 5> numbers2 = numbers1;        // так можно сделать

int nums1[] = { 1,2,3,4,5 };
//int nums2[] = nums1;       // так нельзя следать
```

Также мы можем сравнивать два контейнера array:

Два контейнера сравниваются поэлементно.