Современные технологии разработки ПО

Обобщённые контейнеры C++

Standart Template Library (STL)

- STL была создана как первая библиотека универсальных алгоритмов и структур данных для C++ в 1994 году
- Основана на идеях:
  - обобщённого программирования,
  - абстрагирования без потери эффективности,
  - вычислительной модели фон Неймана,
  - семантики величин (value semantics).
- Архитектура STL была разработана Александром Степановым с участием Менг Ли

Библиотека контейнеров C++ *

- Является универсальным набором:
  - шаблонов классов,
  - итераторов,
  - алгоритмов.
- Позволяет реализовать общие структуры данных для задаваемых типов: очереди, списки, стеки и т.д.

Свойства контейнеров *

- Имеют похожие свойства, реализованные в методах с одинаковыми наименованиями, например:
  - begin, end (cbegin, cend, rbegin, rend);
  - size, empty;
  - clear, erase;
  - insert и т. д.
- Специальные свойства имеют отличные наименования, например:
  - push_back для vector, deque и list;
  - marge для list;
  - и т.д.

Виды контейнеров

- Виды контенеров:
  - последовательные контейнеры,
  - ассоциативные контейнеры,
  - неупорядоченные ассоциативные контейнеры.
- Каждый предназначен для поддержки различных наборов операций

Определение обобщённого контейнера *

- Контейнер управляет выделяемой для его элементов памятью и предоставляет функции-члены для доступа к ним, либо непосредственного, либо через итераторы (объекты, обладающие схожими с указателями свойствами)

Выбор контейнера *

- Большинство контейнеров обладают по крайней мере несколькими общими функциями-членами и общей функциональностью
- Выбор оптимального контейнера для конкретного случая зависит не только от предоставляемой функциональности, но и от его эффективности при различных рабочих нагрузках

Последовательные контейнеры

- Последовательные контейнеры реализуют структуры данных с возможностью последовательного доступа к ним:
  - **`array<T>`** — статический непрерывный массив;
  - **`vector<T>`** — динамический непрерывный массив;
  - **`deque<T>`** — двусторонняя очередь;
  - **`forward_list<T>`** — односвязный список;
  - **`list<T>`** — двусвязный список.

Ассоциативные контейнеры

- Ассоциативные контейнеры реализуют упорядоченные структуры данных с возможностью быстрого поиска (со сложностью O(log n)):
  - **`set<K>`** — коллекция уникальных ключей, отсортированная по ключам;
  - **`map<K,T>`** — коллекция пар ключ-значение, отсортированная по ключам, ключи являются уникальными;
  - **`multiset<K>`** — коллекция ключей, отсортированная по ключам;
  - **`multimap<K,T>`** — коллекция пар ключ-значение, отсортированная по ключам.

Неупорядоченные ассоциативные контейнеры

- Неупорядоченные ассоциативные контейнеры реализуют неупорядоченные (хешированные) структуры данных с возможностью быстрого поиска (со средней сложностью O(1), в худшем случае O(n)):
  - **`unordered_set<K>`** — коллекция уникальных ключей, хешируется по ключам, ключи являются уникальными;
  - **`unordered_map<K,T>`** — коллекция пар ключ-значение, хешируется по ключам, ключи являются уникальными;
  - **`unordered_multiset<K>`** — коллекция ключей, хешируется по ключам;
  - **`unordered_multimap<K,T>`** — коллекция пар ключ-значение, хешируется по ключам.

Адаптеры контейнеров

- Адаптеры контейнеров предоставляют различные интерфейсы для последовательных контейнеров:
  - **`stack<T>`** — адаптер с интерфейсом стека (LIFO);
  - **`queue<T>`** — адаптер с интерфейсом очереди (FIFO);
  - **`priority_queue<T>`** — адаптер с интерфейсом очереди с приоритетом.

Псевдоконтейнеры

- Псевдоконтейнеры не реализуют большинство стандартных для контейнеров операций и сильно специализированы для своих целей:
  - **`bitset<N>`** — хранение битовых масок;
  - **`basic_string<T>`** — хранение и обработка строк;
  - **`valarray<T>`** — хранение числовых массивов, оптимизированое для достижения повышенной вычислительной производительности.

Итераторы

- Обобщённая абстракция, используемая для доступа к элементам контейнеров в качестве посредника
- Каждый контейнер поддерживает свою реализацию итератора, который представляет собой адаптированный интеллектуальный указатель, знающий как получить доступ к элементам конкретного контейнера
- Заголовочный файл <iterator>

Пример работы с итератором

```cpp
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
    std::vector<int> v = { 3, 1, 4 };
    auto vi = std::begin(v);
    std::cout << *vi << std::endl;

    auto nx = std::next(vi, 2);
    std::cout << *nx << std::endl;
}
```

```
3
4
```

Обобщённые алгоритмы

- Обощённые функции для различных целей (поиска, сортировки, подсчета, манипулирования и др.), оперирующие над диапазонами элементов
- Диапазон определяется как [first, last), где last относится к элементу, следующему за последним просматриваемым или изменяемым элементом
- Заголовочный файл <algorithm>

Классификация алгоритмов

- Немодифицирующие операции над контейнерами
- Модифицирующие операции над контейнерами
- Операции разделения
- Операции сортировки (на отсортированных диапазонах)
- Операции двоичного поиска (на отсортированных диапазонах)
- Операции над множествами (на отсортированных диапазонах)
- Операции над кучей
- Операции минимума/максимума
- Операции сравнения
- Операции перестановки
- Числовые операции (заголовочный файл <numeric>)

алгоритмов

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

struct Sum
{
    Sum(): sum{0} { }
    void operator()(int n) { sum += n; }
    int sum;
};

int main()
{
    std::vector<int> nums{3, 4, 2, 8, 15, 267};

    auto print = [](const int& n) { std::cout << " " << n; };

    std::cout << "before:";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << '\n';

    std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });

    // calls Sum::operator() for each number
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());

    std::cout << "after: ";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << '\n';
    std::cout << "sum: " << s.sum << '\n';
}
```

```
before: 3 4 2 8 15 267
after:  4 5 3 9 16 268
sum: 305
```

std::array<T,N> *

- Контейнер, инкапсулирующий массив фиксированного размера
- Имеет ту же семантику, что и C-массивы
- Размер и эффективность array<T,N> такие же, как у C-массива T[N]
- array предоставляет некоторые возможности стандартных контейнеров, такие как знание собственного размера, поддержка присваивания, итераторы произвольного доступа и т.д.

std::array<T,N>
Пример

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

int main()
{
    // конструктор использует агрегатный инициализатор
    std::array<int, 3> a1{ {1,2,3} };  // требуются двойные фигурные скобки,
    std::array<int, 3> a2 = {1, 2, 3}; // за исключением операций присваивания
    std::array<std::string, 2> a3 = { {std::string("a"), "b"} };

    // поддерживаются обобщённые алгоритмы
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

    // поддерживается ranged for цикл
    for(auto& s: a3)
        std::cout << s << ' ';
}
```

std::vector<T> *

- Последовательный контейнер, инкапсулирующий массивы переменного размера
  - (с автоматическим изменением размера при добавлении/удалении элемента)

std::vector<T>
Работа с памятью *

begin
end
capacity

vector<T> v;

nullptr
nullptr
0

v.push_back(T);

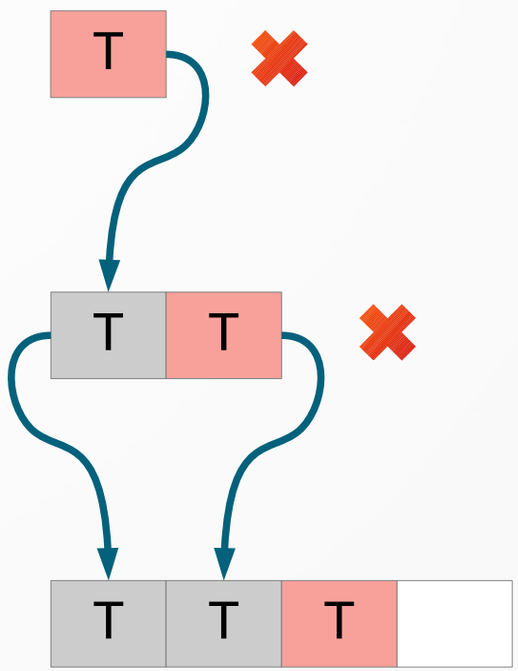
&data
&data[1]
1

v.push_back(T);

&data
&data[2]
2

v.push_back(T);

&data
&data[3]
4



size = (end - begin) / size of T

std::vector<T>
Вычислительная сложность операций

- Перераспределения обычно являются дорогостоящими операциями в плане производительности. Функция reserve() может использоваться для предварительного выделения памяти и устранения перераспределений, если заранее известно количество элементов.
- Доступ по индексу за O(1)
- Добавление-удаление элемента в конец vector занимает амортизированное O(1) время, та же операция в начале или середине vector — O(n)
- Стандартная быстрая сортировка за O(n log(n))
- Поиск элемента перебором занимает O(n)

std::vector<T>
Пример

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

int main ( ) {
    // Создание вектора, содержащего целые числа
    std::vector<int> v = {7, 5, 16, 8};

    // Добавление ещё двух целых чисел в вектор
    v.push_back(25);
    v.push_back(13);

    // Проход по вектору с выводом значений
    for ( int n : v ) {
        std::cout << n << '\n';
    }
}
```

```
7
5
16
8
25
13
```

std::list<T> *

- Двусвязный список, элементы которого хранятся в произвольных частях памяти
  - (в отличие от контейнера vector, где элементы хранятся в непрерывной области памяти)
- Поиск перебором медленнее, чем у вектора из-за большего времени доступа к элементу
- Доступ по индексу за O(n). В любом месте контейнера вставка и удаление производятся очень быстро — за O(1)

std::list<T>
Пример

```cpp
#include <list>
#include <string>

int main()
{
    // c++11 initializer list syntax:
    std::list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};

    // words2 == words1
    std::list<std::string> words2(words1.begin(), words1.end());

    // words3 == words1
    std::list<std::string> words3(words1);

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::list<std::string> words4(5, "Mo");

    return 0;
}
```

std::deque<T>

- Двухсторонняя очередь
- Контейнер похож на vector, но с возможностью быстрой вставки и удаления элементов на обоих концах за O(1)
- Реализован в виде двусвязанного списка линейных массивов
- В отличие от vector, двухсторонняя очередь не гарантирует расположение всех своих элементов в непрерывном участке памяти

std::set<K> *

- Упорядоченное множество уникальных элементов
- При вставке/удалении элементов множества итераторы, указывающие на элементы этого множества, не становятся недействительными
- Обеспечивает стандартные операции над множествами типа объединения, пересечения, вычитания
- Тип элементов множества должен реализовывать оператор сравнения operator<, или требуется предоставить функцию-компаратор
- Реализован на основе самобалансирующего дерева двоичного поиска

std::multiset<K> *

- Коллекция ключей, отсортированная по ключам
- То же, что и set, но позволяет хранить повторяющиеся элементы

std::map<K,T> *

- Упорядоченный ассоциативный массив пар элементов, состоящих из ключей и соответствующих им значений
- Ключи должны быть уникальны
- Порядок следования элементов определяется ключами
- При этом тип ключа должен реализовывать оператор сравнения operator<, либо требуется предоставить функцию-компаратор

std::map<T>
Пример

```
#include <string>
#include <iostream>
#include <map>

int main()
{
    typedef std::map<std::string,int>  mapT;

    mapT my_map;
    my_map["first"]=  11;
    my_map["second"]= 23;

    mapT::iterator  it= my_map.find("first");
    if( it != my_map.end() ) std::cout << "A: " << it->second << "\n";

    it= my_map.find("third");
    if( it != my_map.end() ) std::cout << "B: " << it->second << "\n";

    // Accessing a non-existing element creates it
    if( my_map["third"] == 42 ) std::cout << "Oha!\n";

    it= my_map.find("third");
    if( it != my_map.end() ) std::cout << "C: " << it->second << "\n";

    return 0;
}
```

```
A: 11
C: 0
```

std::multimap<K,T> *

- Коллекция пар ключ-значение, отсортированная по ключам
- То же, что и map, но позволяет хранить несколько одинаковых ключей

Лямбда-функции и выражения

- [captures] (params) specifiers exception attr -> ret { body }
- [captures] (params) -> ret { body }
- [captures] (params) { body }
- [captures] { body }

Лямбда-функции и выражения
Пример

```cpp
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    std::vector<int> nums{3, 4, 2, 8, 15, 267};

    auto print = [](const int& n) { std::cout << " " << n; };

    std::cout << "before:";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << std::endl;

    std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });

    std::cout << "after: ";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << std::endl;
}
```

Синтаксис C++ для управления захватом лямбды

- [] Ничего не захватывается
- [&] Захват всего по ссылке
- [=] Захват всего по значению
- [&ctr] Захват только ctr по ссылке
- [ctr] Захват только ctr по значению
- [&,ctr] Захват ctr по значению, а всего остального — по ссылке
- [=,&v] Захват v по ссылке, а всего остального — по значению
- [&, ctr1, ctr2] Захват ctr1 и ctr2 по значению, а всего остального — по ссылке

Вопросы?


MSKU 365 197 4 22G1
TTNU 308854 4 22G1
GATU 080383 0 22G1
PONU 205769
TTNU 136097 2 22G1
GESU 291470 9
PONU 017163 2 22G1
PONU 038518 8 22G1
MSKU 269526 3 22G1
CAXU 8 17 22 27
MSKU 295545 8 22G1
PONU 201390 7
MAERSK SEALAND
© photohome.co.uk