Языки программирования
Лекция 8

ПМИ Семестр 2

Демяненко Я.М.

2024

Коллекции и итераторы

Коллекция или контейнер — объект программы, содержащий в себе набор значений одного или различных типов и позволяющий обращаться к этим значениям.

Итератор – объект, позволяющий программисту перебирать все элементы коллекции без учета особенностей ее реализации.

Коллекции

Назначение коллекции — служить хранилищем программных объектов и обеспечивать доступ к ним.

Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке.

Для обращения к конкретному элементу коллекции могут использоваться различные функции, в зависимости от ее логической организации.

Реализация может допускать выполнение отдельных операций над коллекциями в целом.
Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

Продолжите фразу

Примерами коллекций являются классы для реализации …

В простейшем случае итератором в низкоуровневых языках является …

Операцию индексирования … можно ли назвать итератором?

Счётчик цикла … можно ли назвать итератором?

Ответы

Примерами коллекций являются классы для реализации … массива, строки, списка.

В простейшем случае итератором в низкоуровневых языках является … указатель.

Операцию индексирования можно считать примитивной формой итератора.

Счётчик цикла иногда называют итератором цикла.
Тем не менее, счётчик цикла обеспечивает только перебор элементов, но не доступ к элементу.

Использование итераторов

Использование итераторов в обобщённом программировании позволяет реализовать универсальные алгоритмы работы с контейнерами.

Примерами коллекций (контейнеров), по которым может перемещаться итератор, могут быть:

- список,
- очередь,
- множество,
- ассоциативный массив
- и др.

Итератор и указатель

Итератор похож на указатель своими основными операциями:
- указание одного отдельного элемента в коллекции объектов (доступ к элементу);
- изменение своего значения так, чтобы указывать на следующий элемент (перебор элементов).

Также должны быть определены
- способ создания итератора, указывающего на первый элемент коллекции;
- способ узнать, достигнут ли конец коллекции.

В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Предназначение итераторов

Главное предназначение итераторов заключается в предоставлении возможности пользователю **обращаться к** любому **элементу контейнера при сокрытии** внутренней **структуры контейнера** от пользователя.

Это позволяет контейнеру **хранить элементы любым способом** при допустимости **работы** пользователя с ним **как с простой послед**овательностью или списком.

Проектирование класса итератора тесно связано с соответствующим классом контейнера.

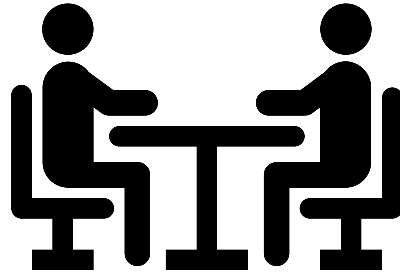
Обычно контейнер предоставляет функции создания итераторов.

Возможности итераторов

- Указание одного отдельного элемента в коллекции объектов (доступ к элементу)
- Изменение своего значения так, чтобы указывать на следующий/предыдущий элемент (перебор элементов)
- Могут поддерживать дополнительные операции или определять различные варианты поведения

Встроенные итераторы и итераторы-объекты

Плюсы



Минусы

Встроенные итераторы

```
struct el {
   int item;
   el* next;
};

class list {
private:
   el* head
   el* tail;
   el * cur;
. . .
};
```

В тех случаях, когда коллекция представляет собой структуру, основанную на указателях, итератор может быть реализован в виде дополнительного поля – текущего указателя.

НЕДОСТАТКИ

- В этом случае следует заботиться о его состоянии при любых модификациях коллекции, даже если его не придётся использовать.
- Кроме того, такой встроенный итератор обычно только один, а могут возникнуть задачи, в которых потребуется наличие двух, трёх или даже большего количества итераторов.

Итераторы-объекты

Итераторы-объекты имеют **преимущество** по сравнению со встроенным текущим указателем, так как позволяют создавать для одной коллекции **любое количество** итераторов, действующих независимо.

Таким образом, итератор позволяет **вынести** рабочий **указатель** за **пределы коллекции**, но при этом **дает возможность перемещаться** по объектам, содержащимся в коллекции, аналогично тому, как текущий указатель позволяет перемещаться внутри коллекции.

Разновидности итераторов

- однонаправленные;
- обратные (реверсные);
- двунаправленные итераторы;
- итераторы ввода и вывода;
- константные итераторы (защищающие контейнер или его элементы от изменения).

Перегруженные операции для итераторов

- Операция доступа к элементу коллекции по ссылке. Такая операция может быть реализована путём перегрузки операции разыменования *

- Операции перемещения вперёд и назад по коллекции объектов. Чаще всего для этого перегружаются операции ++ и – –

- Операции = = и ! = обычно перегружаются для проверки равенства итераторов

Другие операции над коллекцией

Через итератор могут быть реализованы другие операции над коллекцией, например:

- вставка элемента после позиции итератора
- удаление элемента в позиции итератора

При реализации таких операций очень важно оговорить правила поведения итератора после выполнения операции.

Функции в классе коллекции

В классе коллекции для использования итератора создаются две функции:

- iterator begin() – инициализация итератора ссылкой на первый элемент коллекции;
- iterator end() – значение, сообщающее, что итератор достиг конца коллекции.

Или это могут быть внешние функции

Аналогия с диапазонами массивов !!!

Пример. Линейный двусвязный список и итератор для него с возможностью перемещения в двух направлениях

```
struct node{
  int data;
  node* next;
  node* prev;
  node(int data, node* next, node* prev) {
    this->data = data;
    this->next = next;
    this->prev = prev;
  }
};


ostream & operator<<(ostream & out, const node& X) {
  out << X.data;
  return out;
}
```

class **listIterator**; //**предварительное** объявление класса итератора

```
class List{
public:
  class Error {
  public:
    void what(){ cout << "List is empty" << endl; }
  };
  List() { head = nullptr; tail = nullptr; }
  List(const List& l);
  ~List();
  bool isEmpty() const;
  void inHead(int val);
  void inTail(int val);
  int getFirst()const;
  int getLast()const;
  void delFirst();
  void delLast();
  listIterator begin() const;
  listIterator end() const;
  friend ostream& operator<<(ostream & os, const List& l);
  //в полной реализации могут быть еще методы
private:
  node* head;
  node* tail;
  Error err;
};
```

```
List::List(const List& l) {
  head = nullptr; tail = nullptr;
  node* q = l.head;
  while (q)   {
    inTail(q->data);
    q = q->next;
  }
}
List::~List(){
  while (head){
    node* cur = head;
    head = head->next;
    delete cur;
  }
  tail = head = nullptr;
}
bool List::isEmpty() const {
  return (head == nullptr);
}
```

```
void List::inHead(int val){
  node* t = new node(val,head,nullptr);
  if (!head)
    tail = t;
  else
    head->prev = t;
  head = t;
}
void List::inTail(int val){
  node* t = new node(val, nullptr,tail);
  if (head){
    tail->next = t;
    tail = t;
  }
  else{
    head = tail = t;
  }
}
```

```
int List::getFirst() const{
  if (head)
    return head->data;
  else  throw err;
}
int List::getLast()const{
  if (head)
    return tail->data;
  else  throw err;
}
void List::delFirst(){
  if (head){
    node *t = head;
    head = head->next;
    head->prev = nullptr;
    delete t;
  }
  else  throw err;
}
```

```
void List::delLast(){
  if (head){  if (head == tail) {
        delete tail; head = nullptr; tail=nullptr
  }
    else {
      node *t = head;
      while (t->next != tail)   t = t->next;
      delete tail;
      tail = t;
      t->next = nullptr;
    }
  }
  else  throw err;
}

ostream& operator<<(ostream & os, const List& l){
  node *p = l.head;
  while (p) { os << *p << " "; p = p->next; }
  os << endl;
  return os;
}
```

```
class listIterator {
public:
  class Error {
  public:
    void what(){
      cout << "Iterator error" << endl;
    }
  };
private:
  const List *collection;
  node *cur;
public:
  listIterator(const List *s, node *e) :collection(s), cur(e){}
  const int operator *(){
    return cur->data;
  }
  listIterator operator++(); //префиксный ++
  listIterator operator--(); //префиксный --
  int operator == (const listIterator &ri) const;
  int operator != (const listIterator &ri) const;
};
```

```
listIterator listIterator:: operator++() {
  if (cur) {
    cur = cur->next;
    return *this;
  }
  else throw Error();
}

listIterator listIterator:: operator--() {
  if (cur) {
    cur = cur->prev;
    return *this;
  }
  else throw Error();
}
```

```
int listIterator:: operator==(const listIterator &ri) const {
  return ((collection == ri.collection) && (cur == ri.cur));
}
int listIterator:: operator!=(const listIterator &ri) const {
  return !(*this == ri);
}
```

```
listIterator List::begin() const {
  return listIterator(this, head);
}
listIterator List::end() const {
  listIterator iter(this, nullptr);
  return iter;
}
```

Функцию end() нельзя заменить просто указателем с nullptr, так как в операциях == и != для итератора прежде всего проверяется, относятся ли два итератора к одному и тому же списку.

Нахождение суммы элементов списка, расположенных между двумя итераторами

```
int sum(listIterator b, listIterator e) {
  int sum = 0;
  while (b != e){
    sum+= *b;
    ++b;
  }
  return sum;
}
```

Найдите отличия

```
int sum(listIterator b, listIterator e) {
  int sum = 0;
  while (b != e){
    sum+= *b;
    ++b;
  }
  return sum;
}
```

```
int sum(int* b, int* e) {
  int sum = 0;
  while (b != e){
    sum+= *b;
    ++b;
  }
  return sum;
}
```

Нахождение итератора, ссылающегося на максимальный элемент списка

```
listIterator max(listIterator b, listIterator e) {
  listIterator maxx = b;
  while (b != e){
    if ( *b > *maxx )
      maxx = b;
    ++b;
  }
  return maxx;
}
```

Вывод в обратном порядке элементов списка, расположенных между двумя итераторами

```
void reverseprint(listIterator b, listIterator be){
  while (be!=b) {
    cout << *(--be) << ' ';
  }
  cout  << endl;
}
```

```
void reverseprint(listIterator b, listIterator be){
  --be;
  while (be!=b) {
      cout << *be << ' ‘;
  --be;
  }
  cout  << endl;
}
```

Есть ли верные варианты? Если да, то какой или какие?

```
int main(){
  List l1;
  for (int i = 0; i < 5;++i)
    l1.inHead(i);
  for (int i = 5; i < 10; ++i)
    l1.inTail(i);
  List l2(l1);
  cout << " list \n" << l2 << endl;
  cout << sum(l2.begin(), l2.end())<<endl;
  listIterator mt = max(l2.begin(), l2.end());
  cout << *mt << endl;
  reverseprint(l2.begin(),mt );
  //reverseprint(mt,l2.begin());
  //reverseprint(mt, l2.end());
  return 0;
}
```

Пример. Создать шаблон класса matrix для представления матрицы как вектора, элементами которого являются векторы

На основе класса myvector создадим шаблон класса myvector<T>

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
  myvector(int s = 1): size(s){
    vect = new T[s];
  }

  myvector(const myvector<T> & v): size(v.size){
    vect = new T[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }
```

```
  myvector(myvector<T>&& v){
    size = v.size;
    vect= v.vect;
    v.vect = nullptr;
  }

  ~myvector(){
    if (vect != nullptr)
    delete[] vect;
  }
```

```
T& operator [] (int i){
  return vect[i];
 }

 T operator [] (int i) const {
  return vect[i];
 }

 myvector<T>& operator= (const myvector<T> &v){
  if (&v != this) {
    delete[] vect;
    vect = new T[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
    size = v.size;
  }
  return *this;
 }
```

```
myvector<T>& operator=(myvector<T>&& v){
    if (vect != nullptr)
      delete[] vect;
    size = v.size;
    vect = v.vect;
    v.vect = nullptr;
    return *this;
  }

  myvector<T> operator+(const myvector<T>& v){
    if (size == v.size) {
      myvector<T> v1(size);
      for (int i = 0; i < size; ++i)
        v1[i] = vect[i] + v[i];
      return v1;
    }
    return *this;//лучше исключение использовать
  }
};
```

Попробуем объявить матрицу как вектор векторов

```
myvector< myvector<int> > m(3);
```

Какие подводные камни возможны?

Задание размерностей

```
myvector<myvector<int>> m(3);
```

Можно задать только одну размерность — количество строк, поскольку негде указать количество столбцов.

Это связано с тем, что myvector<int> — это параметр типа для шаблонного класса myvector<myvector<int>>.

При этом для каждого элемента m[i] будет вызван конструктор без параметров, в нашей реализации для него задано значение размера массива по умолчанию 1.

Решение проблемы размерностей

Чтобы изменить количество столбцов в матрице, необходимо для каждой строки выполнить функцию изменения размера resize.

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
...
```

```
void resize(int n){
   T* newVect = new T[n];
   int sz = (n < size) ? n: size;
   for (int i = 0; i<sz; ++i)
      newVect[i] = vect[i];
   delete[] vect;
   vect = newVect;
   size = n;
 }
};
```

Шаблон класса matrix

Используем эту функцию в конструкторе шаблона класса матрицы на базе шаблона класса вектора

```
template<typename T>
class matrix{
  int rows;
  myvector<myvector<T>> mdata;

public:
  matrix(int m=1, int n=1): rows(m), mdata(m){
    for (int i = 0; i < m; i++)
      mdata[i].resize(n);
  }
};
```

Конструктор и деструктор

Вызывать конструктор mdata(m) в теле конструктора matrix уже поздно (уже отработает конструктор по умолчанию), а при объявлении еще рано, поэтому конструктор необходимо вызывать в списке инициализации.

Деструктор для данного класса писать не надо, т.к. достаточно сгенерированного деструктора по умолчанию
~matrix(){}.
Поскольку деструктор по умолчанию вызывает деструкторы для всех полей класса, то будет вызван деструктор шаблона класса
myvector<myvector<T>>

Конструктор копии и operator=

Конструктор копии и operator= для шаблона класса matrix также сгенерируются автоматически и будут работать правильно.

Если в классе **есть подобъект**, который **берёт на себя** функции по выделению и освобождению **динамической памяти**, то определять **конструктор копии и operator= не нужно**.

Они необходимы, только если непосредственно в конструкторе данного класса выделяется динамическая память, а в деструкторе возвращается.

Автоматическая генерация

Автоматически сгенерированный конструктор копии вызывает конструкторы копий для всех своих полей.

Автоматически сгенерированная операция присваивания вызывает операции присваивания для всех своих полей.

Например, сгенерированный конструктор копии будет выглядеть так:

```
matrix(const matrix<T> & mm): mdata(mm.mdata) { }
```

Доступ к элементу матрицы по индексам

Можно реализовать двумя способами

- Перегрузка операции индексации для матрицы с возвращением вектора (по номеру строки). А элемент из этой строки возвращается перегруженной для вектора операцией индексации.

- Перегрузка операции вызова функции() с двумя параметрами.

Перегрузка операции индексации для матрицы с возвращением вектора

```
myvector<T>& operator [] (int i) {
  return mdata[i];
}

myvector<T> operator [] (int i)const {
  return mdata[i];
}
```

Перегрузка операции вызова функции() с двумя параметрами

```
T& operator()(int i, int j){
  return mdata[i][j];
}

 T operator() (int i, int j) const {
  return mdata[i][j];
}
```

Функции-члены, которые генерируются «молча»

Рассмотрим следующий (не очень полезный) класс.

```
class Empty {};
```

На самом деле, такое описание класса эквивалентно следующему:

```
class Empty{
public:
   Empty() {}                              //— конструктор без параметров

   Empty(Empty const &) {}                 //— конструктор копий

   Empty & operator=(Empty const &) {}     //— операция копирующего присваивания

   ~Empty() {}                             //— деструктор
};
```

Немного о синглетонах

На практике иногда **создаваемые неявно функции** могут привнести **нежелательную функциональность**, от которой нужно избавиться.

Например, нужно создать класс, для которого должен существовать **только один экземпляр**. Такие классы называются **синглетонами**.

Для них рекомендуется запретить операцию присваивания и конструктор копирования.

С другой стороны, рекомендуется наряду с другими конструкторами всегда иметь конструктор без параметров, даже если его тело является пустым.

Управление генерацией стандартных конструкторов и функций

```
class A {
private:
    int x;
public:
    A(int i) {…}

    // Данная запись указывает на необходимость сгенерировать конструктор без параметров
    A() = default;

    // Запретить генерацию конструктора копии по умолчанию
    A(const A&) = delete;

    // А так можно запретить генерацию operator=
    A& operator = (const A&) = delete;
}
```