## Лабораторная работа №6

### Стандартная библиотека шаблонов

**Цель работы:** получить практические навыки использования стандартной библиотеки шаблонов при написании кода на языке C++.

#### Краткие теоретические сведения

Стандартная библиотека шаблонов (Standard Template Library, STL) входит в стандартную библиотеку языка C++. В неё включены реализации наиболее часто используемых контейнеров и алгоритмов, что избавляет программистов от рутинного их переписывания. Разработчики STL поставили перед собой сверхзадачу - сделать библиотеку одновременно эффективной и универсальной. STL – один из тех немногих программных продуктов, чье появление было встречено единодушно всеми профессиональными программистами. STL строится на основе шаблонов классов, и поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных.

##### Состав STL

Ядро библиотеки образуют три элемента: **контейнеры**, **алгоритмы** и **итераторы**. **Контейнеры** (containers) – это объекты, предназначенные для хранения других элементов. Например, vector, list, set. **Ассоциативные контейнеры** (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям. В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов. **Алгоритмы** (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера. **Итераторы** (iterators) – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера. С итераторами можно работать так же, как с указателями. К ним можно применить операции \*, инкремента, декремента. Типом итератора объявляется тип iterator, который определен в различных контейнерах. В STL также поддерживаются **обратные итераторы** (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается ещё несколько стандартных компонентов. Главными среди них являются **распределители памяти**, **предикаты** и **функции** **сравнения**. У каждого контейнера имеется определенный для него распределитель памяти (**allocator**), который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса **allocator**. Можно определить собственный распределитель. В некоторых алгоритмах и контейнерах используется функция особого типа, называемая **предикатом**. Предикат может быть унарным и бинарным. Возвращаемое значение: истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов **UnPred**, бинарных – **BinPred**. Тип аргументов соответствует типу хранящихся в контейнере объектов. Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется **функцией сравнения** (comparison function). Функция возвращает истину, если первый элемент меньше второго. Типом функции является тип **Comp**.

Особую роль в STL играют объекты-функции. Объекты-функции − это экземпляры класса, в котором определена операция «круглые скобки» (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется operator (). Пример:

class less{ public: bool operator()(int x, int y){ return x<y; } };

##### Классы-контейнеры.

В STL определены два типа контейнеров: последовательности и ассоциативные. Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться **map** (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером **list**. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди **queue**, очереди с двумя концами **deque**, стека **stack**. По умолчанию пользователь должен использовать **vector**; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров и, в общем случае, со всеми видами источников информации – унифицированным способом ведет к понятию обобщенного программирования. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров. В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

|  |  |
| --- | --- |
| **bitset** | множество битов <bitset.h> |
| **vector** | динамический массив <vector.h> |
| **list** | линейный список <list.h> |
| **deque** | двусторонняя очередь <deque.h> |
| **stack** | стек <stack.h> |
| **queue** | очередь <queue.h> |
| **priority\_queue** | очередь с приоритетом <queue.h> |
| **map** | ассоциативный список для хранения пар ключ/значение <map.h> |
| **multimap** | с каждым ключом связано два или более значений <map.h> |
| **set** | множество <set.h> |
| **multiset** | множество, в котором элемент не обязательно уникален <set.h> |

Итераторы: **begin(), end(), rbegin(), rend()**

Доступ к элементам: **front(), back(), operator[]**(i), **at**(i)

Включение элементов: **insert(**p,x**), insert(**p,n,x**), insert(**p,first,last**), push\_back(**x**)**, **push\_front(**x**)**

Удаление элементов: **pop\_back(), pop\_front(), erase(**p**), erase(**first,last**), clear()**

Другие операции: **size(), empty(), capacity(), reserve(**n**), resize(**n**), swap(**x**), ==**, **!=**, **<**

Операции присваивания: **operator=(**x**), assign(**n,x**), assign(**first,last**),** Ассоциативные операции: **operator[](**k**), find(**k**), lower\_bound(**k**), upper\_bound(**k**), equal\_range(**k**)**

##### Контейнер *vector-*вектор.

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу.

template<class T,class Allocator=allocator<T>>class std::vector{…};

где *T* – тип предназначенных для хранения данных. *Allocator* задает распределитель памяти, который по умолчанию является стандартным.

Пример

vector<int> a; vector<double> x(5); vector<char> c(5,’\*’); vector<int> b(a);

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и ==.

Для класса вектор определены следующие операторы сравнения: ==, <, <=, !=, >, >=. Кроме этого, для класса **vector** определяется оператор индекса []. Новые элементы могут включаться с помощью функций *insert(), push\_back(), resize(), assign().* Существующие элементы могут удаляться с помощью функций *erase(), pop\_back(), resize(), clear().* Доступ к элементам осуществляется с помощью итераторов *begin(), end(), rbegin(), rend().* Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций <algorithm.h>.

Пример

#include<iostream.h>

#include<vector.h>

using namespace std;

void main(){

vector<int> v; int i;

for(i=0;i<10;i++) v.push\_back(i);

for(i=0;i<10;i++)v[i]=v[i]+v[i];

for(i=0;i<v.size();i++) cout<<v[i]<<“ ”; cout<<endl; }

Пример – Доступ к вектору через итератор

#include<iostream.h>

#include<vector.h>

using namespace std;

void main(){

vector<int> v; int i;

for(i=0;i<10;i++)v.push\_back(i); cout<<“size=”<<v.size()<<“\n”;

vector<int>::iterator p=v.begin();

while(p!=v.end()) {cout<<\*p<<” “;p++;}

}

Пример – Вектор содержит объекты пользовательского класса

#include<iostream.h>

#include<vector.h>

#include”student.h”

using namespace std;

void main() {

vector<STUDENT> v(3); int i;

v[0]=STUDENT(“Иванов”,45.9);

v[1]=STUDENT(“Петров”,30.4);

v[0]=STUDENT(“Сидоров”,55.6);

for(i=0;i<3;i++) cout<<v[i]<<“ ”; cout<<endl; //вывод

}

Ассоциативные контейнеры (массивы).

Ассоциативный массив содержит пары значений. Зная одно значение, называемое **ключом** (*key*), мы можем получить доступ к другому, называемому **отображенным значением** (*mapped value*). Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип: *V& operator[](const K&)* возвращает ссылку на V, соответствующий K.

Ассоциативные контейнеры – это обобщение понятия ассоциативного массива. Ассоциативный контейнер **map** − это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер **map** предоставляет двунаправленные итераторы. Ассоциативный контейнер **map** требует, чтобы для типов ключа существовала операция “<”. Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса **map**:

template<class Key,classT,class Comp=less<Key>,

class Allocator=allocator<pair> > class std::map

Определена операция присваивания:

map& operator=(const map&);

Определены следующие операции: ==, <, <=, !=, >, >=.

В **map** хранятся пары ключ/значение в виде объектов типа **pair**.

Пример – Создание пар значений

vector <pair <int, int>> a(5);

for (int i = 0; i < 5; i++) {

int num, bal;

cin >> num >> bal;

a[i] = {num, bal}; // создание пары значение – номер

cout << a[i].first << " " << a[i].second << endl;

}

Создавать пары ключ/значение можно не только с помощью конструкторов класса **pair**, но и с помощью функции **make\_pair**, которая создает объекты типа **pair**, используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера – это ассоциативный поиск при помощи операции индексации ([]).

mapped\_type& operator[](const key\_type& K);

Пример – Контейнер map

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main() {

int n;

string temp, temp2, temp3;

cin >> n;

map<string, string> s;

for(int i=0; i<n; i++){

cin >> temp;

cin >> temp2;

s[temp]=temp2;

}

cin >> temp3;

for(auto now : s){

if (now.first == temp3) cout << now.second;

if (now.second == temp3) cout << now.first;

}

return 0;

}

Множества **set** можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи.

template<classT,class Cmp=less<T>,

class Allocator=allocator<T>>class std::set{…};

Множество, как и ассоциативный массив, требует, чтобы для типа **T** существовала операция “меньше” (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Пример – Контейнер set

#include <iostream>

#include <set>

using namespace std;

int main() {

int n, x;

cin >> n;

set<int> arr;

for(int i=0; i<n; i++){

cin >> x;

arr.insert(x);

}

cout << arr.size();

return 0;

}

**Алгоритмы.**

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в <algorithm.h>. Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

Немодифицирующие операции.

**for\_earch()** выполняет операции для каждого элемента

последовательности

**find()** находит первое вхождение значения в последовательность

**find\_if()** находит первое соответствие предикату в последовательности

**count()** подсчитывает количество вхождений значения в

последовательность

**count\_if()** подсчитывает количество выполнений предиката в

последовательности

**search()** находит первое вхождение последовательности как

подпоследовательности

**search\_n()** находит n-е вхождение значения в последовательность

Модифицирующие операции.

**copy()** копирует последовательность, начиная с первого

элемента

**swap()** меняет местами два элемента

**replace()** заменяет элементы с указанным значением

**replace\_if()** заменяет элементы при выполнении предиката

**replace\_copy()** копирует последовательность, заменяя элементы с

указанным значением

**replace\_copy\_if()** копирует последовательность, заменяя элементы

при выполнении предиката

**fill()** заменяет все элементы данным значением

**remove()** удаляет элементы с данным значением

**remove\_if()** удаляет элементы при выполнении предиката

**remove\_copy()** копирует последовательность, удаляя элементы с

указанным значением

**remove\_copy\_if()** копирует последовательность, удаляя элементы при

выполнении предиката

**reverse()** меняет порядок следования элементов на обратный

**random\_shuffle()** перемещает элементы согласно случайному

равномерному распределению

(“тасует” последовательность)

**transform()** выполняет заданную операцию над каждым элементом

последовательности

**unique()** удаляет равные соседнии элементы

**unique\_copy()** копирует последовательность, удаляя равные

соседние элементы

Сортировка.

**sort()** сортирует последовательность с хорошей средней

эф-фективностью

**partial\_sort()** сортирует часть последовательности

**stable\_sort()** сортирует последовательность, сохраняя порядок

следо-вания равных элементов

**lower\_bound()** находит первое вхождение значения в

отсортированной последовательности

**upper\_bound()** находит первый элемент, больший чем заданное значение

**binary\_search()** определяет, есть ли данный элемент в

отсортированной последовательности

**merge()** сливает две отсортированные последовательности

Работа с множествами.

**includes()** проверка на вхождение

**set\_union()** объединение множеств

**set\_intersection()** пересечение множеств

**set\_difference()** разность множеств

Минимумы и максимумы.

**min()** меньшее из двух

**max()** большее из двух

**min\_element()** наименьшее значение в последовательности

**max\_element()** наибольшее значение в последовательности

Перестановки.

**next\_permutation()** следующая перестановка в

лексикографическом порядке

**pred\_permutation()** предыдущая перестановка в

лексикографическом порядке

Пример – Алгоритмы sort, for\_each

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

void Print(int x)

{ cout << x <<' '; }

int main()

{ vector<int> v(4);

v[0] = 3;

v[1] = 1;

v[2] = 5;

v[3] = 2;

sort(v.begin(), v.end() );

for\_each(v.begin(), v.end(), Print);

return 0;

}

Результатом работы программы будет массив

1 2 3 5

Пример – Алгоритмы sort с условием для сортировки

bool comp (int a, int b) {

if (a < b) { //знака будет зависеть направление сортировки

return true;

} else {

return false;

}

}

int main() {

int n;

cin >> n;

vector <int> a(n);

for (int i = 0; i < n; i++) {

int x;

cin >> x;

a[i] = x;

}

sort(a.begin(), a.end(), comp);

for (auto now : a) {

cout << now << endl;

}

return 0;

}

#### Порядок выполнения работы

Написать и отладить три программы. **Первая программа** демонстрирует использование контейнерных классов для хранения встроенных типов данных. **Вторая программа** демонстрирует использование контейнерных классов для хранения пользовательских типов данных. **Третья программа** демонстрирует использование алгоритмов STL.

**В программе № 1** выполнить следующее:

1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания.

2. Просмотреть контейнер.

3. Изменить контейнер, удалив из него одни элементы и изменив другие.

4. Просмотреть контейнер, используя для доступа к его элементам итераторы.

**В программе № 2** то же самое, но для данных пользовательского типа.

**В программе № 3** выполнить следующее:

1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания.

2. Отсортировать его по убыванию элементов.

3. Просмотреть контейнер.

4. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.

5. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер.

6. Просмотреть второй контейнер.

7. Отсортировать первый и второй контейнеры по возрастанию элементов.

8. Просмотреть их.

#### Задания для лабораторной работы по вариантам

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Контейнер | Тип | Пользовательский тип |
| 1 | **vector** | string | Документ |
| 2 | **list** | char | Человек |
| 3 | **deque** | int | Машина |
| 4 | **stack** | double | Студент |
| 5 | **queue** | float | Геометрическая фигура |
| 6 | **set** | string | Мебель |
| 7 | **map** | char | Должность |
| 8 | **list** | int | Факультет |
| 9 | **stack** | double | Автозапчасть |
| 10 | **map** | float | Рецепт |

#### Содержание отчета

1. Ф.И.О., группа, номер, название лабораторной работы.
2. Цель работы.
3. Использованные в работе теоретические сведения.
4. Определение пользовательского класса.
5. Описание контейнера STL.
6. Описание используемых в программах алгоритмов STL.
7. Определения и объяснения, используемых функций сравнения.
8. Результат работы программы.
9. Листинг программы.
10. Выводы.

#### Контрольные вопросы

1. Что содержит STL.
2. В каком случае мы можем использовать vector?
3. В каком случае мы можем использовать stack?
4. В каком случае мы можем использовать map?
5. В каком случае мы можем использовать set?
6. В каком случае мы можем использовать list?
7. В каком случае мы можем использовать queue?