Практическое занятие №5.
Тема: STL. Обработка данных в динамических массивах

Цель: получение практических навыков разработки и анализа объектно-ориентированных программ обработки данных средствами STL.

Введение в библиотеку стандартных шаблонов (STL)

Ядро библиотеки стандартных шаблонов STL (Standard Template Library) образуют три основополагающих элемента: контейнеры, алгоритмы и итераторы. Эти элементы функционируют в тесной взаимосвязи друг с другом, обеспечивая искомые решения проблем программирования.

Контейнеры (containers) — это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов. Например, в классе vector (вектор) определяется динамический массив, в классе queue (очередь) — очередь, в классе list (список) — линейный список. Помимо базовых контейнеров, в библиотеке стандартных шаблонов определены также ассоциативные контейнеры (associative containers), позволяющие с помощью ключей (keys) быстро получать хранящиеся в них значения. Например, в классе map (ассоциативный список) определяется ассоциативный список, обеспечивающий доступ к значениям по уникальным ключам. То есть, в ассоциативных списках хранятся пары величин **ключ**/**значение**, что позволяет при наличии ключа получить соответствующее ключу значение.

В каждом классе-контейнере определяется набор функций для работы с этим контейнером. Например, список содержит функции для вставки, удаления и слияния (merge) элементов. В стеке имеются функции для размещения элемента в стеке и извлечения его из стека.

Алгоритмы (algorithms) выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров.

Итераторы (iterators) — это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

Вдобавок к контейнерам, алгоритмам и итераторам, в библиотеке стандартных шаблонов поддерживается еще 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.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая предикатом (predicate). Предикат может быть бинарным

или унарным. У унарного предиката один аргумент, а у бинарного — два. Возвращаемым значением этих функций является значения истина либо ложь.

Класс vector

Вероятно, самым популярным контейнером является вектор. В классе vector поддерживаются динамические массивы. Динамическим массивом называется массив, размеры которого могут увеличиваться по мере необходимости. Как известно, в C++ в процессе компиляции размеры массива фиксируются. Хотя это наиболее эффективный способ реализации массивов, одновременно он и самый ограниченный, поскольку не позволяет адаптировать размер массива к изменяющимся в процессе выполнения программы условиям. Решает проблему вектор, который выделяет память для массива по мере возникновения потребности в этой памяти. Несмотря на то, что вектор является, по сути, динамическим массивом, для доступа к его элементам подходит обычная индексная нотация, которая используется для доступа к элементам стандартного массива.
Ниже представлена спецификация шаблона для класса vector:

```
template<class T, class Allocator=allocator<T>>class vector
```

Здесь T — это тип предназначенных для хранения в контейнере данных, а ключевое слово Allocator задает распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator( ));

explicit vector (size_type  число, const T &значение = T(),
                 const Allocator &a = Allocator( ));

vector(const vector<T,  Allocator>&объект);

template <class InIter > vector(InIter начало, InIter конец,
                     const Allocator  &a = Allocator ( ));
```

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов — это *число*, а каждый элемент равен значению *значение*. Параметр *значение* может быть значением по умолчанию. В третьей форме конструктора вектор предназначен для одинаковых элементов, каждый из которых — это *объект*. Четвертая форма — это конструктор вектора, содержащего диапазон элементов, заданный итераторами *начало* и *конец*.

Хотя синтаксис шаблона выглядит довольно сложно, в объявлении вектора ничего сложного нет. Ниже представлено несколько примеров такого объявления:

```
vector<int> iv;          // создание вектора нулевой длины для целых

vector<char> cv(5);   // создание пятиэлементного вектора для символов

vector<char> cv(5, 'x');     // создание и инициализация
                             // пятиэлементного вектора для символов
vector<int> iv2(iv);         // создание вектора для целых
                                //из вектора для целых
```

Для класса vector определяются следующие операторы сравнения:
= =, <, <=, ! =, >, >=

Кроме этого для класса vector определяется оператор индекса [ ], что обеспечивает доступ к элементам вектора посредством обычной индексной нотации, которая используется для доступа к элементам стандартного массива.

Функции-члены класса vector

В классе vector несколько десятков функций-членов класса. Наиболее важными функциями-членами являются функции **size(), begin(), end(), push_back(), insert() и erase().** Функция size( ) возвращает текущий размер вектора. Эта функция особенно полезна, поскольку позволяет узнать размер вектора во время выполнения программы. Помните, вектор может расти по мере необходимости, поэтому размер вектора необходимо определять не в процессе компиляции, а в процессе выполнения программы.

Функция begin( ) возвращает итератор начала вектора. Функция end( ) возвращает итератор конца вектора. Как уже говорилось, итераторы очень похожи на указатели и с помощью функций begin( ) и end( ) можно получить итераторы (читай: указатели) начала и конца вектора.

Функция push_back( ) помещает значение в конец вектора. Если это необходимо для размещения нового элемента, вектор удлиняется. В середину вектора элемент можно добавить с помощью функции insert( ). Вектор можно инициализировать. В любом случае, если в векторе хранятся элементы, то с помощью оператора индекса массива к этим элементам можно получить доступ и их изменить. Удалить элементы из вектора можно с помощью функции erase().

А. Добавление элементов в вектор

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

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

```
using  namespace std;

int main()
{
    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
     cout << endl;
    return 0;
}
```

Векторы являются динамическими структурами в отличие от массивов, которые скованы заданными размерами. Поэтому можно динамически добавлять в вектор новые данные.

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

```
vector<int> numbers1 = { 1, 2, 3, 4, 5 };
numbers1.emplace_back(8);  // numbers1 = { 1, 2, 3, 4, 5, 8 };
```

Б. 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:

```
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:

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

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

vector<int> values = { 10, 20, 30, 40, 50 };
vector<int> numbers3 = { 1, 2, 3, 4, 5 };
auto iter3 = numbers3.cbegin(); // константный итератор
/* 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};

vector<int> numbers4 = { 1, 2, 3, 4, 5 };
auto iter4 = numbers4.cend();   // константный итератор
      /* 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};
```

В. Удаление элементов

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

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

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

```
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 **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.

Примеры.

```
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 } */

vector<int> numbers2 = { 1, 2, 3, 4, 5, 6 };
auto begin = numbers2.cbegin(); // указатель на 1-й 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} */
```

Г. Размер вектора

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

```
#include <iostream>
#include <vector>
using  namespace std;

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

    return 0;
}
```

Примеры использования класса vector

В представленном первом примере показаны основные операции, которые можно выполнять при работе с вектором.

Пример 1.

```
# include<iostream>
# include <vector>
using  namespace std;
```

```
int main ( )
{
vector<int> v;       // создание вектора нулевой длины
unsigned int i;
         // вывод на экран размера исходного вектора v
cout « "Размер = " << v.size( ) << endl;
         // помещение значений в конец вектора,
       // по мере необходимости вектор будет расти
for(i=0; i<10;  i++)
        v.push_back(i);
        // вывод на экран текущего размера вектора v
cout << "Новый размер = " << v.size( ) << endl;
         // вывод на экран содержимого вектора v
cout << "Текущее содержимое: \n";
for(i=0;  i<v.size();  i++)
                  cout << v[i] << " ";
                  cout << endl;
          // помещение новых значений в конец вектора,
         //и опять по мере необходимости вектор будет расти
for(i=0;  i<10;  i++)
              v.push_back (i+10) ;
         // вывод на экран текущего размера вектора
cout << "Новый размер = " << v. size () << endl;
           // вывод на экран содержимого вектора
cout << "Текущее содержимое: \n";
for( i=0;  i<v.size(); i++)
                  cout << v[i] << " ";
                     cout << endl;
             // изменение содержимого вектора
for( i=0; i < v.size( ); i++)
                 v[i] = v[i] + v[i];
           // вывод на экран содержимого вектора
cout << "Удвоенное содержимое : \n" ;
for(i=0; i<v.size(); i++)
        cout << v[i] << " ";
           cout << endl;
return 0 ;
}
```

После выполнения программы на экране появится следующее:

```
Размер = 0
Новый размер = 10
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9
Новый размер = 20
```

```
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Удвоенное  содержимое:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

В функции main() создается вектор **v** для хранения целых. Поскольку не используется никакой инициализации, это пустой вектор с равной нулю начальной емкостью, то есть это вектор нулевой длины. Этот факт подтверждается вызовом функции-члена size(). Далее с помощью функции-члена push_back() к концу вектора **v** добавляется десять элементов. Чтобы разместить эти новые элементы, вектор v вынужден увеличиться. Как показывает выводимая на экран информация, его размер стал равным 10. После этого выводится содержимое вектора **v**. Обратите внимание, что для этого используется обычный оператор индекса массива. Далее к вектору добавляется еще десять элементов и, чтобы их разместить, вектор снова автоматически увеличивается. В конце концов, с помощью стандартного оператора индекса массива меняются значения элементов вектора.

Пример 2.

Как вы знаете, в C++ массивы и указатели очень тесно связаны. Доступ к массиву можно получить либо через оператор индекса, либо через указатель. По аналогии с этим в библиотеке стандартных шаблонов имеется тесная связь между векторами и итераторами. Доступ к членам вектора можно получить либо через оператор индекса, либо через итератор. В следующем примере показаны оба этих подхода.

```
#include <iostream>
#include <vector>
using namespace std;

int main ( )
{
    vector<int> v;  // создание вектора нулевой длины
    int i;
             // помещение значений в вектор
    for (i=0; i<10; i++)   v.push_back (i) ;
            // доступ к содержимому вектора
           // с использованием оператора индекса
    for(i=0; i<10; i++)
            cout << v[i] << " ";
                  cout << endl;
          // доступ к вектору через итератор
    vector<int>:: iterator p = v.begin( );
    while(p != v.end)} {
    cout << *p << " ";
```

```
    p++;
    }
return 0;
}
```

После выполнения программы на экране появится следующее:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Пример 3. Хранение в векторе объектов класса

В предыдущих примерах векторы служили для хранения значений только встроенных типов, но этим их возможности не ограничиваются. В вектор можно помещать объекты любого типа, включая объекты классов, создаваемых программистом. Рассмотрим пример, в котором вектор используется для хранения объектов класса **point**. Обратите внимание на то, что в этом классе определяются конструктор по умолчанию и конструктор с параметрами.

```
#include <iostream>
#include <vector>
using namespace std;

class point{
int x,y;
public:
    point(){x=y=0;}
    point(int a, int b){x=a; y=b;}
int get_x( ) {return x;}
int get_y( ) {return y;}
};

int main()
{
    vector<point> v;
    int s = 0;
    unsigned int i;

    // Добавляем в вектор объекты.
       for(i=0; i<5; i++)
          v.push_back(point(i, i-3));

    // Отображаем содержимое вектора.
        for(i=0; i<v.size(); i++)
           cout<<v[i].get_x()<<' '<<v[i].get_y()<<endl;
        cout << endl;

    // Вычисляем сумму x-координат
```

```
    for(i=0; i<v.size(); i++)
        s = s + v[i].get_x();
        cout << "Sum of x: "<< s << endl;

    return 0;
}
```

Как вы поняли, технология заключается в том, что сначала создается вектор нулевой длины для объектов класса **point,** а затем он заполняется объектами. Для удобства выполнения действий с объектами можно перегрузить необходимые для работы операторы.

Практикум

Задание 1. Доработать проект, выполненный для *Практического занятия №3*: создать вектор из 8-ти объектов какого-либо из производных классов. Рассчитать *Вычисляемый показатель*.

Задание 2. Добавить в середину вектора два элемента и рассчитать *Вычисляемый показатель*. Очистить вектор и убедиться, что он пуст.

Задание 3. Разработать меню для демонстрации работы программы и сделанных доработок.

Отчет оформляется по общеустановленным правилам в *электронном виде* со следующим содержанием:

1) титульный лист,
2) тема и цель лабораторной работы,
3) задание на лабораторную работу,
4) текст программы с комментариями,
5) результаты работы программы (вид экрана) и
6) выводы по созданному проекту и использованию средств STL.