Лабораторная работа № 4

Тема работы Шаблоны функций и классов.

Цель работы: изучить принципы и получить практические навыки при использовании шаблонов классов и функций; научиться использовать шаблоны функций для создания группы однотипных функций, а также шаблоны классов для создания группы связанных типов классов; продемонстрировать работу написанной прораммы.

Теоретические сведения: рассмотрены в соответствующих разделах [1, 3−8].

Шаблоны классов и функций.

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. Этот класс можно рассматривать как некоторое описание множества классов, отличающихся только типами их данных. В C++ используется ключевое слово template для обеспечения параметрического полиморфизма. Шаблоны определения класса и шаблоны определения функции позволяют многократно использовать код, корректно по отношению к различным типам.

Формат шаблона имеет вид:

Для классов

```
template <список параметров>
class объявление класса
```

например

```
template <class T>
class Base
{
    T el; // переменная класса с шаблонным типом данных
    …
public:
    Base();
    ~Base();
    void setEl(T _el);  // функция с шаблонным параметром
    …
};
```

Для фунций

```
template <список параметров>
возвращаемое_значение имя_функции (параметры фунции)
```

или

```
template <список параметров>
возвращаемое_значение  имя_класса<список  параметров>::имя_функции
(параметры фунции)
```

например

```
template <typename T>
void print (T el)
{
    std::cout << el;
}
```

Список параметров шаблона класса представляет собой идентификатор типа, подставляемого в объявление данного класса при его генерации. Идентификатору типа предшествует ключевое слово class или typename.

Важно: Определение шаблона может быть только глобальным. Если прототип шаблонного класса/функции располагается в .h-файле, то следует там же и помещать реализацию функции/функции класса. Иначе следует включать в файл как заголовочный файл, так и файл с реализацией, например

```
#inlcude "List.h"
#include "List.cpp"
…
```

Далее будет приведен пример определения шаблона класса вектора (одномерного массива). Какой бы тип ни имели элементы массива (целый, вещественный, с двойной точностью и т. д.), в этом классе должны быть определены одни и те же базовые операции, например, доступ к элементу по индексу и т. д. Если тип элементов вектора задавать как параметр шаблона класса, то система будет формировать вектор нужного типа (и соответствующий класс) при каждом определении конкретного объекта.

В программе шаблон семейства классов с общим именем vector используется для формирования двух классов с массивами целого и символьного типов. В соответствии с требованием синтаксиса имя параметризованного класса, определенное в шаблоне (в примере − vector), используется в программе только

с последующим конкретным фактическим параметром (аргументом), заключенным в угловые скобки. Параметром может быть имя стандартного или определенного пользователем типа. В данном примере использованы стандартные типы int и char. Использовать имя vector без указания фактического параметра шаблона нельзя, т. к. никакое умалчиваемое значение при этом не предусматривается.

В списке параметров шаблона могут присутствовать формальные параметры, не определяющие тип, точнее – это параметры, для которых тип фиксирован.

Рассмотрим пример использования шаблонного класса, реализующего одномерный вектор:

Пример: «Реализовать некоторые методы для работы с вектором, а также перегрузить операцию [].»

```
#include <iostream>
#include <string.h>
#include <locale.h>
#include <iomanip>
#include <typeinfo.h>
using namespace std;

template <class T>
class vector
{      T *ms;                              // указатель на вектор
        int size,ind;                      // размер и индекс
  public:
        vector() : size(0),ind(0),ms(NULL) {}
        vector(int);
        ~vector(){delete [] ms;}
        void set(const T&);         // увеличение размера массива на
один элемент
        T* get_vect();                  // возвращается указатель на
массив
        int get_size();               // возвращается размер массива
        T &operator[](int);          // определение обычного метода
};
template <class T>
vector<T>::vector(int SIZE) : size(SIZE), ind(0)
{
  ms=new T[size];
  const type_info & t=typeid(T);     // получение ссылки t на
  const char* s=t.name();               // объект класса type_info
  for(int i=0;i<size;i++)                   // в зависимости от типа
T
```

```
  if(!strcmp(s,"char")) *(ms+i)=' ';  // заносим пустой символ
  else *(ms+i)=0;                                  // или заносим цифру
нуль
}
template <class T>
void vector<T>::set(const T &t)   // увеличение размера массива на
один элемент
{
   T *tmp = NULL;
   if(++ind>=size)
   {
       tmp=ms;
       ms=new T[size+1];              // ms-указатель на новый массив
   }
   if(tmp) memcpy(ms,tmp,sizeof(T)*size); // перезапись tmp -> ms
   ms[size++]=t;                                   // добавление нового
элемента
   if(tmp) delete [] tmp;                     // удаление временного
массива
}
template <class T>
T* vector<T>::get_vect()
{    return ms;
          }
template <class T>
int vector<T>::get_size()
{      return size;
}
template <class T>
T & vector<T>::operator[](int n)  // определение обычного метода
{
   if(n<0 || (n>=size)) n=0;              // контроль за выходом из
вектора
   return ms[n];
}

int main()
{
    vector <int> VectInt;
    vector <char> VectChar;
    VectInt.set(3);
    VectInt.set(26);
    VectInt.set(12);                          // получен int-вектор из
трех атрибутов
    VectChar.set('a');
    VectChar.set('c');                       // получен char-вектор из
двух атрибутов
```

```
    cout << VectInt[0]<< endl;
    cout << VectChar[0] << endl;
    VectInt[0]=1;
    VectChar[1]='b';
    int *m_i=VectInt.get_vect();
    for(int i=0; i<VectInt.get_size(); i++ )
          cout<<setw(3)<<  *(m_i+i);  cout<<  endl;     //  вывод
целочисленного вектора
    char *m_c=VectChar.get_vect();
    for(int i=0; i<VectChar.get_size(); i++ )
          cout<<setw(3)<<  *(m_c+i);  cout<<  endl;     //  вывод
символьного вектора
    return 0;
}
```

Контрольные вопросы

1. Для чего используются шаблоны классов? Что у них общего с шаблонами функций?
2. Как описываются шаблоны классов?
3. Как создать объект на основе класса, порожденного шаблоном?
4. Каких типов могут быть фактические параметры шаблонов классов?
5. Можно ли описывать в списке параметров шаблона параметры, не определяющие тип?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Задание

Выбрать один из пяти вариантов шаблонных классов, не более 6 человек на один вариант, и указать выбранный варинт в списке.

Реализовать шаблон класса-контейнера в соответствии со своим вариантом. Обязательно реализовать элементы контейнера(Node) через отдельную структуру/класс. Обязательные фукнции:

push() – добавить элемент в контейнер;

pop() – удалить элемент из контейнера;

peek() – получить содержимое элемента контейнера без его удаления;

print() – вывести все элементы контейнера в консоль.
Дополнительное содержимое классов продумать самостоятельно.

Продемонстрировать работу написанного шаблонного класса на нескольких типах данных.

1. List(Список) – реализовать свой шаблон списка. Есть возможность добавления в начало, в конец. Удаления с начала, с конца.
2. Queue(Очередь) – принцип работы: первый пришёл – первым ушёл. Есть возможность добавить в конец, удалить из начала.
3. Linked List(Кольцо) – аналогичени Списку, но последний элемент указывает на первый элемент в списке.
4. Stack(Стек) - принцип работы: первый пришёл – последним ушёл. Есть возможность добавить в конец (в вершину стека), удалить из конца(вершину стека). Доступ к элементам стека осуществляет в обратном порядке относительно очереди, т.е. с конца (вершины).
5. Binary Tree(Бинарное дерево) – представляет из себя структуру данных(рисунок 1), в которой данные от 1-го узла дерева, т.е. корня(родитель), идут к потомкам, т.е. листам. Каждый узел дерева имеет указатель на левого и правого потомка. Перегрузить операторы позволяющие: -- - перейти к левому потомку; ++ - перейти к правому потомку. Дополнительно: *= - перейти к своему родителю.

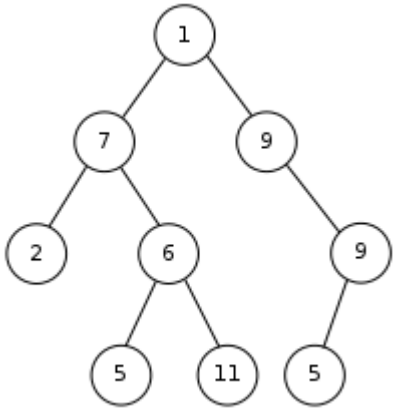

Рисунок 1 – Биарное дерево

Литература

1. Шилдт, Г. Искусство программирования на C++ / пер. с англ. – СПб. : БХВ-Петербург, 2005. – 928 с.

2. Дейтел, Х. Как программировать на C++ / Х. Дейтел, П. Дейтел ; пер. с англ. – М. : Бином-Пресс, 2009. – 1037 с.

3. Страуструп, Б. Программирование. Принципы и практика использования C++ / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2011. – 1246 с.

4. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2012. – 1136 с.

5. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч ; пер. с англ. – М. : Вильямс, 2010 – 720 с.

6. Джамса, К. Учимся программировать на языке C++ / К. Джамса ; пер. с англ. – М. : Мир, 1997. – 320 с.

7. Павловская, Т. C/C++. Программирование на языке высокого уровня / Т. Павловская. – СПб. : Питер, 2013. – 464 с.

8. Луцик Ю.А. Объектно-ориентированное программирование на языке C++ : Учеб. Пособие / Ю.А. Луцик, В.Н. Комличенко. – Минск : БГУИР, 2008. – 266 с.