Лабораторная работа №1.

Шаблоны классов и функций.

Цель работы: изучение средств поддержки шаблонов в языке C++, получение практических навыков создания и использования шаблонов класса и функции.

Теоретические сведения

Одной из возможностей языка C++ является поддержка **шаблонов** (англ. *template*), с помощью которых программист может создавать **обобщенные функции** и **обобщенные классы**. Под обобщенной функцией мы будем понимать функцию C++, не привязанную к определенным типам входных и выходных данных. Иначе говоря, обобщенная функция реализует саму идею алгоритма, и при этом способна работать одинаково с **любыми типами данных**. Простейшим примером может послужить функция, возвращающая наибольшее из двух переданных ей чисел. Так как алгоритм выбора максимального числа не зависит от типа этих чисел (целые, вещественные и т.д.), то удобнее всего он может быть реализован в виде шаблона.

Использование шаблонов класса и функции зачастую позволяет существенно сократить код программы. Кроме того, библиотека STL языка C++ также построена на механизме шаблонов.

1.1. Шаблон класса.

Предположим, что нам необходимо использовать объекты для хранения двух значений – целого числа и символа. Определим для этого класс pair_int_char вида

```
class pair_int_char
{
   private:
     int data1;       // поле для хранения целого числа
     char data2;      // поле для хранения символа
   public:
     pair_int_char(int d1, char d2):        // конструктор
         data1(d1),data2(d2) {};
     void print()                           // вывод на экран
     {
        cout << data1 << "  " << data2 << endl;
     }
}
```

Мы можем использовать этот класс, например, следующим образом

```
pair_int_char a1(5, 'S');     // создаем объект a1
pair_int_char a2(-8, 'i');    // создаем объект a2
a1.print();                   // вызываем метод print() объекта a1
a2.print();                   // вызываем метод print() объекта a2
```

Результатом выполнения этих операторов станут две строки текста, выведенные на экран (определите их содержание самостоятельно).

Допустим, что нам далее потребовались объекты для хранения пары значений типа float и bool. Использовать для этой цели класс pair_int_char мы не можем, так как поля data1 и data2 объявлены с другими типами. Определим новый класс

```
class pair_float_bool
{
   private:
     float data1;     // теперь data1 содержит вещественное число,
     bool data2;      // а в data2 хранится логическая переменная
   public:
     pair_float_bool(float d1, bool d2):
         data1(d1),data2(d2) {}
     void print()
     {
        cout << data1 << "  " << data2 << endl;
     }
}
```

Класс pair_float_bool отличается от pair_int_char лишь типом полей data1, data2 и сигнатурой конструктора (типом аргументов). Использование класса pair_float_bool при этом будет также аналогичным использованию pair_int_char:

```
pair_float_bool b1(1.16, true);
pair_float_bool b2(0.39, false);
b1.print();
b2.print();
```

Очевидно, что любая другая комбинация пары чисел (long-int, float-int, int-float и т.д.) при таком подходе потребует от нас создания нового класса. Объем кода в результате будет быстро расти, хотя назначение и алгоритмическое содержание всех этих классов будет схожим.

Выходом в данной ситуации является использование **шаблона класса** (class template). В рассматриваемом случае он может быть определен следующим образом

```
template <typename T1, typename T2>
class pair
{
   private:
     T1 data1;
     T2 data2;
   public:
     pair(T1 d1, T2 d2):
         data1(d1),data2(d2) {};
     void print()
     {
        cout << data1 << "  " << data2 << endl;
     }
}
```

Здесь определение класса pair записано с применением двух имен типов T1 и T2, которые называются *параметрами типа*. Их имена вводятся в определение класса с помощью конструкции `template <typename T1, typename T2>`.

Теперь с помощью шаблона pair могут создаваться объекты для данных любого типа. Для этого вместо T1 и T2 шаблону необходимо передать *фактические типы*. Например

```
pair<int, char> c1(5, 'S');
pair<float, bool> c2(0.39, false);
c1.print();
c2.print();
```

Запись в первой строке означает *инстанцирование* шаблона класса pair типами int и char, а также создание объекта c1 этого класса с вызовом конструктора. Аналогичным образом создается объект c2 (вторая строка) для пары чисел с типами float и bool. Легко заметить, что объекты c1 и c2 при этом полностью эквивалентны объявленным ранее a1 и b2.

Таким образом, шаблон pair может использоваться огромным количеством способов с любыми комбинациями фактических типов T1 и T2. В качестве T1 и T2 могут применяться также пользовательские типы данных (структуры, классы, перечисления и т.д.). Стандарт C++ допускает также использование ключевого слова class вместо typename в заголовке шаблона.

Ограничения на использование пользовательских типов определяются только самим шаблоном. Например, в конструкторе шаблона pair мы инициализируем поле data1 параметром d1, а поле data2 – параметром d2. Чтобы такой шаблон работал для конкретного типа, необходимо, чтобы для этого типа была определена процедура инициализации. Если T1 и T2 являются классами, то эту роль выполняют конструкторы копирования. Из определения шаблона pair также легко видеть, что для переменных типа T1 и T2 должен быть определен оператор размещения в выходном потоке ostream (`ostream& operator<<(ostream& os, const T1& d1)`, см. пример п. 1.3).

1.2. Шаблон функции.

Как уже было отмечено ранее, шаблоны функций могут использоваться для программной реализации обобщенных алгоритмов. Рассмотрим функцию поиска наибольшего из двух чисел

```
int max_int_int(int a, int b)
{
   if(a > b)
     return a;
   else
     return b;
}
```

Эта функция может использоваться для вычисления максимального из двух значений типа int, но не может быть применена к числам типа float, double, long и т.д. Обобщим ее, сделав шаблоном функции с параметром T

```
template <typename T>
T max(T a, T b)
{
   if(a > b)
     return a;
   else
     return b;
}
```

Тело функции при этом не претерпело изменений, однако в сигнатуре функции (списке аргументов и возвращаемом значении) теперь вместо int указан формальный параметр T. В результате функцию-шаблон max() можно использовать для

различных (в том числе и пользовательских) типов. Существенное отличие шаблона функции от шаблона класса состоит в том, что при вызове такой шаблонной функции тип аргументов явно указываться не должен. Компилятор C++ определит этот тип самостоятельно и вызовет соответствующий вариант функции:

```
int x = 5, y = 1;
float z = 6.2;
cout << max(x, y) << endl;     // здесь используются типы int
cout << max(z, 1.4) << endl;   // здесь используются типы float
```

В шаблоне оба аргумента объявлены с формальным типом T, поэтому компилятор будет требовать, чтобы оба фактических параметра функции были одного и того же типа. Таким образом, попытка вызова функции max, например, с параметрами x и z приведет к ошибке. Кроме того, в теле шаблона используется оператор > для сравнения значений a и b. Если в функцию max передать аргументы, для которых этот оператор не определен, то это вновь приведет к возникновению ошибки. К примеру, исходный код

```
pair <float, int> x(1.5,222), y(0.1,999), z(0.0,0);
z = max(x,y);
```

не будет скомпилирован из-за отсутствия определения оператора > для двух объектов pair. Ошибки можно избежать, если предварительно определить этот оператор, например, следующим образом

```
bool operator> (pair<float, int> x, pair<float, int> y)
{
   return x.data1 > y.data1;   // сравниваем первые значения
}
```

1.3. Пример: массив с контролем выхода за пределы.

Рассмотрим еще один пример, демонстрирующий технику использования шаблонов C++. Пример показывает программную реализацию типонезависимого массива фиксированной длины со средствами контроля выхода за его пределы. Доступ к элементам осуществляется по целочисленному индексу k с помощью метода operator [] (int k). Обратите внимание на использование целочисленной константы n в качестве одного из аргументов шаблона. Этот прием позволяет использовать шаблон array более гибко, создавая массивы произвольной длины.

```
#include <iostream>
#include <exception>

template <class T, int n>
class array
{
 public:
   array();
   ~array();
   T& operator[](int k);
 private:
   T *data;
};

template <class T, int n>
array<T,n>::array()
{
   data = new T[n];
}

template <class T, int n>
array<T,n>::~array()
{
   delete []data;
}

template <class T, int n>
T& array<T,n>::operator[](int k)
{
   if( k<0 || k>=n )
     throw std::out_of_range("Выход за пределы массива!");
   return data[k];
}

template <class T, int n>
std::ostream& operator<<(std::ostream& os, array<T,n>& obj)
{
   os << "\nСодержимое массива:";
   for(int i=0; i<n; i++)
     os << "\n data[" << i << "]=" << obj[i];
   return os;
}
```

```
int main()
{
   setlocale(LC_ALL, "RUS");
   array<float, 20> x;

   for(int i=0; i<20; i++)
     x[i] = 10.1*i+1.15;

   std::cout << x;
   std::cin.get();

   return 0;
}
```

Пример демонстрирует возможность использования константных выражений в качестве аргументов шаблона. Шаблон array объявлен с двумя аргументами: `<class T>` обозначает тип элементов массива (это может быть любой из стандартных типов или пользовательский класс), `<int n>` задает целочисленную константу, которая определяет количество элементов. Инстанцируется данный шаблон также двумя аргументами (см. пример в функции main).

Внутри шаблона array мы используется динамический способ размещения данных в памяти (оператор new). Так как размер массива в нашем случае известен заранее (передается в качестве аргумента шаблона), то выделение памяти удобнее всего произвести на этапе создания объекта (в конструкторе), а освобождение памяти – на этапе его уничтожения (в деструкторе).

Кроме конструктора, деструктора и перегруженного оператора индексирования для доступа к элементам массива, нами также реализована функция размещения объекта класса array в выходном потоке. Эта функция позволяет вывести на экран (или записать в файл) весь массив целиком с помощью единственного оператора <<. Пример использования данной функции можно найти в основной программе.

1.4. Варианты заданий:

1) Создайте шаблон класса TVector для одномерного массива переменной длины. В качестве параметра шаблона используйте тип элементов T. Реализуйте оператор индексирования operator [] для обращения к элементам массива. Продемонстрируйте работу шаблона для различных типов данных.

2) Создайте шаблон класса TMatrix для двумерного массива (матрицы) фиксированного размера n x m. В качестве параметров шаблона используйте тип хранимых данных и размеры матрицы n, m. Реализуйте функцию подсчета среднего арифметического всех элементов, и функцию вывода всей матрицы на экран. Продемонстрируйте работу шаблона для различных типов данных.

3) Создайте шаблоны функций max(), min() и med(), выполняющих поиск максимального элемента, минимального элемента и медианы в одномерном массиве, соответственно. В качестве параметров шаблона используйте тип T и количество n элементов массива. Продемонстрируйте работу шаблонов для различных типов элементов и длины массива.

4) Создайте шаблоны функций sum(), prod() и sqr(), выполняющих расчет суммы всех элементов, произведения всех элементов и суммы квадратов всех элементов одномерного массива, соответственно. В качестве параметров шаблона используйте тип T и количество n элементов массива. Продемонстрируйте работу шаблонов для различных типов элементов и длины массив.

5) Создайте шаблон функции is_monotonous(), проверяющей элементы одномерного массива на монотонность возрастания или убывания. В качестве параметров шаблона используйте тип T и количество n элементов массива. Продемонстрируйте работу шаблона для различных типов элементов и длины массива.

6) Создайте шаблон функции transpose(), выполняющей транспонирование квадратной матрицы. В качестве параметров шаблона используйте тип элементов T и размер матрицы n. Продемонстрируйте работу шаблона для различных типов элементов и размеров матриц.

7) Создайте шаблон функции inverse(), выполняющей обращение квадратной матрицы размером 3 x 3. В качестве параметра шаблона используйте тип элементов T. Продемонстрируйте работу шаблона для матриц с различными типами элементов.

8) Создайте шаблон функции matrix_product(), выполняющей перемножение двух квадратных матриц размером n x n. В качестве параметров шаблона используйте тип

элементов T и размер матриц n. Продемонстрируйте работу шаблона для матриц с различными типами элементов и размерами.

9) Создайте шаблон функции matrix_sum(), выполняющей сложение двух прямоугольных матриц размером n x m. В качестве параметров шаблона используйте тип элементов T и размеры матриц n, m. Продемонстрируйте работу шаблона для матриц с различными типами элементов и размерами.

10) Создайте шаблон функции scalar_product(), рассчитывающей скалярное произведение двух векторов в пространстве размерности n. В качестве параметров шаблона используйте тип элементов вектора T и его размер n. Продемонстрируйте работу шаблона для векторов с различными типами элементов и размерами.

11) Создайте шаблон функции vector_product(), рассчитывающей векторное произведение двух векторов в трехмерном пространстве. В качестве параметра шаблона используйте тип элементов вектора T. Продемонстрируйте работу шаблона для векторов с различными типами элементов.