№ 7 Шаблоны классов и функций

Модифицировать проект, созданный в предыдущем практикуме №6, следующим образом. Создать шаблон заданного класса. Проверить использование шаблона для стандартных типов данных (в качестве стандартных типов использовать целые и вещественные типы). Определить пользовательский класс, который будет использоваться в качестве параметра шаблона. Для пользовательского типа взять класс из проекта лабараторной №2 или 3.

Написать глобальную функцию-шаблон в соответствии с вариантом. Проверить работу функции с int, double , пользовательским классом (для манипуляций выбрать одно из полей класса).

Вариант 1	Написать функцию-шаблон, вычисляющую значение в массиве меньше заданного.
Вариант 2	Написать функцию-шаблон - сортировки методом подсчета.
Вариант 3	Написать функцию-шаблон, суммирующую элементы в множестве.
Вариант 4	Написать функцию-шаблон уменьшающую каждый элемент множества на заданное значение
Вариант 5	Написать параметризованную функцию сортировки методом быстрой сортировки.
Вариант 6	Создать шаблон функции прореживания: три параметра: откуда выбирать, куда выбирать и через сколько выбирать.
Вариант 7	Написать функцию-шаблон, вычисляющую среднее значение в множестве.
Вариант 8	Написать функцию-шаблон, вычисляющую минимальное значение в множестве.
Вариант 9	Написать функцию-шаблон, вычисляющую минимальное значение в списке.
Вариант 10	Создать шаблон функции, которая возвращает максимально близкое к заданному элементу из значений списка. Аргументами функции должны быть значение, адрес списка.
Вариант 11	Написать функцию-шаблон, удваивающую элементы в списке.
Вариант 12	Написать функцию-шаблон, удаляющую элементы равные заданному в списке.
Вариант 13	Написать функцию-шаблон, которая подсчитывет количество элементов больших заданному в стеке
Вариант 14	Создать шаблон функцию прореживания: два параметра:

	откуда выбирать и через сколько выбирать.
Вариант 15	Создать шаблон функцию min, которая возвращает максимальное отрицательное значение вектора
Вариант 16	Написать функцию-шаблон, которая подсчитывет количество элементов больших заданному в матрице
Вариант 17	Написать функцию-шаблон, которая возвращает сумму всех элментов матрицы
Вариант 18	Написать функцию-шаблон, которая обнуляет элменты в матрице равные заданному.
Вариант 19	Написать функцию-шаблон, которая возвращает максимально близкое заданному
Вариант 20	Написать функцию-шаблон, которая возвращает количество элементов в стеке меньших заданного
Вариант 21	Написать функцию-шаблон, которая удаляет из стека элементы больше заданного
Вариант 22	Написать функцию-шаблон, которая возвращает количество четных элементов в очереди
Вариант 23	Написать функцию-шаблон, которая шифрует строку в соответсвии с паролем
Вариант 24	Написать функцию-шаблон, которая прорежевает строку, удаляя каждый n-ый
Вариант 25	Написать функцию-шаблон, которая возвращает количество элементов в строке равных заданному
Вариант 26	Написать функцию-шаблон, которая шифрует пароль (алгорим придумать самим)

Примеры

Массив

```
...
...

//shablon classa massiv
template <class theType>
class Array
{
	theType *a;//pointer to the array
	int count;//number of elements
	int numberToInput;//number of elements to input using
operator >>
public:
//empty constructor
	Array()
	{
		a=NULL;
		count=0;
```

```
            numberToInput=0;
        }

    //constructor too
        Array(int count_)
        {
            a=new theType [count_];
            count=count_;
            for (int i=0 ; i<count ; i++)
                a[i]=0;
            numberToInput=0;
        }

    //consructor, which copys the array to the object
        Array(theType *a_, int count_)
        {
            a=new theType [count_];
            for (int i=0 ; i<count_ ; i++)
                a[i]=a_[i];
            count=count_;
            numberToInput=0;
        }

    //copy constructor
        Array(Array& a_)
        {
            a=new theType [a_.count];
            count=a_.count;
            for (int i=0 ; i<count ; i++)
                a[i]=a_.a[i];
            numberToInput=a_.numberToInput;
        }

    //destructor
        ~Array()
        {
            delete [] a;
            numberToInput=0;
            count=0;
        }

        void setNumberToInput(int n)
        {
            numberToInput=n;
        }

    //operator =
        Array& operator = (Array& a_)
        {
            if (!(a_==(*this)))
            {
                count=a_.count;
                numberToInput=a_.numberToInput;
```

```
                for (int i=0 ; i<count ; i++)
                     a[i]=a_.a[i];
           }
           return *this;
     }

//operator <<
     friend ostream& operator <<(ostream& out, Array& a_)
     {
           for (int i=0 ; i<a_.count ; i++)
                out<<a_.a[i]<<" ";
           out<<endl;
           return out;
     }

//operator >>
     friend istream& operator >>(istream& in, Array& a_)
     {
           cout<<"Input started:\n";
           if(a_.a) delete a_.a;
           a_.a=new theType [a_.numberToInput];
           for (int i=0 ; i<a_.numberToInput ; i++)
           {
             in>>a_.a[i];
                a_.count++;
           }
           a_.numberToInput=0;
           cout<<"\nInput ended.\n\n";
           return in;
     }

     theType& operator [](int i_)
     {
           if (i_<count && i_>=0)
                return a[i_];
     }

//operator ()
     int operator ()()
     {
           return count;
     }
};

#include "Array.h"


int main()
{
//Part 1:
     {
           int *ar;
           int n;
```

```
          cout<<"\n\n\t\t<<< Part 1. >>>\n\n";
          cout<<"Template parameter: int.\n\n";
          cout<<"Let's input array of integers. Enter number of
elements: ";
          cin>>n;
          ar=new int [n];//create massive
          cout<<"Start input.\n";
          for (int i=0 ; i<n ; i++)
               cin>>ar[i];
          cout<<"\nEnd input.\n";
          Array<int> a(ar,n);//+constructor
          cout<<"Print a:\n"<<a<<endl;
          cout<<"\nArray b will be created by copying Array
a.\n";
          Array<int> b(a);
          cout<<"Print b:"<<b<<endl;
          cout<<"We'll create Array c with 10 elements. All
elements = 0.\n";
          Array<int> c(10);
          cout<<"Print c:\n"<<c<<endl;
          cout<<"Let's change some elements in c. And then we'll
print c:\n";
          c[0]=15;
          c[5]=17;
          c[9]=-12;
          cout<<c<<endl;
          cout<<"\nCheck operator int():\n\ta():
"<<a()<<"\n\tb(): "<<b()<<endl;
          cout<<"\nChecking operator []:\n\ta[0]: "<<a[0]<<endl;
          cout<<"\n\n\t\t<<< End of part 1. >>>\n\n\n";
     }
//Part 2:
     {
          double *ar;
          int n;
          cout<<"\n\n\t\t<<< Part 2. >>>\n\n";
          cout<<"Template parameter: double.\n\n";
          cout<<"Let's input array of double. Enter number of
elements: ";
          cin>>n;
          ar=new double [n];
          cout<<"Start input.\n";
          for (int i=0 ; i<n ; i++)
               cin>>ar[i];
          cout<<"\nEnd input.\n";
          Array<double> a(ar,n);
          cout<<"Print a:\n"<<a<<endl;
          cout<<"\nArray b will be created by copying Array
a.\n";
          Array<double> b(a);
          cout<<"Print b:"<<b<<endl;
          cout<<"We'll create Array c with 10 elements. All
elements = 0.\n";
```

```
          Array<double> c(10);
          cout<<"Print c:\n"<<c<<endl;
          cout<<"Let's change some elements in c. And then we'll
print c:\n";
          c[0]=15.45;
          c[5]=17.1;
          c[9]=-12.004;
          cout<<c<<endl;
          cout<<"\nCheck operator int():\n\ta():
"<<a()<<"\n\tb(): "<<b()<<endl;
          cout<<"\nChecking operator []:\n\ta[0]: "<<a[0]<<endl;
          cout<<"\n\n\t\t<<< End of part 2. >>>\n\n\n";
     }
     return 0;
}
```

Комплексные числа

```
#include <iostream>
#include <conio.h>

template <class YYY> class Complex
 {   YYY real, imaginary;
  public:
     Complex() { cin >> *this; }
     friend ostream& operator << (ostream&, const Complex&);
     friend istream& operator >> (istream&, Complex&);
     Complex operator + (Complex);
     Complex operator - (Complex);
     Complex operator * (Complex);
     Complex operator / (Complex);
     void operator == (Complex);
     void operator != (Complex);
};

template <class YYY>
ostream& operator << (ostream& output, const Complex< YYY >& C)
 {
  if(C.imaginary>=0)
     output << C.real << "+" << C.imaginary << "*i"<< endl;
  else
     output << C.real << C.imaginary << "*i"<< endl;
  return output;
 }

template <class YYY >
istream& operator >> (istream& input, Complex< YYY >& C)
 {
     cout << "vvedite veschestvennuy chast -> "; input >> C.real;
     cout << "vvedite mnimuy chast -> "; input >> C.imaginary;
  return input;
 }
```

```
template <class YYY >
Complex< YYY > Complex< YYY >::operator + (Complex< YYY > C)
 {
     C.real=real+C.real; C.imaginary=imaginary+C.imaginary;
  return C;
 }

template <class YYY >
Complex< YYY > Complex< YYY >::operator - (Complex< YYY > C)
 {
     C.real=real-C.real; C.imaginary=imaginary-C.imaginary;
  return C;
 }

template <class YYY >
Complex< YYY > Complex< YYY >::operator * (Complex< YYY > C)
 {
  double REAL=C.real, IMAGINARY=C.imaginary;
     C.real=real*REAL - imaginary*IMAGINARY;
     C.imaginary=imaginary*REAL + real*IMAGINARY;
  return C;
 }

template <class YYY >
Complex< YYY > Complex< YYY >::operator / (Complex< YYY > C)
 {
  double REAL=C.real, IMAGINARY=C.imaginary;
     C.real=(real*REAL + imaginary*IMAGINARY)/(REAL*REAL +
IMAGINARY*IMAGINARY);
     C.imaginary=(imaginary*REAL - real*IMAGINARY)/(REAL*REAL +
IMAGINARY*IMAGINARY);
  return C;
 }

template <class YYY >
void Complex< YYY >::operator == (Complex< YYY > C)
 {
     if(C.real==real&&C.imaginary==imaginary) cout << "chisla
ravni"<<endl;
     else cout << "chisla neravni";
 }

template <class YYY >
void Complex<YYY>::operator != (Complex< YYY > C)
 {
     if(C.real!=real||C.imaginary!=imaginary) cout << "chisla
neravni"<<endl;
     else cout << "chisla ravni"<<endl;
 }



template <class YYY >
```

```
void vybor(Complex< YYY >& c1, Complex< YYY >& c2)
{ int O;
     cout << " 1) slozit(+)\n 2) vichest(-)\n";
     cout << " 3) umnozit(*)\n 4) delit(/)\n";
     cout << " 5) sravnenie na ravenstvo(==)\n";
     cout << " 6) sravnenie na neravenstvo(!=)\n";
     cin >> O;
       switch(O)
       { case 1: cout << "otvet: " << c1+c2; break;
         case 2: cout << "otvet: " << c1-c2; break;
         case 3: cout << "otvet: " << c1*c2; break;
         case 4: cout << "otvet: " << c1/c2; break;
         case 5: c1==c2; break;
         case 6: c1!=c2; break;
       }

}

void main()
 {
     //clrscr();
     cout << "\t\t\t1-e kompleksnoe chislo tipa float:" << endl;
     Complex<float> Complex_Float_1;
     cout << "\t\t\t2-e kompleksnoe chislo tipa float:" << endl;
     Complex<float> Complex_Float_2;
     vybor(Complex_Float_1, Complex_Float_1);

//   clrscr();
     cout << "\t\t\t1-e kompleksnoe chislo tipa int:" << endl;
     Complex<int> Complex_Int_1;
     cout << "\t\t\t2-e kompleksnoe chislo tipa int:" << endl;
     Complex<int> Complex_Int_2;
     vybor(Complex_Int_1, Complex_Int_2);
 }
```

Теория

Аналогично шаблонам функций. определяется шаблон семейства классов:

template<список_параметров_шаблона> определение_класса

Шаблон семейства классов определяет способ построения отдельных классов подобно тому, как класс определяет правила построения и формат отдельных объектов. В определении класса, входящего в шаблон, особую роль играет имя класса. Оно является не именем отдельного класса, а параметризованным именем семейства классов.

Как и для шаблонов функций, определение шаблона класса может быть только глобальным.

Следуя авторам языка и компилятора C++, рассмотрим векторный класс (в число данных входит одномерный массив). 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.

Следующий шаблон автоматически формирует классы векторов с указанными свойствами:

```
// vector.h - шаблон векторов
template<class T> // T - параметр шаблона
class Vector
{
public:
  Vector(int); // Конструктор класса vector

  ~Vector() // Деструктор
  {
    delete [] data;
  }

  // Расширение действия (перегрузка) операции "[]":
  T &operator [](int i)
  {
    return data [i];
  }

protected:
  T *data;  // Начало одномерного массива
  int size; // Количество элементов в массиве
};

// vector.cpp
// Внешнее определение конструктора класса:
template<class T> Vector <T>::Vector(int n)
{
  data = new T[n];
  size = n;
};
```

Когда шаблон введен, у программиста появляется возможность определять конкретные объекты конкретных классов, каждый из которых параметрически

порожден из шаблона. Формат определения объекта одного из классов, порождаемых шаблоном классов:

имя_параметризованного_класса <фактические_параметры_шаблона>

имя_объекта (параметры_конструктора);

В нашем случае определить вектор, имеющий восемь вещественных координат типа double, можно следующим образом:

```
Vector<double> z(8);
```

Проиллюстрируем сказанное следующей программой:

```
// формирование классов с помощью шаблона
#include <iostream>

#include "vector.h" // Шаблон класса "вектор"

int main(void)
{
  Vector<int> X(5);       //Создаем объект класса "целочисленный
вектор"
  Vector<char> C(5);      // Создаем объект класса "символьный вектор"
  for (int i = 0; i < 5; i++) // Определяем компоненты векторов
  {
    X[i] = i;
    C[i] = 'A' + i;
  }
  for (i = 0; i < 5; i++)
  {
    cout << X[i] << C[i]; // 0 A 1 B 2 C 3 D 4 E
  }
  return 0;
}
```

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 - 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:

```
#include <iostream>

template<class T, int size = 64> class Row
{
public:
  Row()
  {
    length = size;
    data = new T [size];
  }

  ~Row()
  {
    delete [] data;
  }

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

protected:
  T *data;
  int length;
};

int main(void)
{
  Row<float, 8> rf;
  Row<int, 8> ri;
  for (int i = 0; i < 8; i++)
  {
    rf[i] = i;
    ri[i] = i * i;
  }
  for (i = 0; i < 8, i++)
  {
    cout << rf[i] << ri[i]; //0 0 1 1 2 4 3 9 4 16 5 25 6 36 7 49
  }
  return 0;
}
```

В качестве аргумента, заменяющего при обращении к шаблону параметр size, взята константа. В общем случае может быть использовано константное выражение, однако выражения, содержащие переменные, использовать в качестве фактических параметров шаблонов нельзя.

Основные свойства шаблонов классов.

1. Компонентные функции параметризованного класса автоматически являются параметризованными. Их не обязательно объявлять как параметризованные с помощью template.
2. Дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными функциями, т.е. по умолчанию такие функции являются дружественными для всех классов, которые организуются по данному шаблону.
3. Если friend-функция содержит в своем описании параметр типа параметризованного класса, то для каждого созданного по данному шаблону класса имеется собственная friend-функция.
4. В рамках параметризованного класса нельзя определить friend-шаблоны (дружественные параметризованные классы).
5. С одной стороны, шаблоны могут быть производными (наследоваться) как от шаблонов, так и от обычных классов, с другой стороны, они могут использоваться в качестве базовых для других шаблонов или классов.
6. Шаблоны функций, которые являются членами классов, нельзя описывать как virtual.
7. Локальные классы не могут содержать шаблоны в качестве своих элементов.
8. Реализация компонентной функции шаблона класса, которая находится вне определения шаблона класса, должна включать дополнительно следующие два элемента:
9. Определение должно начинаться с ключевого слова template, за которым следует такой же список_параметров_типов в угловых скобках, какой указан в определении шаблона класса.
10. За именем_класса, предшествующим операции области видимости (::), должен следовать список_имен_параметров шаблона.

```
template<список_типов>

тип_возвр_значения
имя_класса<список_имен_параметров>::имя_функции(список_параметров)
{ … }
```

Smart-указатель.

Рассмотрим еще один пример использования класса-шаблона. С его помощью мы попытаемся "усовершенствовать" указатели языка Си++. Если указатель указывает на объект, выделенный с помощью операции new, необходимо явно вызывать операцию delete тогда, когда объект становится не нужен.

Однако далеко не всегда просто определить, нужен объект или нет, особенно если на него могут ссылаться несколько разных указателей. Разработаем класс, который ведет себя очень похоже на указатель, но автоматически уничтожает объект, когда уничтожается последняя ссылка на него. Назовем этот класс smart-указатель (Smart Pointer). Идея заключается в том, что настоящий указатель мы окружим специальной оболочкой. Вместе со значением указателя мы будем хранить счетчик - сколько других объектов на него ссылается. Как только значение этого счетчика станет равным нулю, объект, на который указатель указывает, пора уничтожать.

Структура Ref хранит исходный указатель и счетчик ссылок.

```
template<class T> struct Ref
{
  T *realPtr;
  int counter;
};
```

Теперь определим интерфейс smart-указателя:

```
template<class T> class SmartPtr
{
public:
  // конструктор из обычного указателя
 SmartPtr(T *ptr = 0);
  // копирующий конструктор
  SmartPtr(const SmartPtr &s);
  ~SmartPtr();
  SmartPtr &operator =(const SmartPtr &s);
  SmartPtr &operator =(T *ptr);
  T *operator ->() const;
  T &operator *() const;
private:
  Ref<T> *refPtr;
};
```

У класса SmartPtr определены операции обращения к элементу ->, взятия по адресу * и операции присваивания. С объектом класса SmartPtr можно обращаться практически так же, как с обычным указателем.

```
struct A
{
  int x;
  int y;
};

SmartPtr<A> aPtr(new A); // создать новый указатель
```

```
int x1 = aPtr->x;         // обратиться к элементу A
(*aPtr).y = 3;            // обратиться по адресу
```

Рассмотрим реализацию методов класса SmartPtr. Конструктор инициализирует объект указателем. Если указатель равен нулю, то refPtr устанавливается в ноль. Если же конструктору передается ненулевой указатель, то создается структура Ref, счетчик обращений в которой устанавливается в 1, а указатель - в переданный указатель:

```
template<class T> SmartPtr<T>::SmartPtr(T *ptr)
{
  if (!ptr)
  {
    refPtr = 0;
  }
  else
  {
    refPtr = new Ref<T>;
    refPtr->realPtr = ptr;
    refPtr->counter = 1;
  }
}
```

Деструктор уменьшает количество ссылок на 1 и, если оно достигло 0, уничтожает объект

```
template<class T> SmartPtr<T>::~SmartPtr ()
{
  if (refPtr)
  {
    refPtr->counter--;
    if (refPtr->counter <= 0)
    {
      delete refPtr->realPtr;
      delete refPtr;
    }
  }
}
```

Реализация операций -> и * довольно проста:

```
template<class T> T *SmartPtr<T>::operator ->() const
{
  if (refPtr) return refPtr->realPtr;
  else return 0;
}

template<class T> T &SmartPtr<T>::operator *() const
{
```

```
  if (refPtr) return *refPtr->realPtr;
  else throw bad_pointer;
}
```

Самые сложные для реализации - копирующий конструктор и операции присваивания. При создании объекта SmartPtr - копии имеющегося - мы не будем копировать сам исходный объект. Новый smart-указатель будет ссылаться на тот же объект, мы лишь увеличим счетчик ссылок.

```
template<class T> SmartPtr<T>::SmartPtr(const SmartPtr &s)
: refPtr(s.refPtr)
{
  if (refPtr) refPtr->counter++;
}
```

При выполнении присваивания, прежде всего, нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому, подобно тому, как это сделано в копирующем конструкторе.

```
template<class T> SmartPtr &SmartPtr<T>::operator =(const SmartPtr &s)
{
  // отсоединиться от имеющегося указателя
  if (refPtr)
  {
    refPtr->counter--;
    if (refPtr->counter <= 0)
    {
      delete refPtr->realPtr;
      delete refPtr;
    }
  }
  // присоединиться к новому указателю
  refPtr = s.refPtr;
  if (refPtr) refPtr->counter++;
}
```

В следующей функции при ее завершении объект класса Complex будет уничтожен:

```
void foo(void)
{
  SmartPtr<Complex> complex(new Complex);
  SmartPtr<Complex> ptr = complex;
}
```

Задание свойств класса.

Одним из методов использования шаблонов является уточнение поведения с помощью дополнительных параметров шаблона. Предположим, мы пишем функцию сортировки вектора:

```
template<class T> void sort_vector(vector<T> &vec)
{
  for (int i = 0; i < vec.size() -1; i++)
    for (int j = i; j < vec.size(); j++)
    {
      if (vec[i] < vec[j])
      {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Эта функция будет хорошо работать с числами, но если мы захотим использовать ее для массива указателей на строки (char *), то результат будет несколько неожиданный. Сортировка будет выполняться не по значению строк, а по их адресам (операция "меньше" для двух указателей - это сравнение значений этих указателей, т.е. адресов величин, на которые они указывают, а не самих величин). Чтобы исправить данный недостаток, добавим к шаблону второй параметр:

```
template<class T, class Compare> void sort_vector(vector<T> &vec)
{
  for (int i = 0; i < vec.size() -1; i++)
  {
    for (int j = i; j < vec.size(); j++)
    {
      if (Compare::less (vec[i], vec[j]))
      {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
  }
}
```

Класс Compare должен реализовывать статическую функцию less, сравнивающую два значения типа T Для целых чисел этот класс может выглядеть следующим образом:

```
class CompareInt
{
  static bool less(int a, int b)
  {
    return a < b;
  }
};
```
Сортировка вектора будет выглядеть так:

```
vector<int> vec;
sort<int, CompareInt>(vec);
```

Для указателей на байт (строк) можно создать класс

```
class CompareCharStr
{
  static bool less(char *a, char *b)
  {
    return strcmp(a, b) >= 0;
  }
};
```

и, соответственно, сортировать с помощью вызова

```
vector<char *> svec;
sort<char *, CompareCharStr>(svec);
```

Как легко заметить, для всех типов, для которых операция "меньше" имеет нужный нам смысл, можно написать шаблон класса сравнения:

```
template<class T> Compare
{
  static bool less(T a, T b)
  {
    return a < b;
  }
};
```
и использовать его в сортировке (обратите внимание на пробел между закрывающимися угловыми скобками в параметрах шаблона; если его не поставить, компилятор спутает две скобки с операцией сдвига):

```
vector<double> dvec;
sort<double, Compare<double> >(dvec);
```

Чтобы не загромождать запись, воспользуемся возможностью задать значение параметра по умолчанию. Так же, как и для аргументов функций и

методов, для параметров шаблона можно определить значения по умолчанию. Окончательный вид функции сортировки будет следующий:

```
template<class T, class C = Compare<T> >
void sort_vector(vector<T> &vec)
{
  for (int i = 0; i < vec.size() -1; i++)
  {
    for (int j = i; j < vec.size(); j++)
    {
      if (C::less(vec[i], vec[j]))
      {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
  }
}
```

Второй параметр шаблона иногда называют параметром-штрих, поскольку он лишь модифицирует поведение класса, который манипулирует типом, определяемым первым параметром.

Шаблоны функций.

Шаблоны, которые называют иногда родовыми или параметризованными типами, позволяют создавать (конструировать) семейства родственных функций и классов.

Цель введения шаблонов функций - автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждого набора формальных параметров определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризуется. Параметризовать в шаблоне функций можно тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

В определении шаблона семейства функций используется служебное слово template. Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки <>. Каждый формальный параметр шаблона обозначается служебным словом class, за которым следует имя параметра (идентификатор). Пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template<class type> type abs(type x)
```

```
{
  return x > 0 ? x: -x;
      }
```

Описание шаблона семейства функций состоит из двух частей:

template<class тип_данных>
тип_возвр_значения имя_функции(список_параметров)
{тело_функции}

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template<class T> void swap(T *x, T *y)
{
  T z = *x;
  *x = *y;
  *y = x;
}
```

Здесь параметр T шаблона функций используется не только в заголовке для спецификации формальных параметров, но и в теле определения функции, где он задает тип вспомогательной переменной z.

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в теле программы. Например, если программист употребляет обращение abs(-10.3), то на основе приведенного ранее шаблона компилятор сформирует такое определение функции:

```
double abs(double x)
{  return x > 0 ? x: -x; }
```

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется числовое значение 10.3.

Если в программе присутствует приведенный ранее шаблон семейства функций swap()

```
long k = 4, d = 8;
swap(&k, &d);
```

то компилятор сформирует определение функции:

```
void swap(long *x, long *y)
{
  long x = *x;
  *x = *y;
  *y = x;
```

```
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных k, d поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44, b = 66.3;
swap(&a, &b);
```

то сформируется и выполнится функция

```
void swap(double *x, double *y)
{
  double x = *x;
  *x = *y;
  *y = x;
    }
```

Проиллюстрируем сказанное о шаблонах на более конкретном примере. Рассмотрим программу, используем некоторые возможности функций, возвращающих значение типа "ссылка". Но тип ссылки будет определяться параметром шаблона:

```
#include <iostream>

//Функция определяет ссылку на элемент с максимальным значением
template<class type> type &rmax(int n, type d[])
{
  int im = 0;
  for (int i = 1; i < n; i++)
    im = d[im] > d[i] ? im: i;
  return d[im];
}

int main(void)
{
  int n = 4;
  int x[] = { 10, 20, 30, 14 }; //Массив целых чисел
  cout << "\nrmax(n,x) = " << rmax(n, x); // rmax(n,x) = 30
  rmax(n, x) = 0;
  for (int i = 0; i < n; i++)
  cout << "\tx[" << i << "] =" << x[i]; // x[0] = 10 x[1] ...
  float arx[] = { 10.3, 20.4, 10.5 }; //Массив вещественных чисел
  cout << "\nrmax(3,arx) = " << rmax(3, arx); //rmax(3,arx) = 20.4
  rmax(3, arx) = 0;
  for (int i = 0; i < 3; i++)
  cout << "\tarx[" << i << "] =" << arx[i]; //arx[0] = 10.3 ...
  return 0;
}
```

В программе используются два разных обращения к функции rmax(). В одном случае параметр - целочисленный массив и возвращаемое значение - ссылка типа int. Во втором случае фактический параметр - имя массива типа float и возвращаемое значение имеет тип ссылки на float.

По существу механизм шаблонов функций позволяет автоматизировать подготовку переопределений перегруженных функций. При использовании шаблонов уже нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор автоматически,анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращениях. Дальнейшая обработка выполняется так же, как и для перегруженных функций.

Параметры шаблонов.

Можно считать, что параметры шаблона являются его формальными параметрами, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Именно по ним выполняется параметрическая настройка и с учетом этих типов генерируется конкретный текст определения функции. Однако, говоря о шаблоне семейства функций, обычно употребляют термин "список параметров шаблона", не добавляя определения "формальных".

Перечислим основные свойства параметров шаблона:

1. Имена параметров шаблона должны быть уникальными во всем определении шаблона.
2. Список параметров шаблона не может быть пустым.
3. В списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово class.

```
template<class type1, class type2>
```

Соответственно, неверен заголовок:

```
template<class type1, type2, type3>
```

4. Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, то есть ошибочен такой заголовок:

```
template<class t, class t, class t>
```

5. Имя параметра шаблона (в примерах - type1, type2) имеет в определяемой шаблоном функции все права имени типа, то есть с его помощью могут специализироваться формальные параметры, определяться тип возвращаемого функцией значения и типы любых объектов, локализованных в теле функции. Имя параметра шаблона видно во всем определении и скрывает другие использования того же идентификатора в области, глобальной по отношению к данному шаблону функций. Если внутри тела определяемой функции необходим доступ к внешним объектам с

тем же именем, нужно применять операцию изменения области видимости. Следующая программа иллюстрирует указанную особенность имени параметра шаблона функций:

```
#include <iostream>

int N = 0; //статическая, инициализирована нулем

template<class N> N max(N x, N y)
{
  N a = x;
  cout << "\nСчетчик обращений N = " << ++::N;
  if (a < y) a = y;
  return a;
}

int main(void)
{
  int a = 12, b = 42;
  max(a, b); //Счетчик обращений N = 1
  float z = 66.3, f = 222.4;
  max(z, f); //Счетчик обращений N = 2
}
```

Итак, одно имя нельзя использовать для обозначения нескольких параметров одного шаблона, но в разных шаблонах функций могут быть одинаковые имена у параметров шаблонов. Ситуация здесь такая же, как и у формальных параметров при определении обычных функций, и на ней можно не останавливаться подробнее. Действительно, раз действие параметра шаблона заканчивается в конце определения шаблона, то соответствующий идентификатор свободен для последующего использования, в том числе и в качестве имени параметра другого шаблона. Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Таким образом, будет ошибочным такой шаблон:

```
template<class A, class B, class C>
B func(A n, C m)
{
  B value;
  ...
     }
```

В данном неверном примере остался неиспользованным параметр шаблона с именем B. Его применение в качестве типа возвращаемого функцией значения и для определения объекта value в теле функции недостаточно.

Определенная с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть

непараметризованно и возвращаемое функцией значение. Например, в следующей программе шаблон определяет семейство функций, каждая из которых подсчитывает количество нулевых элементов одномерного массива параметризованного типа:

```
#include <iostream>

template<class D> long count0(int, D *); //Прототип шаблона

int main(void)
{
  int A[] = { 1, 0, 6, 0, 4, 10 };
  int n = sizeof(A) / sizeof A[0];
  cout << "\ncount0(n,A) = " << count0(n, A);
  float X[] = { 10.0, 0.0, 3.3, 0.0, 2.1 };
  n = sizeof(X) / sizeof X[];
  cout << "\ncount0(n,X) = " << count0(n, X);
}

template<class T> long count0(int size, T *array)
{
  long k = 0;
  for (int i = 0; i < size; i++)
    if (int(array[i]) == 0) k++;
  return k;
}
```

6. В шаблоне функций count0 параметр T используется только в спецификации одного формального параметра array. Параметр size и возвращаемое функцией значение имеют явно заданные непараметризованные типы. Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве описания шаблона функций используется прототип шаблона:

template<список_параметров_шаблона>

7. В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.
8. При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми. Для определенного выше шаблона функций с прототипом

```
template<class E> void swap(E, E);
```

недопустимо использовать такое обращение к функции:

```
int n = 4;
double d = 4.3;
swap(n, d); // Ошибка в типах параметров
```

Для правильного обращения к такой функции требуется явное приведение типа одного из параметров. Например, вызов

```
swap(double(n), d); // Правильные типы параметров
```

приведет к конкретизации шаблонного определения функций с параметром типа double. При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определенная функция. В обоих случаях "распознавание" конкретного вызова выполняется по сигнатуре, т.е. по типам, порядку и количеству фактических параметров.

Вопросы

1. Что такое шаблон?
2. Для чего может быть определен шаблон?
3. Расскажите о назначении шаблонов.
4. Приведите синтаксис объявления шаблона функции.
5. Приведите синтаксис объявления шаблона класса.
6. Может ли быть список параметров шаблона пустым?
7. Есть ли ошибка в определении шаблона?

   template <class T, class U> void foo (T*, U);
8. Может ли быть непараметризовано возвращаемое шаблоном функции значение? Приведите пример.
9. Можно ли перегружать шаблоны функции?
10. Есть ли ошибка в определении шаблонов?

    template <class T> T min (T* t1, T t2, T t3)

    template <class T> T min (T t1, int t2)
11. Для чего используется ключевое слово typename?
12. Запишите шаблон функции сравнения переменных.
13. Запишите шаблон класса массив.
14. Запишите шаблон класса функция.
15. Могут ли использоваться как шаблонные дружественные функции, которые определены в шаблоне класса?
16. Можно ли определить следующую функцию в шаблоне?

```
template <class T> class vector
{
        virtual   void   insert(T*);
}
```

17. Можно ли в шаблоне класса определить friend-шаблон класса?
18. Может ли обычный класс содержать шаблонные классы?
19. Могут ли шаблоны быть производными от шаблонов?
20. Могут ли шаблоны быть производными от классов?
21. Приведите пример передачи в шаблон класса дополнительных параметров.
22. Может ли быть параметром шаблона тип данных int?
23. Может ли быть параметром шаблона перечисляемый тип?
24. Может ли обычный класс быть производным от шаблона класса?
25. Что выведет следующий код?

```
class Rose {};

class A { public: typedef Rose rose; };

template<typename T>
class B: public T { public: typedef typename T::rose foo; };

template<typename T>
void smell(T) { std::cout << "Жуть!" << std::endl; }
void smell(Rose) { std::cout << "Прелесть!" << std::endl; }

int main()
{
   smell(A::rose());
   smell(B<A>::foo());
   return 0;
}
```

26. Какие из объявлений шаблона функции верны ?

```
template <class T1, class T2, class T3>
T3 func(T1, T2)
{
...
}

template <typename T1, typename T2, typename T3>
T1 func(T3, T2)
```

```
{
...
}

template <class T1, T2, class T3>
T2 func(T1, T3)
{
...
}

template <class T>
T func(T, T)
{
...
}

template <typename T, typename T>
T func(T, T)
{
...
}

template <class T, int sz>
T func(const T (&arr)[sz])
{
...
}
```