МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра информатики и вычислительной математики

ПРОГРАММИРОВАНИЕ В С ++

Утверждено редакционно-издательским советом университета в качестве методических рекомендаций

Самара
Издательство «Самарский университет»
2012

УДК 004.432
ББК 32

Рецензент канд. техн. наук, доцент В. В. Пшеничников

Программирование в C++: методические рекомендации / сост. М. С. Русакова – Самара: Изд-во «Самарский университет», 2012. – 28 с.

Методические рекомендации содержат авторские материалы для организации самостоятельной работы студентов по разделу курса «Программирование в C++ с использованием шаблонов». В них освещаются тонкости шаблонного программирования, особенности применения шаблонов при программировании на C++. Все темы снабжены авторскими примерами кода, наглядно иллюстрирующими удобства, преимущества и особенности использования шаблонов. В рекомендациях содержатся задания для самостоятельного выполнения и список литературы для самостоятельной более глубокой проработки вопросов шаблонного программирования на C++.

Методические рекомендации предназначены для студентов специальностей «Прикладная математика и информатика», «Математическое обеспечение и администрирование информационных систем», «Компьютерная безопасность» при изучении дисциплин специализации и подготовке к производственной практике.

УДК 004.432
ББК 32

Шаблоны

Повторное использование кодов — один из краеугольных принципов ООП (объектно-ориентированного программирования). Композиция и наследование – естественные способы повторного использования объектных кодов. Но часто возникают ситуации, когда приходится много раз переписывать одни и те же исходные тексты, в то время как хорошему программисту хотелось бы повторно их использовать. Например, зачем 3 раза переписывать метод сортировки массива целых чисел, массива вещественных или массива символов? Вполне естественно желание программиста один раз написать некий алгоритм, позволяющий произвести сортировку массива элементов какого-либо произвольного типа (определение которого мы пока что немного отложим), а потом 3 раза применить его к соответствующим конкретизированным массивам.

Итак, перед нами возникла проблема: необходимо выполнить сходные действия (сортировку) над различными типами данных (массивы элементов различного типа). При этом мы желаем лишь однажды написать соответствующий код программы, который впоследствии может быть использован повторно для других типов данных. Решением поставленной задачи могут служить **шаблоны** (а точнее, шаблоны функций).

Шаблоны функций

Шаблоны функций фактически представляют собой алгоритм, который объясняет, как надо действовать. При этом конкретизация типа данных, над которыми производятся действия, отложена.

Синтаксис:

```
template <class имя_формального_типа1[, class имя_ф_т2…]>
тип_возвращаемого_значения имя_функции (список аргументов)
{тело_функции;}
```

Описание шаблона функции начинается с ключевого слова template, после которого в угловых скобках перечисляются имена формальных типов данных, предваренные ключевым словом class. Далее следует обычное описание функции, однако, в нем обязательно хотя бы один раз должно присутствовать имя формального типа. Формальный тип означает, что при работе программы вместо его имени можно подставить любой из примитивных (int, float, char, int*, float*…) или пользовательских типов данных. Тогда на базе одного шаблона функции будет генерироваться код для целого семейства перегруженных функций, которые различаются типами своих аргументов (и, возможно, типом возвращаемого значения).

Приведем пример шаблона функции сортировки массива и поясним принцип работы с шаблоном функции.

Пример

```
// файл основной программы,
// демонстрирующей возможности шаблонов функций
#include <iostream.h>
#include <iomanip.h>

//шаблон функции сортировки массива
template  <class TItem> void sort (TItem *arr, int size)
{
for (int i=0; i<size; i++)
     {TItem Min=arr[i];
          int iMin=i;
          for (int j=i+1; j<size;j++)
               {if (arr[j]<Min)
                    {Min=arr[j];
                    iMin=j;}}
          arr[iMin]=arr[i];
          arr[i]=Min;}}

// шаблон функции вывода
template <class TItem, int pos>
          void print (TItem *arr, int size)
{    for (int i=0; i<size;i++)
     cout<<setw(pos)<<arr[i]<<" ";
     cout<<endl;}

int main()
{    int* arInt;
     float *arFloat;
     int size;
     cout<<"Enter array size: "<<endl;
     cin>>size;

     arInt=new int [size];
     arFloat=new float[size];

     for (int i=0; i<size;i++)
     {    arInt[i]=100-rand()%100;
          arFloat[i]=(float)(rand()%100)/
                         ((float)(rand()%100+1));}

     cout.precision(3);
     // генерация кода для печати массива целых
     print<int, 4>(arInt,size);
     print<float, 6>(arFloat, size);//и вещественных чисел
     // генерация кода для сортировки массива целых
     sort(arInt, size);
     // и для сортировки массива вещественных
     sort(arFloat,size);
```

```
    // генерация кода для печати массива целых
    print<int,4>(arInt,size);
    // и для массива вещественных
    print<float, 6>(arFloat, size);
    return 0;}
```

Вывод программы:

```
Enter array size:
5
59   100  22   36   19
1.91   2.76   0.921   0.109   0.435
19   22   36   59   100
0.109   0.435   0.921   1.91   2.76
```

В программе объявлены 2 шаблона функций. Остановимся подробнее на первом из них. Шаблон функции сортировки устанавливает, что будет работать с одним формальным типом TItem. При объявлении шаблона говорится, что в заголовок функции передается указатель на тип TItem, а в теле сортировки мы работаем с элементами массива, имеющими тот же формальный тип TItem. Далее, в теле программы мы создали 2 динамических массива: arrInt для хранения целых и arrFloat для хранения вещественных. Вызов sort(arInt, size) приводит к поиску функции sort с параметрами (int*, int), однако таковой в явном виде нет. Зато у нас описан шаблон sort(TItem*, int), на основании которого генерируется нужный код функции (он получается подстановкой типа «int» на место «TItem»). Аналогичная ситуация происходит и в том случае, когда вызывается функция сортировки для массива вещественных чисел. Только теперь объявление «TItem» везде будет замещаться типом «float».

Шаблоны и нетиповые параметры

В предыдущем примере присутствуют 2 шаблона функций: первый из них мы рассмотрели, а второй представляет особый интерес, т.к. в нем присутствует нетиповой параметр (int pos).

```
template <class TItem, int pos>
    void print (TItem *arr, int size)
```

Нетиповой параметр означает, что под ним не скрывается никакого имени типа, фактически мы работаем с некой переменной, тип которой известен сразу, а ее значение задается в самой программе. Оно фиксируется еще на стадии компиляции, и его можно рассматривать как константу. В приведенном примере в качестве нетипового параметра задано значение ширины поля вывода элемента массива. Если ранее мы использовали шаблонные функции неявно (компилятор по типу параметров сам определял, какой код ему генерировать), то для шаблона с нетиповым параметром надо явно указать и подставляемый тип, и значение константы. Именно поэтому в программе встречается вызов вида

```
print<int, 4>(arInt, size);
```

Значение типа в таком вызове подставляется «int», а значение константы – «4».

Разумеется, шаблоны функций – далеко не единственный способ повторно использовать исходный текст программы. Приведенный выше пример хорош, но он всего лишь иллюстрирует, как можно улучшить структурное программирование. А нас интересует объектный подход... Возникает вопрос: нельзя ли каким-либо образом применить идеологию шаблонов к классам? Для решения рассмотренной задачи можно было бы организовать класс «Массив», в котором логично было бы реализовать алгоритм простой сортировки, а потом на базе этого класса создать экземпляры массивов с различными типами элементов. Эту идею легко можно реализовать, если воспользоваться шаблоном класса.

Шаблон класса

Прежде чем приступить к реализации шаблона класса «Массив», обсудим подробнее возникшую перед нами задачу. Итак, мы решили хранить данные в массиве, для которого предусмотрены методы заполнения, сортировки и вывода на печать. При этом тип данных может быть любым. Иными словами, нам нужно «нечто», позволяющее хранить в себе данные произвольного типа, но при этом имеющее один интерфейс для работы с любыми типами данных и позволяющее работать с ними по строго определенным алгоритмам. Это необходимое нам «нечто» – контейнер.

Интуитивно мы пришли к выводу, что создание контейнера в C++ базируется на идеологии шаблонов. Насколько было оправдано введение нового подхода (имеются в виду шаблоны)? Нельзя ли было обойтись наследованием? Желание многократного использования кодов привело создателей языков Smalltalk (а впоследствии и Java) к подходу однокоренной иерархии классов. В C++ такого нет – каждый может писать свои собственные типы данных, которые не будут являться наследниками какого-то одного класса Object или какого-либо еще. В связи с этим возникали проблемы совместимости: как в контейнер для какого-то объекта поместить экземпляры классов, не связанные с ними иерархией наследования? Напрашивалось решение: сделать множественное наследование и от классов, объекты которых надо помещать в контейнер, и от самого класса-контейнера. Такая объектно-базированная иерархия крайне быстро становилась запутанной и громоздкой, в связи с чем применять ее было неудобно. Б. Страуструп в первом издании своей книги привел альтернативное решение. Согласно ему, контейнерные классы создавались в виде больших препроцессорных макросов с аргументами, вместо которых надо было подставить нужный тип. Эта идеология уже

гораздо ближе к шаблонам, не правда ли? Сам контейнер создавался несколькими вызовами макросов. Однако такой подход оказался не слишком удобен, в связи с чем в скором времени был заменен другим механизмом подстановки типов, за который теперь был ответственен компилятор (а не препроцессор). Этот механизм подстановки типов и есть шаблон (реализующий концепцию параметризованного типа).

Синтаксис: `template <class имя_формального_типа>`
`class имя_класса {тело_класса};`

Как и для шаблонов функций, имя формального типа должно хотя бы раз встретиться в описании класса. Иначе компилятор выдаст вам сообщение об ошибке.

Итак, разработаем шаблон класса «Массив», обсуждаемый выше.

Пример

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <fstream.h>
// шаблон класса «Массив»
template <class TItem> class array
{public:    array(const char * = "int.txt");
          ~array();
          void print();
          void sort();
 private:    TItem* top;
          int size;};
// реализация метода сортировки из шаблона класса
template <class TItem> void array<TItem>::sort()
{for (int i=0; i<size; i++)
     {    TItem Min=top[i];
          int iMin=i;
          for (int j=i+1; j<size;j++)
          (if (top[j]<Min)
               {    Min=top[j];
                    iMin=j;}}
          top[iMin]=top[i];
          top[i]=Min;}}
// реализация метода вывода из шаблона класса
template <class TItem> void array<TItem>::print()
{    for (int i=0; i<size; i++)
     cout<<top[i]<< " ";
     cout<<endl;}
// реализация конструктора из шаблона класса
template <class TItem> array<TItem>::array(const char *
fName)
{    ifstream file(fName);
     file>>size;
```

```
    top = new TItem[size];
    for (int i=0; i<size;i++)
    file>>top[i];
    file.close();}
// реализация деструктора из шаблона класса
template <class TItem> array<TItem>::~array()
{delete top;}
int main()
{    array<int> testInt;
    testInt.print();
    testInt.sort();
    testInt.print();
    cout<<"---------"<<endl;
    array<float> testFloat("float.txt");
    testFloat.print();
    testFloat.sort();
    testFloat.print();
    cout<<"---------"<<endl;
    array<char> testChar("char.txt");
    testChar.print();
    testChar.sort();
    testChar.print();
    cout<<"---------"<<endl;
    return 0; }
```

Вывод программы:

```
9 23 4 -2 234
-2 4 9 23 234
---------
3.14 -0.234 3.14159 234.232 -234.33
-234.33 -0.234 3.14 3.14159 234.232
---------
f g s n e
e f g n s
---------
```

В приведенном коде следует обратить внимание на несколько моментов. Во-первых, при описании неподставляемых функций в шаблоне класса следует аккуратно произвести разрешение имен. Поэтому заголовок функции предваряется выражением вида

```
template <class TItem> void array<TItem>::sort(),
```

т. е. сначала указывается ключевое слово template (описываем реализацию шаблона), затем перечисляются имена формальных типов, которые встречаются в шаблоне, затем указывается тип возвращаемого из функции значения и область видимости со спецификацией формального типа (или типов) в угловых скобках. Далее после операции разрешения области действия следует имя функции и список ее аргументов.

В теле основной программы создаем экземпляры на основании шаблонных классов. При этом вместо имени формального типа подставляем конкретные типы данных, например,

```
array<float> testFloat("float.txt");
```

Таким образом, мы определили интерфейс и функциональность для работы с набором элементов произвольного типа.

Стоит отметить, что подобная реализация шаблона класса «Массив» не претендует на истину в последней инстанции. Например, ничто не мешало использовать в шаблоне константу для задания размерности массива (а точнее, нетиповой параметр, подобный тому, который был рассмотрен в шаблонах функций). Модифицируем шаблон класса «Массив» так, чтобы размер массива задавался нетиповым параметром шаблона.

```
template <class TItem, int dim> class array
{public:
        array(const char * = "int.txt");
        ~array();
        void print();
        void sort();
 private:
        TItem* top;
        int size;};
```

В списке формальных типов в описании шаблона появился нетиповой параметр dim целого типа. Описание функций-элементов изменится тем, что будет предваряться более сложной конструкцией вида:

```
template <class TItem, int dim>
        void array<TItem, dim>::sort().
```

Изменим также конструктор класса, в котором размер массива будет устанавливаться равным значению константы шаблона:

```
template <class TItem, int dim>
                array<TItem, dim>::array(const char *
                fName)
{    ifstream file(fName);
     size=dim;
     top = new TItem[size];
     for (int i=0; i<size;i++)
     file>>top[i];
     file.close();}
```

Тогда создание экземпляров класса «массив 10 целых», «массив 12 вещественных», «массив 15 символов» будет выглядеть как

```
array<int, 10> testInt;
array<float, 12> testFloat("float.txt");
array<char, 15> testChar("char.txt");
```

В отличие от шаблонов функций, в которых нельзя было задать значение нетипового параметра по умолчанию (подумайте, почему?), для

шаблонов классов такого запрета нет. Поэтому, мы можем еще раз изменить описание шаблона класса array, задав константе значение по умолчанию:

```
template <class TItem, int dim=10> class array {…}
```

Тогда создание экземпляра класса «массив 10 целых» будет выглядеть крайне просто: `array<int> testInt;`

Отметим, что константа является параметром шаблона, следовательно, экземпляры «массив 5 целых чисел» и «массив 10 целых чисел» будут создаваться на основании различных шаблонных классов! Чтобы избежать генерации «лишнего» кода внимательно обращайтесь с нетиповыми параметрами в шаблонах.

Разработаем теперь контейнер, реализующий возможности однонаправленного линейного списка. Вообще говоря, для линейного однонаправленного списка стоило бы предусмотреть возможности добавления элемента в начало и конец списка, удаление из начала и конца списка, можно было бы организовать проверку на заполненность структуры данных и сортировку элементов списка и т.п. Однако в целях экономии времени в рассматриваемом примере ограничимся методами добавления в конец и удаления элементов из конца списка, печати содержимого и проверки, не пуст ли список.

Чтобы контейнер «линейный список» мог хранить элементы любых типов, логично организовать его в виде шаблона. Однако каждый элемент линейного списка в свою очередь представляет собой строго организованную структуру – узел списка, который содержит одно или более информационных полей и указатели на следующий узел. Соответственно, логично было бы описать шаблон «узел списка», чтобы на его основе можно было сгенерировать семейство узлов, хранящих данные нужного нам типа.

Раз мы пришли к выводу о том, что нам придется работать с двумя шаблонами, надо понять, будут ли эти шаблоны связаны друг с другом отношениями наследования или композиции, или, быть может, отношениями дружественности? Очевидно, что для организации списка придется в шаблоне «Список» указать узел-голову списка (а может быть и узел-хвост списка для удобства реализации). Соответственно, два шаблона будут связаны друг с другом отношениями композиции. Кроме того, логично было бы в списке хранить **закрытый** указатель на головной узел, чтобы защитить список от возможного изменения внешними методами. Тогда возникает следующая проблема: мы включили узел в список как закрытый элемент, хотим воспользоваться методами работы из шаблонного класса «Узел списка», но не имеем к ним доступа. Решить проблему можно, если объявить шаблон «Список» другом шаблона «Узел списка», чтобы получить полный доступ к методам и данным. Но как будут работать правила дружественности с шаблонами?

Шаблоны и правила дружественности

Отношения дружественности можно установить между шаблоном класса и глобальной функцией, функцией-элементом другого класса (возможно, шаблонного), классом (возможно, шаблонным), и шаблоном. Пусть объявлен шаблон `template <class T> class X {…};`

1. Тогда если внутри такого шаблона есть запись `friend void f();` то функция f() является дружественной для всего шаблона, а следовательно для всех сгенерированных на его основе классов.
2. Если в шаблоне встречается запись вида `friend void f(X<T>&);` то функция f(X<T>&) будет дружественной только для конкретного шаблонного класса, т.е. для класса X <int> будет дружественная f(X<int>&), для класса X<float> другом будет f(X<float>&) и т.д.
3. Если в шаблоне объявлено `friend void otherClass :: f();` то это означает, что функция-элемент f() из класса otherClass будет являться другом всего шаблона класса X (т.е. всех сгенерированных на его основе классов).
4. Если в шаблоне класса X есть объявление дружественности вида
   ```
   friend void otherClass<T> :: f(X<T>&);
   ```
   то метод f(X<T>&) из шаблонного класса otherClass<T> будет являться другом только для конкретного шаблонного класса X<T>, т.е. f(X<float>&) из шаблонного класса otherClass<float> будет другом шаблонного класса X<float>.
5. Если в шаблоне класса X встречается объявление
   ```
   friend class testFriendshipClass;
   ```
   то все методы класса testFriendshipClass будут являться друзьями всех шаблонных классов X.
6. Если же нам надо «подружить» два шаблона, то в описании шаблона класса X должно быть добавлено объявление вида
   ```
   friend class testFriendshipTempl<T>;
   ```
   Тогда все методы шаблонного класса testFriendshipTempl<T> будут являться друзьями шаблонного класса X<T>, т.е. методы testFriendshipTempl<int> будут дружить с классом X<int> и т.п.

Вернемся к созданию контейнера «Линейный список», который, как мы обсудили выше, будет строиться на основании двух шаблонов, связанных отношениями композиции и дружественности.

Пример

```
//*********************************************************
//заголовочный файл с описанием шаблона «Узел списка»
//*********************************************************
#ifndef TEMPLLISTNODE_H
#define TEMPLLISTNODE_H
#include <iostream.h>
```

```
// необходимо объявить существование шаблона,
//после чего можно приступать к объявлению дружественности
template <class TItem> class List;
template <class TItem> class ListNode
{// объявляем дружественный шаблонный класс
friend class List<TItem>;
public:  ListNode(TItem);
         ~ListNode();
         void setInfo(TItem);
         TItem getInfo() const;
private:    TItem info;
         ListNode* next;};
template <class TItem>
         ListNode<TItem>::ListNode(TItem item)
{   info=item;
    next=0;}
template <class TItem>
         ListNode<TItem>::~ListNode()
{cout<<"Delete ListNode with item value "<< info<<endl;}
template <class TItem>
         void ListNode<TItem>::setInfo(TItem item)
{info=item;}
template <class TItem>
         TItem ListNode<TItem>::getInfo() const
{return info;}
#endif // TEMPLLISTNODE_H
//**********************************************************
//заголовочный файл
//с описанием шаблона класса «Линейный список»
//**********************************************************
#ifndef LIST_H
#define LIST_H
#include "templListNode.h"
template <class TItem> class List
{public:  List();
         ~List();
         void insEnd(TItem);
         void remEnd();
         void print();
         int isEmpty();
private:    ListNode<TItem> *head;
         ListNode<TItem> *tail;};
template <class TItem> List<TItem>::List()
{    head=0;
    tail=head;}
template <class TItem>
         void List<TItem>::insEnd(TItem item)
```

```
{if (isEmpty())
          {head = new ListNode<TItem>(item);
               tail=head;}
     else
          {ListNode<TItem> *temp=
                              new ListNode<TItem>(item);
               tail->next=temp;
               tail=temp;}}
template <class TItem> void List<TItem>::remEnd()
{if (!isEmpty())
          {if (head!=tail)
                    {   ListNode<TItem> *temp = tail,
                                        *stepPtr=head;
                        while (stepPtr->next!=tail)
                             stepPtr=stepPtr->next;
                        stepPtr->next=0;
                        tail=stepPtr;
                        delete temp;}
               else {   ListNode<TItem> *temp = tail;
                        head=0;
                        tail=head;
                        delete temp;}}}
template <class TItem> void List<TItem>::print()
{    if (!isEmpty())
          {    ListNode<TItem> * temp=head;
               while (temp->next!=0)
                    {    cout<<temp->info<<"->";
                         temp=temp->next;}
               cout<<temp->info<<endl;}
     else cout<<"The list is empty!"<<endl;}
template <class TItem> int List<TItem>::isEmpty()
{    if (head==0) return 1; else return 0;}
template <class TItem> List<TItem>::~List()
{    while (!isEmpty()) remEnd();}
#endif // LIST_H
//************************************************************
//основная программа, тестирующая возможности контейнера
//************************************************************
#include <conio.h>
#include "templlistnode.h"
#include "list.h"
int main()
{    // создаем новый указатель на список целых
     List<int> *testList=new List<int>();
     testList->print(); // удостоверяемся, что список пуст
     for (int i=0;i<=10;i++) // заполняем список
     testList->insEnd(i);
     testList->print(); // печатаем заполненный список
```

```
    testList->remEnd(); //удаляем элемент из конца списка
    testList->print(); // и снова печатаем список
    if (testList->isEmpty()) cout<<"Empty"<<endl;
    else cout<<"Not empty"<<endl;
    delete testList; // освобождаем память из-под списка
return 0;}
```

Вывод программы:

```
The list is empty!
0->1->2->3->4->5->6->7->8->9->10
Delete ListNode with item value 10
0->1->2->3->4->5->6->7->8->9
Not empty
Delete ListNode with item value 9
Delete ListNode with item value 8
Delete ListNode with item value 7
Delete ListNode with item value 6
Delete ListNode with item value 5
Delete ListNode with item value 4
Delete ListNode with item value 3
Delete ListNode with item value 2
Delete ListNode with item value 1
Delete ListNode with item value 0
```

Итак, в приведенном выше примере стоит обратить особое внимание на предварительное объявление прототипа шаблона **template <class** TItem> **class** List; в заголовочном файле с описанием шаблона класса «Узел списка». Дело в том, что без такого упреждающего объявления компилятор не сможет понять, что мы объявляем другом класса шаблон, который будет описан далее. Только с таким порядком объявлений программа откомпилируется и будет работать. Дальнейший код программы очевиден и каких-либо особых пояснений не требует. Остановимся чуть более подробно на вопросах принадлежности объектов контейнеру.

Управление принадлежностью

В случае если объекты в контейнере хранятся по значению, проблем с принадлежностью возникнуть не может, т.к. все объекты принадлежат самому контейнеру и он ответственен за их корректное удаление. Однако если в контейнере хранятся указатели, то возникает неоднозначность. Самому контейнеру объекты могут быть уже не нужны, и он может попытаться их удалить, но указатели могут использоваться где-то еще, допустим, в другом контейнере. Тогда возникает вопрос: как определить принадлежность объекта?

Рассмотрим проблему на примере нашего контейнера «Линейный список». В нем хранится 2 указателя – на головной и хвостовой узлы

списка. Вообще говоря, ничего не запрещает нам, используя копирующий конструктор, создать один список на основе другого, тогда один и тот же указатель будет одновременно использоваться двумя (или более) контейнерами. Например, если мы добавим в описание шаблона «Список» такие методы

```
List(ListNode<TItem>&);
ListNode<TItem> *getHead();
```

с реализацией

```
template <class TItem>
        List<TItem>::List(ListNode<TItem>& h)
{    head=&h;
     tail=head;}
template <class TItem>
        ListNode<TItem>* List<TItem>::getHead()
{return head;}
```

то в основном коде программы мы можем попытаться создать нечто похожее:

```
int main()
{    List<int> *testList=new List<int>();
     testList->insEnd(123);
     List<int> *intList=
                  new List<int>(*(testList->getHead()));
     delete testList;
     cout<<"del intList"<<endl;
     delete intList;
     return 0;}
```

Но тогда программа выдаст неожиданный результат:

```
Delete ListNode with item value 123
Список удален
del intList
Delete ListNode with item value 206936
HEAP[class_template.exe]:
Invalid Address specified to RtlFreeHeap( 00030000,
00032898 )
```

Дело в том, что оба списка были построены на основании одного и того же указателя, который мы удалили при освобождении первого контейнера, в связи с чем и произошла указанная ошибка. А уж если вы во второй список добавите что-то еще, а потом удалите первый список... То вообще получите «зависший» указатель.

Устранить неприятность можно, если ввести аргумент конструктора, отвечающий за принадлежность объекта. Как правило, при этом в интерфейсе контейнера предусматриваются методы управления принадлежностью и флаг принадлежности. Кроме того, можно связать флаг принадлежности с каждым элементом контейнера, чтобы решение об уничтожении принималось каждый раз для каждого объекта контейнера.

Модифицируем копирующий конструктор шаблона «Линейный список»: `List (ListNode<TItem>&, bool);` и введем логическую переменную, ответственную за принадлежность **private**: `bool flag;`

Изменим также реализацию конструкторов и деструктора

```
template <class TItem> List<TItem>::List()
{    head=0;
     tail=head;
     flag=true;}
template <class TItem>
          List<TItem>::List(ListNode<TItem>& h, bool f)
{    head=&h;
     tail=head;
     flag=f;}
template <class TItem> List<TItem>::~List()
{    if (flag)
     while (!isEmpty())
     remEnd();
     cout<<"Список удален"<<endl;}
```

Тогда код основной программы будет выглядеть как

```
int main()
{   List<int> *testList=new List<int>();
     testList->insEnd(123);
     List<int> *intList=
               new List<int>(*(testList->getHead()),false);
     cout<<"del intList"<<endl;
     delete intList; // удаление не происходит!!!!
     cout<<"-------------"<<endl;
     cout<<"del testList"<<endl;
     delete testList;
     return 0;}
```

Вывод программы:

```
del intList
Список удален
-------------
del testList
Delete ListNode with item value 123
Список удален
```

Шаблоны как параметры шаблона, вложенные шаблоны

Когда мы говорили о нетиповых параметрах в шаблонах классов (и функций), то ограничились только упоминанием целочисленных параметров. Однако в C++ реализована возможность записать один шаблон в качестве параметра другого шаблона. Например, можно было бы попробовать контейнер, описанный выше, организовать как шаблон с параметром, который был бы шаблоном узла этого списка (довольно

экзотический способ программирования, однако для решения прикладных задач возможностями шаблонов как параметров пренебрегать все же не стоит).

Объявление шаблона параметром выглядит следующим образом:

```
template <class T, template <class> class Seq>
        class Container {…};
```

Итак, мы объявляем шаблон контейнера, в котором присутствует 2 параметра: первый из них – формальный тип данных, а второй – шаблон класса Seq, использующий в свою очередь один формальный тип данных. В реализации шаблона контейнера обязательно в начале должна присутствовать строка-объявление класса Seq шаблоном:

```
template <class T, template <class> class Seq>
        class Container
        {Seq<T> var1;…};
```

В основной программе тогда при создании объекта типа «Контейнер» в качестве шаблона-параметра указывается имя настоящего шаблона, который есть в программе. Например,

```
Container <float, ListNode> cont;
```

Еще одним интересным применением шаблонов является использование вложенных шаблонов для упрощения приведения типов данных. Рассмотрим такой пример: пусть мы хотим описать шаблон класса комплексных чисел, вещественная и мнимая части которых могут быть целыми или вещественными числами (этот тип и будет являться формальным типом шаблона класса). Но для решения общих задач нам желательно иметь копирующий конструктор, который позволял бы создать комплексное число на базе существующего, да еще хотелось бы, чтобы при таком создании можно было сделать автоматическое приведение типов (скажем, от float в исходном числе к double в копии). Такую возможность реализуют вложенные шаблоны. Иными словами, наш пример может выглядеть следующим образом:

```
template <class T> class complex
{     public:
      template <class X> complex(const complex<X>&);…};
```

Реализация такого шаблона-конструктора вне класса выглядит довольно хитро, т.к. компилятору нужно сообщить, что мы описываем реализацию шаблона, спрятанную внутри другого шаблона.

```
template <class T>
      template <class X>
          complex<T>::complex(const complex<X>& nmb)
                    {тело_функции;}
```

Теперь в программе можно создать комплексные числа вида

```
complex <float> z(1,2);
complex <double> x(z);
```

Подобные вложенные шаблоны применяются в стандартных библиотеках C++ (для преобразования типов, а также для инициализации контейнеров друг другом). Кроме вложенных шаблонов функций разрешается применение вложенных шаблонов класса. На вложенные шаблоны функций налагается ограничение – их нельзя делать виртуальными, т.к. тогда компилятор не сможет на стадии компиляции создать таблицу виртуальных функций.

Характеристики

Концепция характеристик впервые была предложена Натаном Майерсом (Nathan Myers). Характеристики предназначены для группировки объявлений, зависящих от типа. Поясним данную концепцию на примере.

Рассмотрим такое понятие как «Специальность», на которой учится студент. Характеристиками этой специальности могут являться наборы экзаменов в сессию (т. к. они различны для студентов разных специальностей). Поэтому мы можем выделить каждый предмет в описание своего собственного класса-характеристики. Аналогично можем поступить и с описанием специальности, т.к. ее можно рассматривать как характеристику студента.

Чтобы связать между собой набор предметов, сдаваемых в сессию и специальность, можно воспользоваться шаблоном характеристики, который явно специфицируется для каждого набора экзаменов в сессии. Отметим, что такой подход не мешает создать нестандартный набор экзаменов (не основанный на шаблоне характеристик). Для этого достаточно описать класс (или новый шаблон), в котором перечисляются типы экзаменов для «нестандартной» специальности.

Чтобы связать студента с набором сдаваемых экзаменов, можно ввести еще один шаблон, описывающий студента и использующий характеристику специальности. Таким образом, мы получаем гибкое связывание типов и значений с контекстами, использующими их.

Пример

```
// заголовочный файл с описанием характеристик
#ifndef SPEC_H
#define SPEC_H
# include <iostream.h>
// Классы предметов - характеристики специальностей
class Delphi{
friend ostream& operator<<(ostream& out, const Delphi&)
{return out<<"Программирование в Delphi;"<<endl;}};
class CPP{
friend ostream& operator<<(ostream& out, const CPP&)
{return out<<"Программирование в C++;"<<endl;}};
```

```
class Java{
friend ostream& operator<<(ostream& out, const Java&)
{return out<<"Программирование на Java;"<<endl;}};
class FAN{
friend ostream& operator<<(ostream& out, const FAN&)
{return out<<"ФАН;"<<endl;}};
class UMF{
friend ostream& operator<<(ostream& out, const UMF&)
{return out<<"УМФ;"<<endl;}};
class MathAn{
friend ostream& operator<<(ostream& out, const MathAn&)
{return out<<"Мат. анализ;"<<endl;}};
//Классы специальностей
class AIS{
friend ostream& operator<<(ostream& out, const AIS&)
{return out<<"МОАИС."<<endl;}};
class PM{friend ostream& operator<<(ostream& out, const PM&)
{return out<<"ПМ."<<endl;}};
// Основной шаблон характеристик пуст,
// для каждой специальности конкретизируем его
template <class Spec> class Speciality;
template <> class Speciality <AIS>
{public: typedef Delphi exam_1;
     typedef CPP exam_2;
     typedef Java exam_3;};
template <> class Speciality <PM>
{public: typedef UMF exam_1;
     typedef MathAn exam_2;
     typedef Delphi exam_3;};
#endif // SPEC_H
// файл основной программы
#include "spec.h"
#include <string.h>
class Math // нестандартный класс характеристик
{public: typedef FAN exam_1;
     typedef MathAn exam_2;
     typedef UMF exam_3;};
//шаблон класса Студент, в нем используем характеристики
// Spec м. б. AIS или PM, exm - указан по умолчанию
template <class Spec, class exm=Speciality<Spec> >
class Student
{public: Student (const char* fio, const Spec& spc)
         {    FIO = new char [strlen(fio)+1];
              strcpy(FIO,fio);
              sp=spc;}
```

```
        void session()
        {    cout <<FIO
        <<" учится на специальности "<<sp<<endl;
        cout<<"В сессию сдает экзамены: "<<endl;
        cout<<e1<<e2<<e3<<endl;}
private:
    Spec sp;
    char* FIO;
    typedef typename exm::exam_1 exam_1;
    typedef typename exm::exam_2 exam_2;
    typedef typename exm::exam_3 exam_3;
    exam_1 e1;
    exam_2 e2;
    exam_3 e3;};
int main()
{   AIS sp1;
    PM sp2;
    Student<AIS> st1("Борисов Иван",sp1);
    Student<PM> st2("Горобец Егор",sp2);
    Student<PM,Math> st3("Дмитриев Иван",sp2);
    st1.session();
    st2.session();
    st3.session(); return 0;}
```

Вывод программы:

```
Борисов Иван учится на специальности МОАИС.
В сессию сдает экзамены:
Программирование в Delphi;
Программирование в C++;
Программирование на Java;
Горобец Егор учится на специальности ПМ.
В сессию сдает экзамены:
УМФ;
Мат. анализ;
Программирование в Delphi;
Дмитриев Иван учится на специальности ПМ.
В сессию сдает экзамены:
ФАН;
Мат. анализ;
УМФ;
```

Обсудим некоторые моменты. Классы Delphi, CPP, Java, FAN, UMF, MathAn – характеристики специальностей, которые мы ввели классами AIS и PM и которые в свою очередь являются характеристиками студента. Далее в программе появился шаблон характеристик Speciality, связывающий между собой предметы и специальности. Особенностью этого шаблона является то, что он использует один формальный тип Spec и является пустым. Далее мы конкретизируем (точнее, явно

специализируем) шаблон для каждого вида специальности PM и AIS (эти имена классов подставляются вместо имени формального параметра Spec). В конкретизации шаблона характеристик мы просто переопределяем имена типов сдаваемых предметов на имена экзамен1, экзамен2 и экзамен3. Теперь в зависимости от вида специальности имеется индивидуальный набор экзаменов.

Далее в основной программе вводим нестандартный набор экзаменов (нестандартную характеристику) по математическим дисциплинам. Такой подход нужен для того, чтобы проиллюстрировать работу еще одного шаблона характеристик – класса «Студент». Этот шаблон использует 2 формальных типа: тип Spec (вместо которого мы можем указать конкретную специальность) и формальный тип exm, заданный по умолчанию как специализированный шаблон характеристик (иными словами, под ним скрывается набор экзаменов для конкретной специальности, или нестандартный набор экзаменов для произвольной специальности). Особое внимание стоит обратить на переопределение типов вида `typedef typename exm::exam_1 exam_1;`
Здесь мы создаем псевдоним exam_1 для типа exam_1 из пространства имен exm (под формальным именем типа exm скрывается класс с набором экзаменов). Ключевое слово typename понадобилось, чтобы явно указать компилятору, что к exm надо относиться как к имени типа.

В основной программе созданы 2 специальности и 3 студента: первые 2 студента со стандартными характеристиками, и последний – с нестандартным. Далее для каждого студента вызывается метод session(), формирующий список экзаменов в соответствии с шаблонами характеристик.

Политики

В случае если необходимо связать функциональность с аргументами шаблонов (т.е. определить особое поведение в зависимости от типа данных), можно использовать классы политик в качестве аргументов шаблонов. Классы политик инкапсулируют в себе функциональность в виде статических функций, и в простейшем случае они могут быть обычными классами, а для мощного программирования классами политик могут быть шаблоны классов, или даже целые иерархии, включающие в себя полиморфизм. Рассмотрим применение политик в шаблонах на нашем примере. Введем классы «Сдать» в качестве родительского и «Сдать экзамен», «Сдать зачет» в качестве дочерних. Они и будут являться нашими классами политик.

```
// классы политик
class Pass
{public: static void pass(){});
```

```
class ExamPass: public Pass
{public:
static void pass(){cout<<"сдает экзамены: " <<endl;}};
class CreditPass: public Pass
{public: static void pass(){cout<<"сдает зачеты: "<<endl;}};
```

В основной программе модифицируем шаблон класса «Студент», который будет теперь использовать и характеристики, и политики.

```
template <class Spec,
          class WhatPass, class exm=Speciality<Spec> >
{...
    public: void session()
        {   cout <<FIO
            << " учится на специальности "<<sp<<endl;
            cout<<"В сессию ";
            WhatPass::pass(); // используем политики
            cout<<e1<<e2<<e3<<endl;}
            ...};
```

Многоточием обозначили здесь часть неизмененного кода. Теперь шаблон класса «Студент» использует 3 формальных типа данных, один из которых отвечает за классы политик. Если теперь в основной программе написать нечто вроде

```
int main()
{   AIS s1; PM s2;
    Student<AIS,CreditPass> st1("Борисов Иван",s1);
    Student<PM,ExamPass> st2("Горобец Егор",s2);
    Student<PM,CreditPass,Math> st3("Дмитриев Иван",s2);
    st1.session();
    st2.session();
    st3.session(); return 0;}
```

то вывод программы станет следующим:

```
Борисов Иван учится на специальности МОАИС.
В сессию сдает зачеты:
Программирование в Delphi;
Программирование в C++;
Программирование на Java;
Горобец Егор учится на специальности ПМ.
В сессию сдает экзамены:
УМФ;
Мат. анализ;
Программирование в Delphi;
Дмитриев Иван учится на специальности ПМ.
В сессию сдает зачеты:
ФАН;
Мат. анализ;
УМФ;
```

Таким образом, нам удалось разделить функциональность.

Шаблонное метапрограммирование

Где-то начиная с 1993 года, когда была введена поддержка простых шаблонных конструкций в C++, программисты столкнулись с весьма интересным поведением шаблонов. Рассмотрим простую программу, в которой определен шаблон для вычисления факториала в следующем виде:

```
#include <iostream.h>
template <int n>
struct Factorial
{enum {val=Factorial<n-1>::val*n};};
template <> struct Factorial<0>
{ enum {val=1};};
int main()
{    cout<<Factorial<5>::val<<endl;
     return 0;}
```

Программа выводит верный результат – 120. Однако считает она его еще на стадии компиляции. Действительно, чтобы сгенерировать шаблонную структуру Factorial<5> компилятору требуется знать шаблонную структуру Factorial<4>, и т.д. Специализация Factorial<0> останавливает нашу рекурсию. Получается, что все значения, присутствующие в структуре Factorial, построенной на основании шаблона, являются константами времени компиляции, а значит, их можно использовать как обычные константы и, например, указать их в качестве размерности массива. Получается, что мы обнаружили некий механизм программирования на стадии компиляции. Такие программы называют шаблонным метапрограммированием. Оно обладает полнотой по Тьюрингу, т.к. поддерживает выбор (if-else) и циклы (посредством рекурсии), а следовательно с его помощью можно выполнять любые вычисления.

Шаблоны выражений

В 1994 году Тодд Вельдхузен (Todd Veldhuizen) и Дэвид Вандеворде (Daveed Vandevoorde) независимо друг от друга ввели шаблоны выражений. Шаблоны выражений значительно оптимизируют код, сохраняя математическую запись выражений, а также позволяют реализовывать в C++ парадигмы и механизмы других языков программирования. Шаблоны выражений позволяют хорошо оптимизировать код вычислений, приближая его по скорости к оптимизированному коду на Фортране. При этом за счет механизма перегрузки операторов сохраняется естественная математическая запись. Шаблоны выражений заложены в основу многих высокопроизводительных математических библиотек.

С помощью шаблонов выражений можно избежать создания промежуточных объектов при математических вычислениях, а воспользоваться отложенными вычислениями, сохраняя ссылки на операнды. Желающие подробнее ознакомиться с парадигмой программирования, использующей шаблоны выражений, могут обратиться к книге Б. Эккеля и Ч. Элиссона «Философия C++. Практическое программирование», с. 242-247.

Разумеется, мы далеко не полностью осветили все аспекты применения шаблонов. Мы отметили основные возможности использования шаблонов – как удобного способа для организации контейнеров, как средства для организации удобочитаемого сокращенного программного кода, как способа управления функциональностью и связи между контекстами. За рамками обсуждения остались такие вопросы, как модели компиляции шаблонов, явная и частичная специализации шаблонов, подсчет объектов при помощи шаблонов, вопросы получения адресов сгенерированных шаблонных функций... Тем не менее, приведенного здесь материала достаточно для того, чтобы понять идеологию шаблонов и начать успешно их применять.

Практические задания

1. Используя шаблоны классов с самоадресацией, разработайте предложенные варианты программ. Выведите на экран построенные вами деревья. Постарайтесь написать программу наиболее структурированно, используйте несколько модулей, разделите интерфейс класса и его реализацию. Не забудьте про деструкторы!
    1. Двоичное дерево считается идеально сбалансированным, если для каждой его вершины количество вершин в левом и правом поддеревьях различается не более, чем на 1. Напишите функцию проверки идеальной сбалансированности двоичного дерева.
    2. Написать программу, определяющую, есть ли в дереве T хотя бы два одинаковых элемента.
    3. Дан текстовый файл. Построить дерево, используя элементы текстового файла в качестве ключей. Определить количество вершин дерева, содержащих слова, начинающиеся на одну и ту же букву.
    4. Написать программу, которая меняет местами максимальный и минимальный элементы непустого дерева.
    5. Написать программу, которая строит дерево, беря ключи из текстового файла. Определить наиболее часто встречающееся ключи.
    6. Написать программу, которая определяет число вхождений заданного элемента в дерево.

7. Написать программу, которая подчитывает число вершин на N-ном уровне непустого дерева (корень считать вершиной 0-го уровня)
8. Написать программу, которая находит максимальную глубину непустого дерева, т.е. число ветвей в самом длинном из путей от корня дерева до листьев.

2. Используя шаблоны, постройте подходящие контейнеры C++ (стеки, очереди, отображения и т.п.) и решите следующие задания.
   1. В заданном текстовом файле text1.txt хранятся пары значений ФИО – номер телефона. При этом ФИО может повторяться несколько раз. Используя подходящий контейнер, организуйте text2.txt в следующем виде: ФИО – список всех его телефонов. Отсортируйте по ФИО по возрастанию.
   2. Дан текстовый файл text1.txt. Требуется отсортировать все входящие в него слова по алфавиту и записать результат в файл text2.txt.
   3. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке файла находится запись вида: «фамилия» – «имя» – «отчество» – «год рождения». Записать в файл text2.txt количество однофамильцев. Записать в text3.txt всех ровесников (год рождения задается с клавиатуры).
   4. В файле text1.txt имеется список процессов Windows с указанием количества памяти, которую они занимают. В файле text2.txt имеется список процессов, которые мы дополнительно запускаем, также с указанием количества занимаемой памяти. Записать в файл text3.txt общий список процессов, отсортированный по количеству занимаемой памяти в порядке убывания.
   5. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке находится запись вида: «фамилия студента» – «изучаемая дисциплина». Названия дисциплин и фамилии студентов могут повторяться (в произвольном порядке). По заданной фамилии студента вывести все изучаемые им дисциплины в файл text2.txt и на консоль.
   6. В каждой строке файла text1.txt находится запись вида: «год выхода» – «наименование статьи» – «вид публикации». Годы и виды публикации (тезисы, статья, статья в журнале списка ВАК, монография…) могут повторяться в любом порядке. Отсортировать информацию по принципу: самый ранний год, потом все монографии, затем статьи в реферируемых журналах, потом статьи и тезисы. Такую сортировку произвести для каждого года. Результат записать в text2.txt.

7. Имеется текстовый файл text1.txt. Требуется исключить из него все дубликаты слов. Результат записать в text2.txt.
8. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке находится запись вида: «исполнитель» – «композиция» – «альбом» – «год выхода записи». В файл text2.txt записать все композиции одного исполнителя (можно задать с клавиатуры какого). В файл text3.txt записать все композиции из одного альбома (альбом вводится с клавиатуры).
9. Имеется текстовый файл text1.txt со следующей структурой: в первой строке указано число n, меньшее числа строк в файле. В остальных строках записаны имена. Построить программу-считалочку, которая выбрасывает каждый раз по одному имени на позиции n, если доходит до конца списка, то продолжает счет с начала. В файл text2.txt записать последовательность «выброса» имен и оставшегося игрока.
10. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке находится запись вида: «масть карты» – «старшинство». Количество строк в файле не превосходит 8. Определить, сколько карт каждой масти на руках у игрока. Найти количество карт одного достоинства для каждого вида старшинства карты. Результат записать в файл text2.txt.
11. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке находится запись вида: «масть карты» – «старшинство». В файле перечислена вся колода в произвольном порядке. Требуется отсортировать колоду по принципу следования мастей: червы, трефы, бубны, пики. Внутри каждой масти отсортировать карты по старшинству по возрастанию.
12. В текстовом файле text1.txt хранится информация следующего вида: количество человек в очереди в кассу, количество касс, время обслуживания в каждой кассе. Определить, за какое время очередь будет обслужена.
13. Имеется текстовый файл text1.txt со следующей структурой: в каждой строке файла записаны пары «слово» – «синоним». Заменить в заданном текстовом файле text2.txt все слова, встречающиеся в text1.txt, на синонимы.

Литература

1. Б. Эккель, Философия C++. Введение в стандартный C++. СПб.: Питер, 2004.
2. Б. Эккель, Ч. Элиссон. Философия C++. Практическое программирование. СПб.: Питер, 2004.
3. Страуструп Б. Язык программирования C++.- М.: Бином, 2006.
4. Дейтел Х., Дейтел П. Дж. Как программировать на C++. - М.: Бином-Пресс, 2006.
5. Русакова М. С. Программирование в С ++: практикум. Самара: Самарский университет, 2009.
6. Павловская Т. А. C/C++. Программирование на языке высокого уровня. СПб.: Питер, 2010.

Оглавление

Публикуется в авторской редакции
Титульное редактирование *Т. А. Мурзиновой*
Компьютерная верстка, макет *Н. П. Бариновой*

Подписано в печать 05.12.12. Формат 60x84/16. Бумага офсетная. Печать оперативная.
Усл.-печ. л. 1,6; уч.-изд. л. 1,75. Гарнитура Times.
Тираж 100 экз. Заказ № 2257.
Управление по информационно-издательской деятельности Самарского государственного университета: www.infopress.samsu.ru
Издательство «Самарский университет», 443011, г. Самара, ул. Акад. Павлова, 1.
Тел. 8 (846) 334-54-23
Отпечатано на УОП СамГУ