В.Л. Тарасов

Лекции по программированию на C++

Лекция 17

Шаблоны, исключения

Шаблон - это обобщенное описание функции или класса. На основе шаблона компилятор может автоматически создавать конкретные функции и конкретные классы. Использование шаблонов позволяет переложить на компилятор существенную часть рутинной работы по написанию многих схожих функций или классов.

17.1. Шаблоны функций

Шаблон функции представляет собой ее обобщенное описание, по которому автоматически создаются функции для работы с данными разных типов.

В определении шаблона семейства функций используются ключевое слово `template`. В приводимом далее примере объявлен шаблон функции `swap(T& x, T& y)` для обмена значений переменных `x` и `y` любого типа `T`.

Программа 17.1. Шаблон функции обмена значений переменных

```
// Файл SwapTemplate.cpp

template <class T>            // T – некий тип данных
void Swap(T& x, T& y)         // Шаблон функции Swap() для обмена
{                             // значений переменных x и y типа T
    T tmp;
    tmp = x;
    x = y;
    y = tmp;
}

#include <iostream>
#include <locale>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
```

```
    int ia = 1, ib = 2;
    cout << "ia = " << ia << " ib = " << ib << endl;
    Swap(ia, ib);                // Обмен значений переменных типа int
    cout << "ia = " << ia << " ib = " << ib << endl;
    double da = 3.14, db = 2.72;
    cout << "da = " << da << " db = " << db << endl;
    Swap(da, db);                // Обмен значений переменных типа double
    cout << "da = " << da << " db = " << db << endl;
    string sa = "12345", sb = "abcd";
    cout << "sa = " << sa << " sb = " << sb << endl;
    Swap(sa, sb);                // Обмен значений переменных типа string
    cout << "sa = " << sa << " sb = " << sb << endl;
    cin.get();
    return 0;
}
```

Программа выводит:

```
ia = 1 ib = 2
ia = 2 ib = 1
da = 3.14 db = 2.72
da = 2.72 db = 3.14
sa = 12345 sb = abcd
sa = abcd sb = 12345
```

В объявлении шаблона `T` – формальный параметр, обозначаемый ключевым словом `class`. Здесь слово `class` означает здесь «тип данных», и не обязательно класс, разработанный пользователем. Вместо слова `class` можно использовать ключевое слово `typename`. Параметры шаблона заключаются в угловые скобки (<) и (>). Шаблонную функцию `swap()` можно использовать для аргументов любых типов. При ее вызове `Swap(ia, ib);` автоматически создается функция для обмена переменных типа `int`:

```
void swap(int& x, int& y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Здесь вместо типа `T`, использованного в шаблоне, используется `int`.

Аналогично, для обмена значений типа `double` и `string` автоматически создаются еще две функции:

```
void swap(double& x, double& y){…}

void swap(string& x, string& y){…}
```

Без использования шаблона пришлось бы писать определения трех функций для трех разных типов данных, которые делают похожую работу.

Имя типа в шаблоне является *параметром*, вместо которого можно подставлять любые типы. В приведенном примере вместо формального параметра шаблона `T` использованы типы `int` и `double` и `string`.

Параметризовать можно типы аргументов функции и тип возвращаемого значения. Шаблон функции может иметь также и непараметризованные аргументы.

Программа 17.2. Шаблон функции экстремума

Шаблон функции `Extremum(T& x, T& y, int extr)` имеет аргумент целого типа `extr`, который определяет тип экстремума, возвращаемого функцией. Если значение `extr` отрицательное, функция возвращает значение минимального аргумента, иначе – максимального.

```
// Файл TemplateExtremum.cpp

template <class T>                // T - некий тип данных
T Extremum(T& x, T& y, int extr)  // Определение максимума или минимума
{
    if( extr < 0 )                // Если аргумент extr отрицательный,
        return (x < y ? x : y);   // возвращаем минимум
    else                          // Иначе возвращаем значение
        return (x > y ? x : y);   // максимального аргумента
}

#include <iostream>
#include <locale>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    int ia = 1, ib = 2;
    cout << "min(" << ia << ", " << ib << ") = "
        << Extremum(ia, ib, -1) << endl;
    double da = 3.14, db = 2.72;
    cout << "max(" << da << ", " << db << ") = "
        << Extremum(da, db,  1) << endl;
    string sa = "12345", sb = "abcd";
    cout << "min(" << sa << ", " << sb << ") = "
        << Extremum(sa, sb, -1) << endl;
    cin.get();
    return 0;
}
```

Программа выводит:

```
min(1, 2) = 1
```

```
max(3.14, 2.72) = 3.14
min(12345, abcd) = 12345
```

17.2. Классы и шаблоны

Шаблон семейства классов определяется инструкцией:

```
template <СПИСОК_ПАРАМЕТРОВ_ШАБЛОНА>
ОПРЕДЕЛЕНИЕ_КЛАССА
```

Шаблоны семейства классов определяют способ построения отдельных классов. В следующей программе приведен пример создания шаблона класса для векторов, которые могут состоять из элементов разных типов.

Программа 17.3. Шаблон класса векторов

В данной программе определен шаблон класса `Vect` для моделирования векторов, элементы которых могут иметь любой тип.

```
// Файл TemplateClassVect.cpp

template<class T>                  //T-параметр шаблона

class Vect{                        // Шаблон класса векторов
    T* ar;                         // Одномерный массив из элементов типа T
    int sz;                        // Размер вектора
public:
    Vect(int);                     // Объявление конструктора
    ~Vect()                        // Деструктор
    { delete[]ar; }                // освобождает память
    T&operator[](int i)            // Доступ к элементу вектора
    { return ar[i]; }
};

// Внешнее определение конструктора. Повторяется объявление параметра
// шаблона с помощью template

template<class T>
Vect<T>::Vect(int size)  // Имя шаблона класса используется с параметром T
{
    sz = size;
    ar = new T[sz];
}

// Использование шаблона класса векторов Vect

#include<iostream>
using namespace std;

void main()
{
    Vect<int>  X(5);                  // Вектор из 5 int
    Vect<char> S(5);                  // Вектор  из 5 char
```

```
    for(int i = 0; i < 5; i++){       // Заполнение
        X[i] = i;                     // векторов
        S[i] = 'A' + i;
    }
    for(int i = 0; i < 5; i++)        // Вывод векторов
        cout << "   " << X[i] << ' ' << S[i];
    cin.get();
}
```

Программа выводит:

```
0 A    1 B    2 C    3 D    4 E
```

Строка функции main()

```
Vect<int>  X(5);
```

приводит к тому, что на основе шаблона класса `Vect` автоматически создается класс содержащий массив `V` из элементов целого типа. Аналогично, строка

```
Vect<char> S(5);
```

приводит к генерации класса с элементами типа `char`. Без использования шаблонов пришлось бы писать отдельно эти оба класса.

Значением параметра шаблона может быть стандартный тип (`int`, `char`, `double`,...) или тип, определённый пользователем.

В списке параметров шаблона могут присутствовать параметры с фиксированным типом, причем параметр шаблона может иметь значение по умолчанию. Это иллюстрируется в программе, которая развивает шаблон классов `Vect`.

Программа 17.4. Развитие шаблона класса векторов

Для шаблона векторов добавлен параметр целого типа, задающий размер вектора, причем указано значение по умолчанию 12. Шаблон класса динамических массивов похож на шаблон классов векторов. Добавлена возможность устанавливать размер массива по умолчанию и функция нахождения максимального элемента массива. Объявление шаблона можно разместить в отдельном заголовочном файле. Именно в заголовочном, потому что объявление шаблона не компилируется, а используется как образец, по которому автоматически создаются конкретные классы. Шаблон используется, когда в коде возникает необходимость создать реализацию шаблона для конкретного типа данных.

```
// Файл TemplatClassVect.h

#ifndef TemplateClassVect
#define TemplateClassVect
```

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

template<class T, int sz = 12>    //T - тип элементов вектора
class Vect{                       // sz - размер создаваемого вектора
    T* ar;                        // Одномерний массив из элементов типа T
    int size;                     // Размер вектора
public:
    Vect();                       // Конструктор
    ~Vect()                       // Деструктор
    { delete[]ar; }               // освобождает память
    T&operator[](int i)           // Оператор доступа
    { return ar[i]; }             // к элементу вектора
    T GetMax();                   // Значение максимального элемента
    int GetSize()                 // Размер вектора
    { return sz; }
};

// Определение конструктора

template<class T, int sz>
Vect<T, sz>::Vect()
{
    size = sz;                    // Запоминаем размер вектора
    ar = new T[size];             // Создание массива для элементов вектора
    for(int i = 0; i < sz; i++)   // Заполнение массива случайными числами
        ar[i] = rand() / T(100); // T(100) - преобразование целого 100
}                                 // к типу T

// Определение GetMax() - функции-члена шаблона классов

template <class T, int sz>
T Vect<T, sz>::GetMax()           // Возвращает значение
{                                 // максимального элемента
    T max = ar[0];
    for(int i = 1; i < size; i++)
        if(ar[i] > max)
            max = ar[i];
    return max;
}

// Шаблон функции вывода вектора

template<class T, int sz>
ostream& operator<<(ostream& os, Vect<T, sz>& v)
{
    for(int i = 0; i < v.GetSize(); i++)
        os << v[i] << ' ';
    return os;
}

#endif
```

Теперь приведем программу, в которой используется наблон класса векторов.

```
// Файл UseTemplateClassVect.cpp

#include "TemplateClassVect.h"

int main()
{
    setlocale(LC_ALL, "Russian");
    srand(unsigned(time(NULL))); // Инициализация датчика случайных чисел
    Vect<double, 7> dv;          // Вектор dv из 7 double
    Vect<int> iv;                // вектор iv из 12 int
    cout << "Вектор dv:\n";
    cout << dv << endl;
    cout << "Максимальное значение dv = " << dv.GetMax() << endl;
    cout << "Вектор iv:\n";
    cout << iv << endl;
    cout << "Максимальное значение iv = " << iv.GetMax() << endl;
    cin.get();
    return 0;
}
```

Программа выводит:

```
Вектор dv:
72.84 176.07 52.53 277.35 276.93 3.78 171.73
Максимальное значение dv = 277.35
Вектор iv:
312 86 180 160 130 208 37 130 0 160 208 118
Максимальное значение iv = 312
```

В главной функции при выполнении инструкции

```
Vect<double, 7> dv;
```

создается по шаблону класс векторов из элементов типа `double` размером 7 элементов, затем с использованием этого класса создается вектор `dv`. Аналогично при выполнении инструкции

```
Vect<int> iv;
```

создается класс векторов из элементов типа `int` размером 12, который задан по умолчанию, и объект этого класса `iv`.

При создании `dv` и `iv` вызываются конструкторы соответствующих классов, которые заполняют вектора случайными числами соответствующего типа. Для этого случайные целые числа, генерируемые функцией `int rand(void)`, преобразуются к типу `T` инструкцией:

```
ar[i] = rand() / T(100);
```

Здесь целое число 100 приводится к типу `T`, затем целое число делится на число типа `T`, в результате получается значение типа `T`.

17.3. Обработка исключений

Исключение – это особая ситуация, возникающая в ходе работы программы. Встроенными особыми ситуациями являются: «деление на нуль», «конец файла», «переполнение при вычислении». В языке C++ любое состояние, достигнутое в процессе выполнения программы, можно заранее определить как особую ситуацию (исключение), и предусмотреть действия, которые нужно выполнить при её возникновении.

Для реализации механизма обработки исключений в языке C++ есть три ключевых слова:

`try` (пробовать, пытаться, испытывать),
`catch` (ловить, обнаруживать),
`throw` (бросать, посылать).

Эти ключевые слова позволяют выделить контролируемый блок, зафиксировать исключение и назначить обработчик исключения. Схема обработки исключений имеет вид:

```
try{                        // Контролируемый блок
    ОПЕРАТОРЫ               // Обычные операторы
    throw ВЫР;              // Передача управления обработчику исключений
    ОПЕРАТОРЫ
}

catch(ТИП_ИСКЛЮЧЕНИЯ ИМЯ){      // Обработчик исключений
  ОПЕРАТОРЫ
}
```

Программист должен предусмотреть передачу управления оператору `throw` при возникновении исключения. Оператор `throw` прерывает естественный ход работы программы и передает управление в тот блок `catch`, у которого `ТИП_ИСКЛЮЧЕНИЯ` соответствует типу `ВЫР`, при этом объект `ИМЯ` инициализируется значением `ВЫР`.

В приводимой ниже программе демонстрируется обработка исключений в задаче о нахождении наибольшего общего делителя двух целых чисел x и y.

Программа 17.5. Расчет наибольшего общего делителя

Справедливы следующие свойства наибольшего общего делителя двух чисел:

$Nod(x, y) = x$, если $x = y$;
$Nod(x, y) = Nod(y, x - y)$, если $x > y$,
$Nod(x, y) = Nod(y, x)$;

если $x <= 0$, или $y <= 0$, то *Nod* не определён.

Рассмотрим пример: $x = 245$, $y = 105$. Используя приведенные формулы, получим:

$$Nod(245, 105) = Nod(105, 140) = Nod(140, 105) = Nod(105, 35) = Nod(35, 70) = Nod(70,35) = Nod(35, 35) = 35.$$

Исключительными ситуациями при расчете НОД являются равенство нулю или отрицательное значение одного из чисел.

Для передачи информации из точки, где может возникнуть исключение, в обработчик исключений, определим специальные классы `ExceptZero` и `ExceptNegative` членами которого будут две целые переменные `m, n`, хранящие значение x и y, и строка `mess` с сообщением о характере особой ситуации.

```
#include <iostream>

using namespace std;

struct ExceptZero{              // Класс для информации об исключении
  int m, n;                     // Числа
  char* mess;                   // Описание исключения
  ExceptZero(int a, int b)      // Конструктор
  { m = a; n = b; mess = "Zero: "; }
};

struct ExceptNegative{      // Еще класс для информации об исключении
  int m, n;                     // Числа
  char* mess;                   // Описание исключения
  ExceptNegative(int a, int b)  // Конструктор
  { m = a; n = b; mess = "Negative: "; }
};

int Nod(int x, int y)           // Вычисляет наибольший общий
{                               // делитель чисел x и y
    if(x == 0 || y == 0)
        throw ExceptZero(x,y);
    if(x < 0)
        throw ExceptNegative(x, y);
    if(y < 0)
        throw ExceptNegative(x, y);
    if(x == y)
        return x;
    if(x < y)
        return Nod(y, x);
    return Nod(y, x - y);
}

void main()
{
    try{
        cout << "\nNod(245, 105) = " << Nod(245, 105) << '\n';
```

```
        cout << "\nNod(0, 7) = "     << Nod(0, 7)   << '\n';
        cout << "\nNod(-12, 8) = "   << Nod(-12, 8) << '\n';
    }
    catch(ExceptZero z){    // Перехват исключений типа ExceptZero
                            // В z содержится информация об исключении
        cerr << z.mess << " x = " << z.m << ", y = " << z.n <<'\n';
    }
    catch(ExceptNegative ng){  // Перехват исключений типа ExceptZero
                             // В ng содержится информация об исключении
        cerr << ng.mess << " x = " << ng.m << ", y = " << ng.n <<'\n';
    }
    cin.get();
}
```

В функции Nod() после throw стоит вызов конструктора класса ExceptZero, когда один из аргументов равен нулю, и конструктора класса ExceptNegative, когда один из аргументов отрицателен. Конструкторы создают безымянные объекты, инициализированный значениями аргументов Nod(). Оператор throw обеспечивает передачу этих объектов в соответствующий блок catch, где они используются для инициализации либо параметра z, либо параметра ng. Тем самым информация об исключении становится доступной в обработчике исключений. Обработка исключения состоит в выводе в поток для ошибок cerr сведений о возникшей проблеме, передаваемых через объект z и ng.

Программа выдает:

```
Nod(245, 105) = 35
Zero:  x = 0, y = 7
```

Здесь исключение возникает при вычислении Nod(0, 7). Оператор throw ExceptZero(x,y) прерывает выполнение функции Nod() и передает управление за пределы блока try. При этом отыскивается тот из нескольких возможных блоков catch, у которого тип объекта-исключения соответствует типу исключения, сгенерированному в операторе throw. В данном случае управление передается в блок catch(ExceptZero z){…}.

Закомментируем строку:

```
//cout << "\nNod(0, 7) = "     << Nod(0, 7)   << '\n';
```

Исключение возникнет при выполнении следующей строки и программы выведет:

```
Nod(66, 44) = 22
Negative:  x = -12, y = 8
```

Оператор catch, у которого вместо типа исключения стоит многоточие:

```
catch(…){       // Обработчик любых исключений
  ИНСТРУКЦИИ
}
```

перехватывает исключения любых типов. Он срабатывает, если для исключения, возникшего в предшествующем блоке try, не найдено обработчика подходящего типа.

17.4. Стандартная библиотека шаблонов

Для обработки данных используются различные *структуры данных*: массивы, списки, стеки, очереди, множества и т.д. Такая структура данных, как массив, встроена в C++, так же, как и в другие языки программирования. Стандартом C++ предусмотрена стандартная библиотека шаблонов STL (Standard Template Library), предназначенная для создания и обработки различных структур данных. В STL содержатся три основные сущности: *контейнеры*, *алгоритмы* и *итераторы*.

Контейнер организует хранение данных. Контейнеры STL реализованы в виде шаблонов классов, поэтому обеспечивают хранение как величин базовых типов `int`, `float`,…, так и объектов пользовательских классов. Необходимость в наличии специальных контейнеров вызвана тем, что обычный массив бывает неэффективен, например, часто трудно предугадать требуемый размер массива.

Контейнеры бывают *последовательные* и *ассоциативные*. Последовательными контейнерами являются, например, векторы (`vector`), списки (`list`). Ассоциативные контейнеры – это, например, множества (`set`), отображения или ассоциативные массивы (`map`). Здесь `vector`, `list`, `set`, `map` – имена шаблонных классов, для использования которых в программу надо включить заголовочные файлы с такими же названиями.

Контейнеры имеют методы, позволяющие работать с ними. Перечислим некоторые:

`size()` – возвращает текущее число элементов в контейнере;
`empty()` – возвращает `true`, если контейнер пуст;
`max_size()` – размер самого большого возможного контейнера;
`resize()` – изменяет размер контейнера (только вектора, списка).

Алгоритмы – это независимые шаблонные функции, не являющиеся членами контейнерных классов, поэтому их можно применять как к классам-контейнерам, так и к обычным массивам. Алгоритмы

становятся доступными после включения в программу заголовочного файла `algorithm`. Вот некоторые алгоритмы:

`find()` – находит первое вхождение значения в последовательность;
`count()` – подсчитывает число вхождений данного значения в последовательность;
`equal()` – возвращает `true`, если элементы двух последовательностей попарно равны;
`search()` – находит первое вхождение подпоследовательности в последовательность;
`sort()` – сортирует последовательность в указанном порядке.

Итераторы – это обобщение понятия указателей. Итераторы ссылаются на элементы контейнеров и обеспечивают к ним доступ. К итераторам можно применять оператор ++, после выполнения которого итератор ссылается на следующий элемент контейнера. Переход от элемента к элементу называется *итерацией*. Значение элемента по итератору получается оператором *. В STL итераторы представляет собой объекты класса `iterator`.

Для разных типов контейнеров используются свои итераторы. Существует три типа итераторов.

Прямой итератор может проходить по контейнеру только в прямом направлении, к нему применим оператор ++.

Двунаправленный итератор может передвигаться по контейнеру в обоих направлениях и реагирует на операторы ++ и --.

Итератор *со случайным доступом* может двигаться в обоих направлениях и перескакивать на любой элемент контейнера.

Контейнеры имеют методы, возвращающие итераторы. Приведем для примера два таких метода:

`begin()` – возвращает итератор на начало контейнера;
`end()` – возвращает итератор на элемент, расположенный после последнего элемента контейнера.

В следующей программе приводится пример использования небольшой части возможностей стандартной библиотеки шаблонов STL.

Программа 17.6. Вектора и алгоритмы

```
//Работа с типом vector и алгоритмами
#include <iostream>
```

```
#include <vector>
#include <algorithm>
#include <locale>
using namespace std;

// Шаблон функции-оператора для вывода вектора в поток
template <class T>
ostream& operator<<(ostream& os, vector<T>v)
{
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << ' ';
    return os;
}

void main()
{
    double Max = 1000.0;
    srand(time(0));              // Инициализация генератора случайных чисел
    setlocale(LC_ALL, "Russian");
    int N;                       // Размер векторов
    cout << "Введите размер вектора ";
    cin >> N;

    // Создание векторов заданного размера
    vector <int> vct(N);     // vct - вектор из N элементов типа int
    vector<double> dvct(N);  // dvct - вектор из N элементов типа double

// Заполнение векторов случайными числами
    for(int i = 0; i < vct.size(); i++){
        dvct[i] = rand() / Max;          // Заполнение вектора из double
        vct[i]  = dvct[i];               // Копирование в вектор из int
    }

    cout << "Вектор из double:\n";
    cout << dvct << endl;

    cout << "Исходный вектор из int:\n";
    cout << vct << endl;
    vector <int> svct(N);            // Еще один вектор из int
    svct = vct;                      // Присваивание векторов
    sort(svct.begin(), svct.end());  // Сортировка svct от начала до конца

    cout << "Упорядоченный вектор из int :\n";
    cout << svct << endl;
    vct.resize(N * 2);               // Удвоение размера вектора
    for(int i = 0; i< N; i++)        // Добавление в расширенный вектор
        vct[N + i] = svct[i];

    cout << "Объединенный вектор из int:\n";
    cout << vct << endl;
    cin.get();
}
```

Программа выводит:

```
Введите размер вектора 6
```

```
Вектор из double:
28.872 11.551 17.555 5.56 9.464 21.405
Исходный вектор из int:
28 11 17 5 9 21
Упорядоченный вектор из int :
5 9 11 17 21 28
Объединенный вектор из int:
28 11 17 5 9 21 5 9 11 17 21 28
```

Для вывода векторов из элементов разного типа написан шаблон функции вывода `ostream& operator<<(ostream& os, vector<T>v)`.

В программе создается вектор `dvct` из элементов типа `double` и два вектора `vct` и `svct` из элементов типа `int`. Вектор `svct` сортируется стандартным алгоритмом `sort`. Размер вектора `vct` удваивается и в добавленную память копируются элементы `svct`.

Использование шаблонов и алгоритмом позволяет существенно ускорить разработку программ.