**Лабораторная работа №11: Шаблоны (параметризованные типы)**

Цель работы: изучить представление и правила работы с шаблонами в С++. Теоретические сведения

Использование шаблонов призвано, облегчить процесс написания полноценных программ, где под понятием "написание" подразумевается не только процедура первоначального написания кода программы, но и последующий за этим долгий процесс отладки, модификации и сопровождения, созданного вами программного продукта. Чем же шаблоны могут упростить процесс написания программ?

Ранее мы дублировали и размножали части программ, используя простое, но эффективное средство - текстовый редактор. Сегодня С++ предлагает нам более совершенный способ дублирования, и имя ему - "шаблоны".

Наиболее распространенным поводом для дублирования фрагментов программ является необходимость реализовать некий новый объем кода, аналогичный уже написанному, но изменив типы данных (например, целые на целые длинные). Чаще всего для подобной операции с помощью программы-редактора повторяли текст программы еще раз и меняли типы данных. Программируя на С++, вы могли бы воспользоваться средствами переопределения (перегрузки) и дать обеим функциям одно и тоже имя. Переопределение делает текст программы более наглядным, но не избавляет нас от необходимости повторять один и тот же алгоритм в нескольких местах.

Механизм шаблонов предлагает совершенное решение, позволяющее отделить общий алгоритм от его реализации применительно к конкретным типам данных. Вы можете составить текст подпрограммы сейчас, а используемые типы уточнять позднее. Это возможно, так как используемый тип данных является в этом случае параметром. Шаблоны сочетают в себе преимущества однократной подготовки фрагментов программы (аналогично макрокомандам) и контроль типов, присущий переопределяемым функциям.

В языке С++ имеются два типа шаблонов - шаблоны функций и шаблоны классов.

**Шаблоны функций**

Объявление шаблона функции начинается с заголовка, состоящего из ключевого слова template, за которым следует список параметров шаблона.

// Описание шаблона функции

template <class X>

X min (X a, X b)

{ return a<b ? a : b;

}

Ключевое слово **class** в описании шаблона означает тип, идентификатор в списке параметров **шаблона X** означает имя любого типа. В описании заголовка функции этот же идентификатор означает тип возвращаемого функцией значения и типы параметров функции.

// Использование шаблона функции

int m = min (1, 2);

...

//Экземпляр шаблона функции порождается (генерируется) компилятором

int min (int a, int b)

{ return a<b ? a : b;

}

В списке параметров шаблона слово **class** может также относится к обычному типу данных. *Таким образом, список параметров шаблона просто означает, что Т представляет собой тип, который будет задан позднее*. Так как **Т** является параметром, обозначающим тип, шаблоны иногда называют параметризованными типами. Приведем описание шаблона функции

template <class T>

T toPower (T base, int exponent)

{ T result = base;

if (exponent==0) return (T)1;

if (exponent<0) return (T)0;

while (--exponent) result \*= base;

return result;

}

Переменная **result** имеет **тип Т**, так что, когда передаваемое в программу значение есть 1 или 0, то оно сначала приводится к типу Т, чтобы соответствовать объявлению шаблона функции. Типовой аргумент шаблона функции определяется согласно типам данных, используемых в вызове этой функции: int i = toPower (10, 3); long l = toPower (1000L, 4); double d = toPower (1e5, 5); В первом примере Т становится типом int, во втором - long. Наконец, в третьем примере Т становится типом double. Следующий пример приведет к ошибке компиляции, так как в нем принимающая переменная и возвращаемое значение имеют разные типы: int i = toPower (1000L, 4);

**Требования к фактическим параметрам шаблона**

Шаблон функции toPower() может быть использован почти для любого типа данных. Предостережение "почти" проистекает из характера операций, выполняемых над параметром base и переменной result в теле функции toPower(). Какой бы тип мы ни использовали в функции toPower(), эти операции для нее должны быть определены. В противном случае компилятор не будет знать, что ему делать. Вот список действий, выполняемых в функции toPower() с переменными base и result:

1. T result = base;
2. return (T)1;
3. return (T)0;
4. result \*= base;
5. return result;

Все эти действия определены для встроенных типов. Однако если вы создадите функцию toPower() для какого-либо классового типа, то в этом случае такой класс должен будет включать общедоступные принадлежащие функции, которые обеспечивают следующие возможности:

* действие 1 инициализирует объект типа Т таким образом, что класс Т должен содержать конструктор копирования,
* действия 2 и 3 преобразуют значения типа int в объект типа Т, поэтому класс Т должен содержать конструктор с параметром типа int, поскольку именно таким способом в классах реализуется преобразование к классовым типам,
* действие 4 использует операцию \*= над типом Т, поэтому класс должен содержать собственную функцию-operator \*=().
* действие 5 предполагает, что в типе T предусмотрена возможность построения безопасной копии возвращаемого объекта (см. конструктор копирования).

Схема такого класса выглядит следующим образом:

class T

{ public:

T (const T &base); // конструктор копирования

T (int i); //приведение int к Т

operator \*= (T base);

// ... прочие методы

}

Используя классы в шаблонах функций, убедитесь в том, что вы знаете, какие действия с ними выполняются в шаблоне функции, и определены ли для класса эти действия. Если вы не снабдили класс необходимыми функциями, возникнут различные невразумительные сообщения об ошибках.

**Отождествление типов аргументов**

Так как компилятор генерирует экземпляры шаблонов функций согласно типам, заданным при их вызовах, то критическим моментом является передача корректных типов, особенно если шаблон функции имеет два или более параметров. Хорошим примером является классическая функция max():

template <class T>

T max (T a, T b)

{ return a > b ? a : b;

}

Функция max() будет работать правильно, если оба ее аргумента имеют один и тот же тип: int i = max (1, 2); double d = max (1.2, 3.4); Однако, если аргументы различных типов, то вызов max() приведет к ошибке, так как компилятор не сможет понять, что ему делать. Один из возможных способов для разрешения неоднозначности состоит в использовании приведения типов, чтобы прояснить наши намерения: int i = max ((int)'a', 100); Вторая возможность - это явно объявить версию экземпляра шаблона функции перед ее вызовом: int max (int, int); int j = max ('a', 100); Третий способ решить проблему состоит в создании шаблона функций, который имеет параметры различных типов.

template <class T1, class T2>

T1 max (T1 a, T2 b)

{ return a > (T1)b ? a : (T1)b;

}

Использование этой новой версии max() не приведет к неоднозначности в случае использования двух различных типов. Например, если написать max ('a', 100); то компилятор будет использовать два заданных (посредством аргументов типа) и построит версию функции max() с заголовком char max (char, int); Далее компилятор перед выполнением сравнение приведет тип второго аргумента к типу первого аргумента. Такой способ допустим, однако использование двух типовых параметров в шаблоне функции, которая должна была бы работать только с одним типом, часто лишь затрудняет жизнь. Довольно тяжело помнить, что max ('a', 100) дает значение типа char, в то время как max (100, 'a') передает в вызывающую программу int.

**Шаблоны классов**

Вы можете создавать шаблоны и для классов, что позволяет столь же красиво работать с любыми типами данных. Классическим примером являются контейнерные классы (т.е. классы, содержащие типы), например множества. Используя шаблон, можно создавать родовой класс для множеств, после чего порождать конкретные множества - цветов, строк и т.д. Сначала рассмотрим пример. Рассматриваемый далее пример - класс, который хранит пару значений. Функции-элементы этого класса возвращают минимальное и максимальное значения, а также позволяют определить, являются ли два значения одинаковыми. Поскольку перед нами шаблон класса, то тип значения может быть почти любым.

template <class T>

class Pair

{ T a, b;

public:

Pair (T t1, T t2);

T Max();

T Min ();

int isEqual ();

};

Пока все выглядит также отлично, как и для шаблонов функций. Единственная разница состоит в том, что вместо описания функции используется объявление класса. Шаблоны классов становятся все более сложными, когда вы описываете принадлежащие функции класса. Вот, например, описание принадлежащей функции Min() класса Pair:

template <class T>

T Pair <T>::Min()

{ return a < b ? a : b;

}

Чтобы понять эту запись, давайте вернемся немного назад. Если бы Pair был обычным классом (а не шаблоном класса) и T был бы некоторым конкретным типом, то функция Min класса Pair была бы описана таким образом:

T Pair::Min()

{ return a < b ? a : b;

}

Для случая шаблонной версии нам необходимо, во-первых, добавить заголовок шаблона template Затем нужно дать имя классу. Помните, что на самом деле мы описываем множество классов - семейство Pair. Повторяя синтаксис префикса (заголовка) шаблона, экземпляр класса Pair для целых типов, можно назвать Pair, экземпляр для типа double - Pair, для типа Vector - Pair. Однако в описании принадлежащей классу функции нам необходимо использовать имя класса Pair. Это имеет смысл, так как заголовок шаблона говорит, что Т означает имя любого типа. Приведем текст остальных методов класса Pair. Описания методов помещаются в заголовочный файл, так как они должна быть видимы везде, где используется класс Pair.

// конструктор

template <class T>

Pair <T>::Pair (T t1, T t2) : a(t1), b(t2)

{}

// метод Max

template <class T>

T Pair <T>::Max()

{ return a>b ? a : b;

}

// метод isEqual

template <class T>

int Pair <T>::isEqual()

{ if (a==b) return 1;

return 0;

}

Ранее уже отмечалось, что шаблоны функций могут работать только для тех (встроенных) типов данных или классов, которые поддерживают необходимые операции. То же самое справедливо и для шаблонов классов. Чтобы создать экземпляр класса Pair для некоторого классового типа, например для класса X, этот класс должен содержать следующие общедоступные функции

X (X &); // конструктор копирования

int operator == (X)

int operator < (X);

Три указанные функции необходимы, так как они реализуют операции, выполняемые над объектами типа T в метода X шаблона класса Pair. Если вы собираетесь использовать некоторый шаблон класса, как узнать какие операции требуются? Если шаблон класса снабжен документацией, то эти требования должны быть в ней указаны. В противном случае придется читать первичную документацию - исходный текст шаблона. При этом учитывайте, что встроенные типы данных поддерживают все стандартные операции.

**Шаблоны классов: не только для типов**

Параметризовать некоторый класс так, чтобы он работал для любого типа данных - это только половина того, что шаблоны обеспечивают для классов. Другой аспект состоит в том, чтобы дать возможность задания числовых параметров. Это позволяет Вам, например, создавать объекты типов "Вектор из 20 целых", "Вектор из 1000 целых" или "Вектор из 10 переменных типа double". Основная идея проста, хотя используемый синтаксис может показаться сложным. Давайте в качестве примера рассмотрим некоторый обобщенный класс Vector. Как и класс Pair, класс Vector содержит функции Min(), Max(), isEqual(). Однако в нем может быть любое количество участников, а не два. В классе Pair число участников фиксировано и задаются они в качестве аргументов конструктора. В шаблоне Vector вместо этого используется второй параметр заголовка шаблона:

template <class T, int n> class Vector

{ public:

Vector();

~Vector() {delete[] coord;}

void newCoord (T x);

T Max ();

T Min();

int isEqual();

private:

T \*coord;

int current;

};

Значение n, заданное в заголовке шаблона не используется в описании класса, но применяется в описании его методов. Конструктор Vector, использующий значение n для задания размера массива, выглядит так:

// конструктор

template <class T, int n>

Vector <T, n>::Vector():

{ coord = new T[n];

current = 0;

}

// метод Max

template <class T, int n>

T Vector <T, n>::Max():

{ T result (coord[0]); // \*

for (int i=0; i<n; i++)

if (result < coord[i]) // \*\*

result = coord[i]; // \*\*\*

}

В конструкторе задается список инициализаций, устанавливающих начальные значения для двух элементов класса. Элемент coord инициализируется адресом динамически размещенного массива размером n и состоящего из элементов типа Т, а элемент current инициализируется значением 0.

Опять, заметим, что в качестве Т может выступать почти любой тип. Однако и в этом случае успешная реализация возможна лишь при определенных условиях - для объектов, чей тип передается в шаблон в качестве параметра, должны быть определены следующие операции

1. конструктор копирования (\*),
2. оператор < (\*\*), и > для метода Max(),
3. оператор = (\*\*\*).

Имеется несколько вариантов использования шаблонов с параметрами-значениями для динамического размещения массивов различных размеров. Например, можно передать размер массива конструктору. Указание размеров объекта во время конструирования или путем обращения к некоторому методу действительно обеспечивает задание размера, однако такой способ не позволяет создать отдельный тип для каждого отдельного размера. Подход с использованием шаблона гарантирует, что размер становится частью типа. Так, Vector с пятью элементами типа double является типом, отличным от Vector с четырьмя элементами типа double.

Хотите ли Вы, чтобы различные размеры были различными типами, зависит от ваших нужд. Если сравнение двух векторов с четырьмя и пятью координатами не имеет особого смысла, то было бы неплохо сделать их различными типами. Вместе с тем, в случае классов для матриц, Вы, возможно, не захотите иметь особый тип для каждого размера матриц, так как, например, умножение, работает с матрицами различных размеров. Если Вы столкнетесь с подобной проблемой, то Вам потребуются только разумные проверки времени выполнения, а не контроль типов при компиляции.

Хотя числовые параметры шаблонов часто используются для задания размеров различных элементов, как это было показано для класса Vector, этим их применение не ограничивается. Например, с помощью числовых параметров можно задавать диапазоны значений графических координат в графическом классе.

**Контрольные вопросы**

1. Что такое шаблоны и с какой целью они используются?
2. Какого типа шаблоны используются в программах?
3. Как оформляются шаблоны функций?
4. Какие требования предъявляются к фактическим параметрам шаблонов?
5. Какие преимущества программы обеспечиваются при использовании шаблонов классов?

**Варианты заданий**

1. Опишите параметризованный класс Vector - односвязный список элементов (параметр – тип).
2. Опишите параметризованный класс List - двусвязный список элементов (параметр – тип).
3. Опишите параметризованный класс геометрическая фигура (параметр – тип данных будет различный для стороны).
4. Опишите параметризованную функцию сортировки вставкой.
5. Опишите параметризованную функцию сортировки выборкой.
6. Опишите параметризованную функцию сортировки пузырьком.
7. Опишите параметризованную функцию нахождения элемента в неупорядоченном массиве.
8. Опишите параметризованную функцию нахождения элемента в упорядоченном массиве.
9. Опишите параметризованную функцию замены одного элемента массива на другой.
10. Опишите параметризованную функцию инверсии массива элементов.
11. Опишите параметризованную функцию вычисления среднего арифметического значения массива элементов.
12. Опишите параметризованный класс «массив» для вычисления минимального элемента в массиве разного типа.

\*\*\*Сортировка методом пузырька:  
#include <iostream>

#include <iomanip>

#include <ctime>

using namespace std;

void bubbleSort(int \*, int); // прототип функции сортировки пузырьком

int main(int argc, char\* argv[])

{

    srand(time(NULL));

    setlocale(LC\_ALL, "rus");

    cout << "Введите размер массива: ";

    int size\_array; // длинна массива

    cin >> size\_array;

    int \*sorted\_array = new int [size\_array]; // одномерный динамический массив

    for (int counter = 0; counter < size\_array; counter++)

    {

        sorted\_array[counter] = rand() % 100; // заполняем массив случайными числами

        cout << setw(2) << sorted\_array[counter] << "  "; // вывод массива на экран

    }

    cout << "\n\n";

    bubbleSort(sorted\_array, size\_array); // вызов функции сортировки пузырьком

    for (int counter = 0; counter < size\_array; counter++)

    {

        cout << setw(2) << sorted\_array[counter] << "  "; // печать отсортированного массива

    }

    cout << "\n";

    system("pause");

    return 0;

}

void bubbleSort(int\* arrayPtr, int length\_array) // сортировка пузырьком

{

 int temp = 0; // временная переменная для хранения элемента массива

 bool exit = false; // болевая переменная для выхода из цикла, если массив отсортирован

 while (!exit) // пока массив не отсортирован

 {

  exit = true;

  for (int int\_counter = 0; int\_counter < (length\_array - 1); int\_counter++) // внутренний цикл

    //сортировка пузырьком по возрастанию - знак >

    //сортировка пузырьком по убыванию - знак <

    if (arrayPtr[int\_counter] > arrayPtr[int\_counter + 1]) // сравниваем два соседних элемента

    {

     // выполняем перестановку элементов массива

     temp = arrayPtr[int\_counter];

     arrayPtr[int\_counter] = arrayPtr[int\_counter + 1];

     arrayPtr[int\_counter + 1] = temp;

     exit = false; // на очередной итерации была произведена перестановка элементов

    }

 }

}

\*\*\*Сортировка выбором

#include <iostream>

#include <ctime>

#include <iomanip>

using namespace std;

void choicesSort(int\*, int); // прототип функции сортировки выбором

int main(int argc, char\* argv[])

{

    srand(time(NULL));

    setlocale(LC\_ALL, "rus");

    cout << "Введите размер массива: ";

    int size\_array; // длинна массива

    cin >> size\_array;

    int \*sorted\_array = new int [size\_array]; // одномерный динамический массив

    for (int counter = 0; counter < size\_array; counter++)

    {

        sorted\_array[counter] = rand() % 100; // заполняем массив случайными числами

        cout << setw(2) << sorted\_array[counter] << "  "; // вывод массива на экран

    }

    cout << "\n\n";

    choicesSort(sorted\_array, size\_array); // вызов функции сортировки выбором

    for (int counter = 0; counter < size\_array; counter++)

    {

        cout << setw(2) << sorted\_array[counter] << "  "; // печать отсортированного массива

    }

    cout << "\n";

    delete [] sorted\_array; // высвобождаем память

    system("pause");

    return 0;

}

void choicesSort(int\* arrayPtr, int length\_array) // сортировка выбором

{

    for (int repeat\_counter = 0; repeat\_counter < length\_array; repeat\_counter++)

    {

        int temp = arrayPtr[0]; // временная переменная для хранения значения перестановки

        for (int element\_counter = repeat\_counter + 1; element\_counter < length\_array; element\_counter++)

        {

            if (arrayPtr[repeat\_counter] > arrayPtr[element\_counter])

            {

                temp = arrayPtr[repeat\_counter];

                arrayPtr[repeat\_counter] = arrayPtr[element\_counter];

                arrayPtr[element\_counter] = temp;

            }

        }

    }

}

\*\*\*Сортировка вставкой

#include <iostream>

#include <ctime>

#include <iomanip>

using namespace std;

void insertionSort(int \*, int); // прототип функции сортировки вставками

int main(int argc, char\* argv[])

{

    srand(time(NULL));

    setlocale(LC\_ALL, "rus");

    cout << "Введите размер массива: ";

    int size\_array; // длинна массива

    cin >> size\_array;

    int \*sorted\_array = new int [size\_array]; // одномерный динамический массив

    for (int counter = 0; counter < size\_array; counter++)

    {

        sorted\_array[counter] = rand() % 100; // заполняем массив случайными числами

        cout << setw(2) << sorted\_array[counter] << "  "; // вывод массива на экран

    }

    cout << "n";

    insertionSort(sorted\_array, size\_array); // вызов функции сортировки вставками

    for (int counter = 0; counter < size\_array; counter++)

    {

        cout << setw(2) << sorted\_array[counter] << "  "; // печать отсортированного массива

    }

    cout << "n";

    delete [] sorted\_array; // высвобождаем память

    system("pause");

    return 0;

}

void insertionSort(int \*arrayPtr, int length) // сортировка вставками

{

    int temp, // временная переменная для хранения значения элемента сортируемого массива

        item; // индекс предыдущего элемента

    for (int counter = 1; counter < length; counter++)

    {

        temp = arrayPtr[counter]; // инициализируем временную переменную текущим значением элемента массива

        item = counter-1; // запоминаем индекс предыдущего элемента массива

        while(item >= 0 && arrayPtr[item] > temp) // пока индекс не равен 0 и предыдущий элемент массива больше текущего

        {

            arrayPtr[item + 1] = arrayPtr[item]; // перестановка элементов массива

            arrayPtr[item] = temp;

            item--;

        }

    }

}