Работа №1 Программирование с шаблонами функций

Цель работы - научиться программировать и применять шаблоны функций для решения задач, передавая тип данных в функцию как параметр.

Теория и практика разработки шаблонов функций

Многие алгоритмы не зависят от типов данных, с которыми они работают, классический пример – сортировка. Передача типа данных как параметра функции называется *параметрическим полиморфизмом*. В C++ средством параметризации являются шаблоны функций и шаблоны классов.

С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя.

Для описания шаблонов используется ключевое слово **template**, вслед за которым указываются аргументы (формальные параметры шаблона), заключенные в угловые скобки. Формальные параметры шаблона перечисляются через запятую, и могут быть как именами объектов, так и параметрическими именами типов (встроенных или пользовательских). Параметр-тип описывается с помощью служебного слова **class** или служебного слова **typename**.

Синтаксис описания шаблоны функции:

```
template < typename | class  тип параметра>
   заголовок функции { /* тело функции */ },
```

где тип параметра – это идентификатор типа.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только базовым типом, но и пользовательским типом, например:

```
template <class A, class B, int i> void f( ) { ... }
```

При обращении к функции-шаблону после имени функции в угловых скобках указываются фактические параметры шаблона - имена реальных типов или значения объектов:

имя функции<фактические параметры типов> (*фактические параметры*);

Пример 1. Функция, сортирующая методом выбора массив *b* из *n* элементов любого типа (параметр *Type*), в виде шаблона описана так:

```
template <class Type>
 void sort(Type *b, int n) {
      Type temp;
      for (int i = 0; i < n - 1; i++)
            for (int j = i + 1; j < n; j++) if (b[j] < b[i])
            {
                  temp = b[i]; b[i] = b[j]; b[j] = temp;
            }
}
```

Программа-клиент, вызывающая эту функцию-шаблон, будет иметь вид:

```
int main()
{
      const int n = 7;
      int i, x[n];
      for (i = 0; i < n; i++) std::cin >> x[i];

     // Сортировка целочисленного массива
      sort<int>(x, n);

     for (i = 0; i < n; i++) std::cout << x[i] << '\t';
      std::cout << std::endl;

     double a[] = { 0.22,  117,  -0.08,  0.21, 42.5 };

     // Сортировка массива вещественных чисел
      sort<double>(a, 5);

     for (i = 0; i < n; i++) std::cout << a[i] << '\t';

}
```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется **инстанцированием** шаблона (instantiation), а результат является **порожденной функцией.** Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

☺ Использование шаблонов сокращает текст программы, но не сокращает программный код. В программе реально будет сгенерировано столько порожденных функций, сколько имеется вызовов функций с разными наборами фактических параметров

Типы параметров при вызове функции-шаблона могут быть заданы явно и неявно. При вызове с неявными параметрами компилятор будет определять тип по фактическим параметрам функции. Это

называется **выведением типов аргументов шаблона**. Выведение типов аргументов возможно при условии, что список фактических параметров вызова функции однозначно идентифицирует список параметров шаблона.

Пример 2. Рассмотрим пример неявного задания параметров шаблона при вызове:

```
template <class X, class Y, class Z> void fun(Y, Z);

void main() {
	fun<int, char*, double> ("Work_2", 7.0);  // типы заданы явно
	fun<int, char*>(" Work_2", 7.0);          // Z определяется как double
	fun<int>(" Work_2", 7.0);                 // Y как char*, а Z - как double
	fun(" Work_2", 7.0);                      // ошибка: X определить невозможно
}
```

Так как допускается параметризировать шаблоны не только именами типов, но и объектами, то аргумент *n* можно указать в виде параметра шаблона:

```
template <class Type, int n>
 void sort(Type *b);
```

Тогда вызов sort будет выглядеть соответственно:

```
sort<double, 5>(a);
```

Если шаблонный алгоритм является неудовлетворительным для конкретного типа аргументов или неприменим к ним, то можно описать обычную функцию, список типов аргументов и возвращаемого значения которой соответствуют объявлению шаблона. Такая, перегружающая шаблон, функция, называется **специализацией шаблонной функции**.

С одним и тем же именем функции можно написать несколько шаблонов и перегруженных обычных функций. Алгоритм выбора перегруженной функции с учетом шаблонов является обобщением правил выбора перегруженной функции.

1. Для каждого шаблона, подходящего по набору формальных параметров, осуществляется формирование специализации, соответствующей списку фактических параметров.

2. Если могут быть два шаблона функции и один из них более специализирован, то на следующих этапах рассматривается только он (порядок специализаций описан далее).

3. Осуществляется поиск оптимально отождествляемой функции из полученного набора функций, включая определения обычных функций, подходящие по количеству параметров. При этом если параметры некоторого шаблона функции были определены путем выведения по типам фактических параметров вызова функции, то при дальнейшем поиске

оптимально отождествляемой функции к параметрам данной специализации шаблона нельзя применять никаких описанных выше преобразований, кроме преобразований **Точного** отождествления.

4. Если обычная функция и специализация подходят одинаково хорошо, то выбирается обычная функция.

5. Так же, как и при поиске оптимально отождествляемой функции для обычных функций, если полученное множество подходящих вариантов состоит из одной функции, то вызов разрешим. Если множество пусто или содержит более одной функции, то генерируется сообщение об ошибке.

Использование шаблонов позволяет создавать универсальные алгоритмы, без привязки к конкретным типам.

При разработке унифицированных функций-шаблонов возникает проблема логических условных вычислений для проверки, поиска и др. Это проблема разрешается путем применения функций-предикатов, как аргументов функций-шаблонов. **Функция-предикат** - это функция, где возвращаемым результатом будет параметр логического типа bool.

Пример 3. Вычислить произведение положительных элементов массива заданного типа.

```
template <typename T, int size >
T ProductOfArray (T* mas, bool(*predicate) (T))
{
	T p = 1;
	for (int i = 0; i < size; i++)
		if ((*predicate)(mas[i])) p *= mas[i];
	return p;
}
bool polInt (int k) { return k > 0; }	// фактическая функция- предикат
bool polFloat(float k) { return k > 0; }

int main(int argc, char* argv[])
{
	int M[5] = { -1,2,4,-7,0 };
	float X[7] = {2.3, -5.4, 7.2, 0, -11.3, 4.1, -3.3};

	cout << ProductOfArray<int, 5> (M, polInt);	//применение предиката
	cout << ProductOfArray<float, 7>(X, polFloat);
	return 0;
}
```

Использование указателей функций как аргументов в описании параметров функции делает код очень гибким в применении. Это не обязательно предикатные функции, это могут быть любые вычисления. В Примере 3 можно оператор вычисления произведение заменить на любое другое вычисление (сумма, количество, выражение), определив этот оператор как функцию.

Методика выполнения работы

Выполнение задания работы предполагает решение следующих задач:

разработать шаблоны функций алгоритмов обработки одномерных массивов разных типов;
разработать программу - клиент, где продемонстрировать корректную работу всех функций-шаблонов на массивах разных типов: int, float, string;
создать отдельный модуль функций-шаблонов обработки одномерных массивов;
применить модуль функций-шаблонов для решения конкретной задачи обработки одномерного массива.

Пусть в задании требуется разработать модуль функций-шаблонов обработки одномерных массивов: сортировка массива по возрастанию/по убыванию, вывод элементов массива в поток вывода, выполнение различных вычислений с элементами массива, поиск индекса максимального/минимального элемента. Применить универсальные функции для решения конкретной задачи.

Разберем условие задачи и её решение.

✓ разработать шаблоны функций алгоритмов обработки одномерных массивов разных типов, функции должны быть написаны в одном стиле, так как это будет единый модуль;

Решение А. Шаблон функции сортировки разработан с учетом того, сортировать требуется и в возрастающем, и в убывающем порядке, это определит предикат. Размер массива будет задан как параметр шаблона.

```
template <class Type>
void sort(Type *mas, int size, bool(*predicate) (T,T) ) {
       Type temp;
       for (int i = 0; i < size - 1; i++)
         for (int j = i + 1; j < size; j++)
              if ((*predicate) (mas[j],  mas[i])) {
                   temp = mas[i]; mas[i] = mas[j]; mas[j] = temp; }
}
```

Решение В. Шаблон функции вывода будет давать возможность выводить массив в стандартный поток. Размер массива будет задан как параметр шаблона.

```
template <class Type>
void list(Type* mas,  int size) {
       for (int i = 0; i < size; i++)
              cout << mas[i] << '\t';
       cout << endl;
}

//специализация шаблона list для типа string
template < >
void list(string* mas, int size) {
       for (int i = 0; i < size; i++)
              cout << mas[i].c_str() << '\t';
       cout << endl;
}
```

Решение C. Шаблон функции вычисления будет давать возможность проверять элемент массива на заданное условие и вычислять результат по заданному выражению. Переменная результата *p* задана по умолчанию, но при необходимости её начальное значение можно будет передать в функцию, см. *Решение J.* Условие и выражение будут заданы указателями на функции как аргументы функции `ProductOfArray`.

```
template <class Type>
Type ProductOfArray(Type* mas, int size, bool(*predicate) (Type), Type(*fun) (Type,
Type), Type p = 0)
{
	for (int i = 0; i < size; i++)
		if ((*predicate)(mas[i])) p = fun(p, mas[i]);
	return p;
}
```

Решение D. Шаблон функции поиска индекса минимального/максимального элемента массива.

```
template <class Type>
int IndexMinMax(Type *mas, int size, bool(*predicate) (Type, Type)) {
	int Index = 0;
	for (int i = 1; i < size; i++)
		if ((*predicate) (mas[Index], mas[i]))  Index = i;
	return Index;
}
```

✓ разработать программу - клиент, где продемонстрировать корректную работу всех функций-шаблонов на массивах разных типов: int, float, string;

Решение A. Необходимо разработать набор функций-параметров и функций-предикатов для вызовов функций шаблонов. Определить ряд переменных разных типов согласно требованиям задания:

```
bool polInt(int k) { return k > 0; }              // фактическая функция- предикат
bool polFloat(float k) { return k > 0; }
bool minSt(string a, string b) { return a > b; }
bool min(float a, float b) { return a > b; }
float mult(float res, float xi) { return res + xi; } // функция-параметр

int main(int argc, char* argv[])
{
	int m[5] = { -1,2,4,-7,0 }; sort<int>(m, 5); list<int>(m, 5);
	float x[10];
	rnd<float>(x, 10, 9); list<float>(x, 10);
	cout << endl << "Summa = "
	 << ProductOfArray<float>(x, 10, polFloat, mult) << endl;
	cout << "Min=" <<x[IndexMinMax<float>(x, 10, min)] << endl;
	sort<float>(x, 10); list<float>(x, 10);
	string st[7] = {"Mary", "true", "Hello", "false", "Santa", "ivan", "Work"};
	list(st, 7);                                  //специализация шаблона
	cout << " Min="<< st[IndexMinMax<string>(st, 7, minSt)].c_str()<<endl;
	sort<string>(st, 7);       list(st, 7);
}
```

✓ создать отдельный модуль функций-шаблонов обработки одномерных массивов;

Решение В. Для функций-шаблонов в качестве модуля может выступать заголовочный файл, так как компиляция/инстанцирование происходит во время выполнения программы-клиента и вызова соответствующего шаблона. Добавим в решение заголовочный файл.

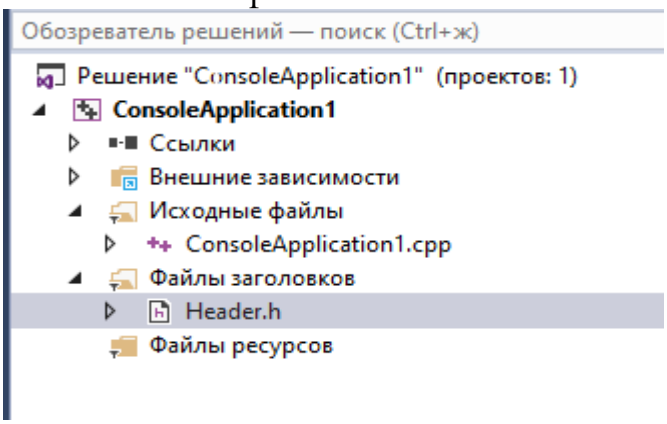

Рис.1 Добавление модуля в решение

При добавлении модуля надо правильно настроить его свойства (Включен в проект, Полный путь, Содержимое и др.) и подключить к программе-клиенту с функцией main() с помощью команды `#include "Header.h"`.

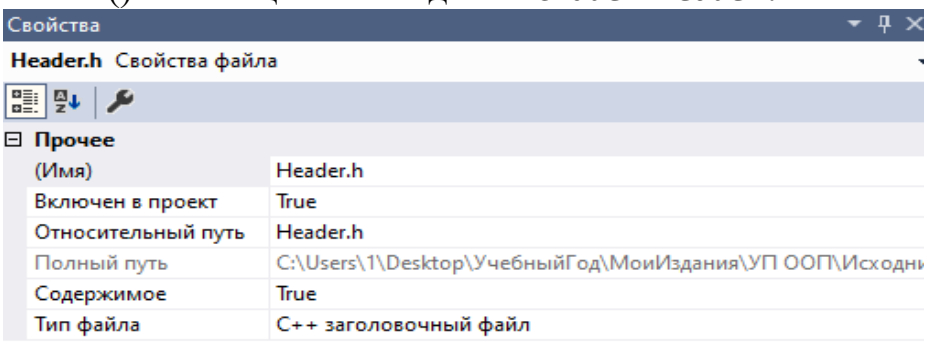

Рис.2 Свойства модуля

Заполняем модуль `Header.h` разработанными функциями- шаблонами.

```
#pragma once
#include <iostream>

using namespace std;
template <class Type>
int IndexMinMax(Type *mas, int size, bool(*predicate) (Type, Type)) {
	int Index = 0;
	for (int i = 1; i < size; i++)
		if ((*predicate) (mas[Index], mas[i]))  Index = i;
	return Index;
}

template <class Type>
void sort(Type *mas, int size) {
	Type temp;
	for (int i = 0; i < size - 1; i++)
		for (int j = i + 1; j < size; j++)
			if (mas[j] < mas[i]) { temp = mas[i]; mas[i] = mas[j]; mas[j] = temp;
}
}
```

```
template <class Type>
void list(Type* mas, int size) {
	for (int i = 0; i < size; i++)
		cout << mas[i] << '\t';
	cout << endl;
}
//специализация шаблона list для типа string
template <>
void list(string* mas, int size) {
	for (int i = 0; i < size; i++)
		cout << mas[i].c_str() << '\t';
	cout << endl;
}

template <class Type>
void rnd(Type* mas, int size, int r) {
	for (int i = 0; i < size; i++)
		mas[i] = (Type)(1 + rand() % r);
}

template <class Type>
Type ProductOfArray(Type* mas, int size, bool(*predicate) (Type), Type(*fun) (Type,
Type), Type p = 0)
{
	for (int i = 0; i < size; i++)
		if ((*predicate)(mas[i])) p = fun(p, mas[i]);
	return p;
}
```

Выполняем проект решения и проверяем работу. Все должно быть как и в предыдущем выполнении.

✓ применить модуль функций-шаблонов для решения конкретной задачи обработки одномерного массива.

Пусть вариант задания будет следующий. В одномерном массиве, состоящем из *n* целых элементов, вычислить:

a. произведение положительных элементов массива;
b. сумму элементов массива, расположенных до минимального элемента.
c. упорядочить по возрастанию отдельно элементы, стоящие на четных местах, и элементы, стоящие на нечетных местах.

Решение С.

```
#include <iostream>
#include "Header.h"

bool polInt(int k) { return k > 0; }            // фактическая функция- предикат
int multInt(int res, int xi) { return res * xi; } // функция-параметр
bool min(int a, int b) { return a > b; }
bool isTrue(int a) { return true; }
int mult(int res, int xi) { return res + xi; }


int main(int argc, char* argv[])
{
	const int n = 10;    int x[n];
	rnd<int>(x, n, 9); list<int>(x, n);
	// a.
	cout << "P = " << ProductOfArray<int>(x, n, polInt, multInt, 1)<< endl;
	// b.
	int r= IndexMinMax<int>(x, n, min);
```

```
        cout << "S = " << ProductOfArray<int>(x, r, isTrue, mult) << endl;
        // c.
        int b[n / 2 + 1], k=0; int a[n / 2 + 1], j=0;
        for (int i = 0; i < n; i++)
                if (i % 2 == 0) { a[j] = x[i]; j++; }
                   else { b[k] = x[i]; k++; }
        list<int>(b, k); sort<int>(b, k); list<int>(b, k);
        list<int>(a, j); sort<int>(a, j); list<int>(a, j);
}
```

Задание

Создать модуль шаблонов функций по обработке одномерных массивов и продемонстрировать работу библиотеки для типов int, float, string, struct (определить самостоятельно!). Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается. Основные алгоритмы, которые должны быть реализованы в библиотеке:

Основные алгоритмы, которые должны быть реализованы:

√ заполнение массива случайными значениями, включая специализацию для отдельных типов;

√ вставка и удаление k-ого элемента в массив на основе алгоритмов сдвига;

√ сортировка массива, не менее трех алгоритмов, включая один улучшенный метод сортировки;

√ поиск номеров (индексов) максимального и минимального элемента массива;

√ поиск элемента массива алгоритмом линейного и бинарного поиска (итерационная и рекурсивная реализация);

√ вычисление количества, суммы и произведения элементов массива с указанной характеристикой (положительные, отрицательные, нулевые, кратные m), используя функцию-предикат для определения характеристики как параметр функции вычисления;

√ ввод/вывод массива потоками стандартными и файловыми.

Применить модуль функций-шаблонов для решения задачи по варианту.

Варианты заданий

Вариант 1. В наборе данных, состоящем из *n* целых элементов, определить:

a. используя сортировку набора, определить значение третьего минимума;
b. произведение элементов между первым и вторым нулевыми элементами;
c. преобразовать набор таким образом, чтобы сначала располагались все кратные 3 элементы, а потом – все остальные.

Вариант 2. В наборе данных, состоящем из *n* вещественных элементов, определить:

a. максимальный элемент в наборе;
b. среднее значение элементов, расположенных между первым и последним отрицательными элементами;
c. сжать набор, удалив из него все положительные элементы. Освободившиеся в конце элементы заполнить нулями.

Вариант 3. В наборе данных, состоящем из *n* целых элементов, определить:

a. количество нулевых элементов, стоящих после отрицательных элементов;
b. произведение элементов массива, расположенных до последнего отрицательного элемента;
c. сжать набор, удалив из него все элементы, кратные k. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 4. В наборе данных, состоящем из *n* вещественных элементов, определить:

a. минимальный по модулю элемент;
b. сумму элементов, значение которых принадлежит интервалу [a, b];
c. преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом – все остальные.

Вариант 5. В наборе данных, состоящем из *n* целых элементов, определить:

a. номер первого элемента набора кратного k;
b. произведение элементов, расположенных между первым и вторым нулевыми элементами набора;
c. преобразовать набор, чтобы в начале его располагались элементы больше заданного L, а потом – остальные элементы.

Вариант 6. В наборе данных, состоящем из *n* целых элементов, определить:

a. номер максимального по модулю элемента;
b. произведение элементов, расположенных между первым и вторым нулевыми элементами.
c. сжать набор, удалив из него элементы заданного интервала номеров.

Вариант 7. В наборе данных, состоящем из *n* вещественных элементов, определить:

a. максимальный положительный элемент в наборе;
b. сумму элементов массива, расположенных между первым и вторым элементом, где десятичная часть больше 0,5;
c. преобразовать набор, добавив 0 после каждого элемента, стоящего на месте, кратным 3.

Вариант 8. В наборе данных, состоящем из *n* целых элементов, определить:
a. используя сортировку набора, определить количество элементов, совпадающих по значению с минимальным элементом;
b. среднее значение суммы модулей элементов набора;
c. преобразовать набор, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине – элементы, стоявшие в четных позициях.

Вариант 9. В наборе данных, состоящем из *n* вещественных элементов, определить:
a. используя сортировку набора, определить значение второго максимума;
b. сумму модулей элементов, расположенных после первого нулевого элемента;
c. сжать массив, удалив из него все одинаковые элементы, оставив их первые вхождения. Освободившиеся в конце элементы заполнить нулями.

Вариант 10. В наборе данных, состоящем из *n* целых элементов, определить:
a. используя сортировку массива, определить количество различных положительных элементов массива;
b. сумму элементов набора с нечетными номерами;
c. преобразовать набор данных, осуществив циклический сдвиг элементов массива вправо на одну позицию.