МИНИСТЕРСТВО РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

«КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ»

РЕФЕРАТ

по дисциплине: Основы алгоритмизации и программирования

на тему: Использование шаблонов в С++

Выполнил:

студент 1 курса 7 группы

факультет ИТ

Кишко Иван Петрович

Проверил:

доцент, к.т.н.

Белодед Николай Иванович

Минск 2020

Оглавление

[Шаблоны в С++ 3](#_Toc41856063)

[Шаблоны функций 3](#_Toc41856064)

[Примеры реализации сортировок с применением шаблонов 6](#_Toc41856065)

[Сортровка пузырьком 7](#_Toc41856066)

[Пирамидальная сортировка 7](#_Toc41856067)

[Перегрузка шаблонов 9](#_Toc41856068)

[Шаблоны классов 10](#_Toc41856069)

# Шаблоны в С++

Существуют шаблоны функций и шаблоны классов:

***Шаблоны функций****-– это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.* Другими словами, шаблон функции (шаблонная функция, обобщённая функция) представляет собой семейство разных функций (или описание алгоритма). По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными.  
  
Шаблоны классов -– обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов.

Рассмотрим более подробно шаблоны функций.

## Шаблоны функций

Допустим,нам нужно написать функцию вычисления модуля чисел.

**#include** <iostream>  
**using namespace** std;  
**int** \_abs(**int** n)  
{  
 **return** n < 0 ? -n : n;*//если отрицательное, то вернуть число с противоположным знаком, иначе само число*}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** pointA = 13, pointB = 22;  
 cout << **"Расстояние от точки А до Б = "** << \_abs(pointA - pointB);  
}

Результат:



А теперь нам нужна такая же функция, но чтобы еще и float сравнивало

**#include** <iostream>  
**using namespace** std;  
**int** \_abs(**int** n)   
{  
 **return** n < 0 ? -n : n;  
}  
  
**float** \_abs(**float** n)   
{  
 **return** n < 0 ? -n : n;  
}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **float** salary1=1323.2332, salary2=4363.34;  
 cout << **"Разница в наших доходах = "** << \_abs(salary1 - salary2);  
}

Результат:



Для каждой из функции приходится писать перегрузку, что не очень-то и хорошо. Это ухудшает читаемость кода. И если где-нибудь обнаружится ошибка, придется исправлять её в каждой функции.

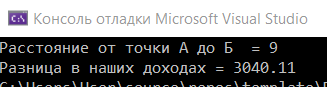
Для ликвидации этих недостатков были придуманы шаблонные функции .

**Общий синтаксис**: template<typename (имя аргумента)> затем следует оъявление функции

Определим нашу функцию \_abs шаблонной и посмотрим, что выйдет

**#include** <iostream>  
**using namespace** std;  
template<**typename** T>  
T \_abs(T arg)  
{  
 **return** arg < 0 ? -arg : arg;  
}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** pointA = 13, pointB = 22;  
 cout << **"Расстояние от точки А до Б = "** << \_abs(pointA - pointB)<<endl;  
  
 **float** salary1 = 1323.2332, salary2 = 4363.34;  
 cout << **"Разница в наших доходах = "** << \_abs(salary1 - salary2);  
}

Результат работы:



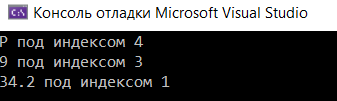
Как видно, функция корректна как для типа int, так и для float.

Сам по себе шаблон функции не вызывает генерации какого-либо кода до тех пор, пока функция не будет вызвана.Когда же функция вызывается,то компилятор знает тип передаваемой переменной, поэтому он генерирует код для функции ,подставляя вместо T передаваемый тип данных.

Пусть функция принимает 3 параметра :указатель на массив, искомое значение и размер массива ,а затем выведем результат её работы на экран:

**#include** <iostream>  
**using namespace** std;  
template <**typename** T>  
**int** find(T\* arr, T value, **int** size)*//ф-ция для поиска элемента в массиве*{  
 **for** (**int** i = 0; i < size; i++)  
 **if** (arr[i] == value) *//если нашли элемент,то возвращаем его индекс* **return** i;  
 **return** -1; *//если такого элемента нет,то возвращаем -1*}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **char** chArr[] = { 21,4,22,13,80 };  
 **char** ch = 80;*//искомое значение* **int** iArr[] = { -12,34,55,9 };  
 **int** i = 9;*//искомое значение* **float** fArr[] = { -12.34,34.2,99.01,-3 };  
 **float** f = 34.2;*//искомое значение* cout << ch << **" под индексом "** << find(chArr, ch, strlen(chArr))<<endl;  
 cout << i << **" под индексом "** << find(iArr, i, **sizeof**(iArr)/**sizeof**(iArr[0])) << endl;  
 cout << f << **" под индексом "** << find(fArr, f, **sizeof**(fArr) / **sizeof**(fArr[0])) << endl;  
}

Результат:



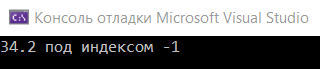
Однако если мы попытаемся передать в функцию аргументы типа int\*,float,int то компилятор выдаст ошибку, потому что все экземпляры T должны быть согласованы

**#include** <iostream>  
**using namespace** std;  
template <**typename** T>  
**int** find(T\* arr, T value, **int** size)*//ф-ция для поиска элемента в массиве*{  
 **for** (**int** i = 0; i < size; i++)  
 **if** (arr[i] == value) *//если нашли элемент,то возвращаем его индекс* **return** i;  
 **return** -1; *//если такого элемента нет,то возвращаем -1*}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** iArr[] = { -12,34,55,9 };  
 **int** i = 9;*//искомое значение* **float** fArr[] = { -12.34,34.2,99.01,-3 };  
 **float** f = 34.2;*//искомое значение* cout << i << **" под индексом "** << find(iArr, f, **sizeof**(iArr)/**sizeof**(iArr[0])) << endl;*//ошибка*}

Исправить это можно, если создать еще один шаблон аргумента

**#include** <iostream>  
**using namespace** std;  
template <**typename** T,**typename** NotT>  
**int** find(T\* arr, NotT value, **int** size)*//ф-ция для поиска элемента в массиве*{  
 **for** (**int** i = 0; i < size; i++)  
 **if** (arr[i] == value) *//если нашли элемент,то возвращаем его индекс* **return** i;  
 **return** -1; *//если такого элемента нет,то возвращаем -1*}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** iArr[] = { -12,34,55,9 };  
 **int** i = 9;*//искомое значение* **float** fArr[] = { -12.34,34.2,99.01,-3 };  
 **float** f = 34.2;*//искомое значение* cout << f << **" под индексом "** << find(iArr, f, **sizeof**(iArr)/**sizeof**(iArr[0])) << endl;  
}

Результат:

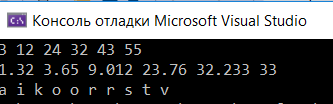


## Примеры реализации сортировок с применением шаблонов

### Сортровка пузырьком

**#include** <iostream>  
**using namespace** std;  
template<**typename** T>  
**void** bubble\_sort(T array[], **int** size)  
{  
 **for** (**int** i = 0; i < size - 1; i++)  
 **for** (**int** j = 0; j < size - i - 1; j++)  
 **if** (array[j + 1] < array[j])  
 swap(array[j], array[j + 1]);  
}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** iArr[] = { 32,43,55,3,24,12 };  
  
 bubble\_sort(iArr, **sizeof**(iArr) / **sizeof**(iArr[0]));  
 **for** (**int** i = 0; i < **sizeof**(iArr) / **sizeof**(iArr[0]); i++)  
 cout << iArr[i] << **" "**;  
  
 cout << endl;  
 **float** fArr[] = { 32.233,1.32,3.65,9.012,33,23.76 };  
 bubble\_sort(fArr, **sizeof**(fArr) / **sizeof**(fArr[0]));  
 **for**(**int** i=0;i< **sizeof**(fArr) / **sizeof**(fArr[0]);i++)  
 cout << fArr[i] << **" "**;  
 cout << endl;  
 **char** chArr[] = **"sortirovka"**;  
 bubble\_sort(chArr, **sizeof**(chArr)-1);  
 **for** (**int** i = 0; i < **sizeof**(chArr) -1; i++)  
 cout << chArr[i] << **" "**;  
}

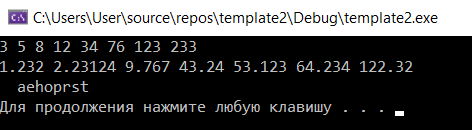
Результат:



### Пирамидальная сортировка

**#include** <iostream>  
template< **class** T >  
**void** downHeap(T a[], **long** k, **long** n)  
{  
 *// процедура просеивания следующего элемента   
 // До процедуры: a[k+1]...a[n] - пирамида   
 // После: a[k]...a[n] - пирамида* T new\_elem;  
 **long** child;  
 new\_elem = a[k];  
 **while** (k <= n / 2) *// пока у a[k] есть дети* {  
 child = 2 \* k;  
  
 **if** (child < n && a[child] < a[child + 1]) *// выбираем большего сына* child++;  
 **if** (new\_elem >= a[child])  
 **break**;  
 *// иначе* a[k] = a[child]; *// переносим сына наверх* k = child;  
 }  
 a[k] = new\_elem;  
}  
  
template< **class** T >  
**void** heapSort(T a[], **long** size)  
{  
 **long** i;  
 T temp;  
 *// Формируем нижний ряд пирамиды* **for** (i = size / 2 - 1; i >= 0; --i)  
 downHeap(a, i, size - 1);  
 *// Просеиваем через пирамиду остальные элементы* **for** (i = size - 1; i > 0; --i)  
 {  
 *// меняем первый с последним* temp = a[i];  
 a[i] = a[0];  
 a[0] = temp;  
 *// восстанавливаем пирамидальность a[0]...a[i-1]* downHeap(a, 0, i - 1);  
 }  
}  
  
**int** main()  
{  
 **int** iArr[] = { 123,3,12,233,76,8,5,34 };*//инициализация массива* **double** fArr[] = { 43.24,64.234,2.231245,53.123,122.32,1.232,9.767 };*//инициализация массива* **char** chArr[] = **"heap sort"**;*//инициализация массива* heapSort(iArr, **sizeof**(iArr) / **sizeof**(iArr[0]));*//вызов функции* **for** (**int** i = 0; i < **sizeof**(iArr) / **sizeof**(iArr[0]); i++)*//вывод отсортированного массива* std::cout << iArr[i] << **" "**;  
 std::cout << std::endl;  
 heapSort(fArr, **sizeof**(fArr) / **sizeof**(fArr[0]));*//вызов функции* **for** (**int** i = 0; i < **sizeof**(fArr) / **sizeof**(fArr[0]); i++)*//вывод отсортированного массива* std::cout << fArr[i] << **" "**;  
 std::cout << std::endl;  
 heapSort(chArr, **sizeof**(chArr) / **sizeof**(chArr[0]));*//вызов функции* **for** (**int** i = 0; i < **sizeof**(chArr) / **sizeof**(chArr[0]); i++)*//вывод отсортированного массива* std::cout << chArr[i];  
 std::cout << std::endl;  
  
 system(**"pause"**);  
 **return** 0;  
}

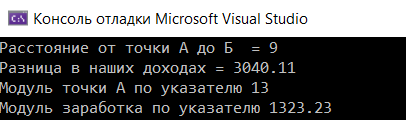
Результат:



## Перегрузка шаблонов

Шаблоны тоже можно перегружать .Разберем на примере функции \_abs

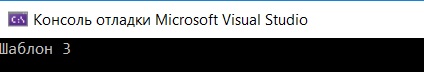
**#include** <iostream>  
**using namespace** std;  
template<**typename** T>  
T \_abs(T arg)  
{  
 **return** arg < 0 ? -arg : arg;  
}  
template<**typename** T>  
T \_abs(T \*arg)  
{  
 **return** \*arg < 0 ? -\*arg : \*arg;  
}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** pointA = 13, pointB = 22;  
 cout << **"Расстояние от точки А до Б = "** << \_abs(pointA - pointB) << endl;  
  
 **float** salary1 = 1323.2332, salary2 = 4363.34;  
 cout << **"Разница в наших доходах = "** << \_abs(salary1 - salary2)<<endl;  
  
 **int**\* pA=&pointA;  
 **float**\* pSalary=&salary1;  
 cout << **"Модуль точки А по указателю "** << \_abs(pA) << endl;  
 cout << **"Модуль заработка по указателю "** << \_abs(pSalary);  
}



Если имеется обычная функция и её шаблонная перегрузка, то предпочтение компилятор отдает обычной функции.

**#include** <iostream>  
**using namespace** std;  
template<**typename** T> **void** foo(T\*)  
{  
 cout << **"Шаблон 1"** << endl;  
}  
template<**typename** T> **void** foo(T)  
{  
 cout << **"Шаблон 2"**<<endl;  
}  
**void** foo(**int**\*)  
{  
 cout << **"Шаблон 3"** << endl;  
}  
**int** main()  
{  
 setlocale(LC\_ALL, **"ru"**);  
 **int** i =54;  
 foo(&i);  
}

Результат:



## Шаблоны классов

Для применения шаблонов перед классом указывается ключевое слово template, после которого идут угловые скобки. В угловых скобках после слова typename идет параметр шаблона. Можно определить несколько параметров шаблона.

Общий синтаксис:

template <**typename** T>  
**class** Name  
{  
*//тело шаблона класса*};

При использовании шаблона класса необходимо в угловых скобках после названия класса указать конкретный тип, который будет применяться вместо параметра T.

Например List<**int**> iList;

При вынесении методов из класса необходимо в <> указывать параметр T.При объявлении указателя на шаблонный класс/структуру так же необходимо указывать параметр: Node<T>\* head;

Применение на примере простого односвязного списка:

**#include** <iostream>  
**using namespace** std;  
template <**typename** T>  
**struct** Node *//Узел списка*{  
 T data;*//данные* Node\* next;*//указатель на след.элемент* Node();*//конструктор по умолчанию* Node(T dat,Node\* nd=NULL)*//конструктор с параметрами* {  
 data = dat;  
 next = nd;  
 }  
};  
  
template <**typename** T> **class** List *//шаблонный класс "список"*{  
**private**:  
 Node<T>\* head;*//указатель на голову списка***public**:  
 List() *//конструктор по умолчанию* {  
 head = NULL;  
 }  
 ~List()  
 {  
 **while** (head)  
 remove();  
 }  
 **void** pushStart(T val) *//добавить элемент в* {  
 Node<T>\* node = **new** Node<T>(val,head);  
 head = node;  
 }  
 **void** pushBack(T value)  
 {  
 **if** (head)  
 {  
 Node<T>\* node = head;  
 **while** (node**->**next)  
 node = node **->** next;  
 node**->**next = **new** Node<T>(value);  
 }  
 **else** head=**new** Node<T>(value);  
 }  
 **void** remove()  
 {  
 **if** (head)  
 {  
 Node<T>\* tmpHead = head;  
 head = head**->**next;  
 **delete** tmpHead;  
 }  
 }  
 **void** remove(T value)  
 {  
 Node<T>\* pointer = head;  
 Node<T>\* temp;  
 **if** (pointer != NULL)  
 {  
 **if** (pointer**->**data == value)  
 {  
 head = head**->**next;  
 **delete** pointer;  
 **return**;  
 }  
 **while** (pointer != NULL)  
 {  
 **if** (pointer**->**next != NULL && pointer**->**next**->**data == value)  
 {  
 temp = pointer**->**next;  
 pointer**->**next = pointer**->**next**->**next;  
 **delete** temp;  
 **return**;  
 }  
 pointer = pointer**->**next;  
 }  
 }  
 **else** { cout << **"Список пуст\n"**; **return**; }  
  
 cout << **"Такого элемента нет\n"**;  
 }  
 **void** showList()  
 {  
 **if** (head != NULL)  
 {  
 Node<T>\* temp=head;  
 **while** (temp != NULL)  
 {  
 cout<< temp**->**data << **" "**;  
 temp = temp**->**next;  
 }  
 cout << endl;  
 }  
 **else** cout << **"Список пуст\n"**;  
 }  
   
};  
  
**int** main()   
{  
 setlocale(LC\_ALL, **"ru"**);  
 List<**int**> iList;  
 iList.pushStart(14);  
 iList.pushBack(33);  
 iList.pushBack(5);  
 iList.pushBack(37);  
 iList.showList();  
 iList.remove(14);  
 iList.showList();  
 iList.remove();  
 iList.showList();  
  
 List<**char**> chList;  
 chList.pushStart(40);  
 chList.pushBack(33);  
 chList.pushBack(65);  
 chList.pushBack(77);  
 chList.showList();  
 chList.remove(33);  
 chList.showList();  
 chList.remove();  
 chList.showList();  
  
 List<**float**> fList;  
 fList.pushStart(40.342);  
 fList.pushBack(33.123);  
 fList.pushBack(65.034);  
 fList.pushBack(77.112);  
 fList.showList();  
 fList.remove(33.123);  
 fList.showList();  
 fList.remove();  
 fList.showList();  
 **return** 0;  
}

Результат:

