Белорусский государственный университет

Факультет прикладной математики и информатики

Слюнькова Валерия Александровна

**РЕФЕРАТ**

**на тему «Библиотека стандартных шаблонов (STL) – новые возможности (контейнеры, алгоритмы)»**

студента 1 курса 2 группы

**Преподаватель**

*Баранов Геннадий Аркадьевич*

Минск, 2017

**Содержание**

Введение

1. Структура библиотеки
2. Контейнеры
3. Алгоритмы
4. Итераторы
5. Другие возможности STL

5.1 Распределители

5.2 Функциональные объекты

5.3 Предикаты

5.4 Привязки и отрицатели

Заключение

Литература

**Введение**

Большинство компьютеров предназначено для обработки информации. В качестве данных может быть характеристика студентов, информация о наличии товаров на складе, текстовый документ, результат научных исследований и т. д. Что бы данные собой ни представляли, хранятся и обрабатываются они практически одинаковым образом. Классы С++ предоставляют отличный механизм для создания библиотеки структур данных. Раньше производители компиляторов и сторонние разработчики программного обеспечения предлагали на рынке библиотеки *классов-контейнеров* для хранения и обработки данных. Теперь же в стандарт С++ входит собственная встроенная библиотека классов-контейнеров. Она называется **Стандартной библиотекой шаблонов** **STL** (Standard Template Library) и разработана Александром Степановым и Meng Li из фирмы Hewlett Packard. STL – это часть Стандартной библиотеки классов С++, которая является удобным инструментом для хранения и обработки данных.

1. **Структура библиотеки**

Стандартная библиотека шаблонов предоставляет набор хорошо сконструированных и согласованно работающих вместе обобщённых компонентов C++*.* Все шаблонные алгоритмы работают не только со структурами данных в библиотеке, но и с встроенными структурами данных C++*.* STL позволяет программистам использовать библиотечные структуры данных со своими собственными алгоритмами, а библиотечные алгоритмы - со своими структурами данных. Хорошо определённые семантические требования и требования сложности гарантируют, что компонент пользователя будет работать с библиотекой эффективно. Эта гибкость обеспечивает широкую применимость библиотеки. STL также можно назвать достаточно эффективным инструментом. C++ успешен, потому что он объединяет колоссальную мощность с эффективностью. Этот факт объясняется тем, что каждый шаблонный компонент в библиотеке имеет обобщённую реализацию, которая имеет эффективность выполнения близкую к эффективности соответствующей программы ручной кодировки.

В библиотеке содержится несколько основных компонентов. Три наиболее важные из них – это контейнеры, алгоритмы и итераторы.

*Контейнер* – это способ организации хранения данных. Контейнеры бывают самые разнообразные и подключаются к программе с помощью шаблонных классов, что говорит о лёгкости изменения типа хранимых в них данных.

Под *алгоритмами* в STL понимают процедуры, применяемые к контейнерам для обработки данных различными способами. Например, существуют алгоритмы сортировки, копирования, поиска и объединения. Они представлены в STL в виде шаблонных функций, но не являются методами классов-контейнеров. Наоборот, эти алгоритмы являются совершенно независимыми функциями, что обеспечивает их универсальность. Таким образом алгоритмы можно использовать не только в объектах классов-контейнеров, но и в обычных массивах и собственных контейнерах. Контейнеры, тем не менее, содержат методы для выполнения некоторых специфических задач.

*Итераторы* – это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как и указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы являются неотъемлемой частью STL, так как они связывают алгоритмы с контейнерами.

1. **Контейнеры**

Контейнеры представляют собой различные структуры для хранения данных. STL включает в себя семь основных типов контейнеров и три производных. Достаточно большое количество различных типов контейнеров обеспечивает их эффективность при использовании с различными типами данных. Контейнеры STL подразделяются на две категории: *последовательные* и *ассоциативные*.

В последовательных контейнерах каждый элемент связывается с другим посредством номера своей позиции в ряду. Все элементы, кроме конечных, имеют по одному соседу с каждой стороны. Последовательные контейнеры поддерживают указанный пользователем порядок вставляемых элементов. Среди последовательных контейнеров в STL выделяют:

а) *векторы* – контейнер, который ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер вектор является наиболее предпочтительным последовательным контейнером для большинства областей применения. Существует специализация шаблона вектора для типа [bool](https://ru.wikipedia.org/wiki/Bool" \o "Bool), которая требует меньше памяти за счёт хранения элементов в виде битов, однако она не поддерживает всех возможностей работы с итераторами.

б) *списки* – это двунаправленный список, который обеспечивает двунаправленный доступ, быструю вставку и удаления в любом месте контейнера, но не поддерживает прямой доступ к элементам контейнера. Элементы списка хранятся в произвольных кусках памяти, в отличие от контейнера вектор, где элементы хранятся в непрерывной области памяти.

в) *очереди с двусторонним доступом* – похожий на вектор контейнер, но с возможностью быстрой вставки и удаления элементов на обоих концах. Реализован в виде [двусвязного списка](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA) [линейных массивов](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2). С другой стороны, в отличие от вектора, двухсторонняя очередь не гарантирует расположение всех своих элементов в [непрерывном участке памяти](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%80%D0%B0%D0%B2%D0%BD%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), что делает невозможным безопасное использование арифметики указателей для доступа к элементам контейнера.

Важно заметить, что в STL не нужно специфицировать размеры контейнеров, так как они сами заботятся о размещении своих данных в памяти.

Ассоциативный контейнер – это несколько иная организация данных. Данные располагаются не последовательно, доступ к ним осуществляется посредством ключей. Ключи используются для выстраивания хранящихся элементов в определенном порядке и модифицируются контейнерами автоматически. Среди ассоциативных контейнеров выделяют:

а) *множества* – упорядоченные наборы уникальных элементов. При вставке или удалении элементов множества итераторы, указывающие на элементы этого множества, не становятся недействительными. Реализованы на основе самобалансирующегося дерева двоичного поиска.

*б) мультимножества* – контейнеры аналогичные множествам, однако позволяющие хранить повторяющиеся элементы.

*в) отображения* - упорядоченные [ассоциативные массив](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2)ы пар элементов, состоящих из ключей и соответствующих им значений. Ключи должны быть уникальны. Порядок следования элементов определяется ключами. При этом тип ключа должен реализовывать оператор сравнения, либо требуется предоставить функцию-компаратор.

*г) мультиотображения* – контейнеры аналогичные отображениям, однако позволяющие хранить несколько одинаковых ключей.

Упорядоченные ассоциативные контейнеры (отображение, мультиотображение, множество и мультимножество) теперь поддерживают разнородный поиск. Это означает, что уже не требуется передавать объект точно такого же типа как ключ или элемент в функциях-членах, таких как find() и lower\_bound(). Можно передавать объект любого типа, для которого определен перегруженный operator<, позволяющий выполнять сравнение с типом ключа.

Следующие функции-члены в контейнерах "отображение", "мультиотображение", "множество" и "мультимножество" были перегружены для поддержки разнородного поиска:

а) find

б) count

в) lower\_bound

г) upper\_bound

д) equal\_range

Кроме того существуют специализированные контейнеры, которые можно создавать из базовых (приведенных выше) с помощью конструкции, называющейся адаптером контейнера. Контейнер-адаптер — это разновидность последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности. Контейнеры-адаптеры не поддерживают итераторы. Специализированные контейнеры, реализованные в STL, это:

*а)* *стек* - контейнер, в котором добавление и удаление элементов осуществляется с одного конца;

*б)* *очередь* - контейнер, с одного конца которого можно добавлять элементы, а с другого — вынимать;

*в)* *приоритетная очередь* – очередь организованная таким образом, что самый большой элемент всегда стоит на первом месте;

Поскольку контейнеры-адаптеры не поддерживают итераторы, их нельзя использовать в алгоритмах STL.

1. **Алгоритмы**

*Алгоритм* — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Как уже говорилось выше, алгоритмы в STL не являются методами классов и даже не являются дружественными функциями по отношению к контейнерам, как было раньше. В нынешнем стандарте языка они являются независимыми шаблонными функциями. Их можно использовать при работе как с обычными массивами C++, так и с собственными классами-контейнерами.

Алгоритмы выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью, которая представляет собой линейный список элементов внутри контейнера. Алгоритмы являются важной частью библиотеки стандартных шаблонов. Они работают с контейнерами не самостоятельно, а с помощью итераторов. Поэтому один и тот же алгоритм можно использовать с большинством, а то и со всеми контейнерами STL.  В таблице 1 показаны некоторые популярные алгоритмы.

Таблица 1. Некоторые популярные алгоритмы STL

|  |  |
| --- | --- |
| Алгоритм | Назначение |
| *Find* | Возвращает первый элемент с указанным значением. |
| *Count* | Считает количество элементов, имеющих указанное значение. |
| *Equal* | Сравнивает содержимое двух контейнеров и возвращает true, если все соответствующие элементы эквивалентны. |
| *search* | Ищет последовательность значений в одном контейнере, которая соответствует такой же последовательности в другом. |
| *Copy* | Копирует последовательность значений из одного контейнера в другой (или в другое место того же контейнера). |
| *Swap* | Обменивает значения, хранящиеся в разных местах. |
| *Sort* | Сортирует значения в указанном порядке. |
| *merge* | Объединяет два сортированных диапазона значений. |
| *accumulate* | Возвращает сумму элементов в заданном диапазоне. |
| *for\_each* | Выполняет указанную функцию для каждого элемента контейнера. |

1. **Итераторы**

*Итератор* — это объект, который может выполнять итерацию элементов в контейнере STL и предоставлять доступ к отдельным элементам. Все контейнеры STL предоставляют итераторы, чтобы алгоритмы могли получить доступ к своим элементам стандартным способом, независимо от типа контейнера, в котором сохранены элементы. Существует возможность использовать итераторы явно, с помощью члена и глобальных функций, таких как begin() и end(), а также операторов ++ и -- для перемещения вперед или назад. Можно также использовать итераторы неявно, с циклом range-for или (для некоторых типов итераторов) подстрочным оператором [ ].

В STL началом последовательности или диапазона является первый элемент. Конец последовательности или диапазона всегда определяется как элемент, следующий за последним элементом. Глобальные функции begin и end возвращают итераторы в указанный контейнер.

Косвенность со ссылок снимается оператором \*, после чего можно получить значение элемента, на который ссылается итератор. В STL итератор представляет собой объект класса iterator.

Для разных типов контейнеров используются свои итераторы. Всего существует три основных класса итераторов: прямые, двунаправленные и со случайным доступом. *Прямой итератор* может проходить по контейнеру только в прямом направлении. Проход осуществляется поэлементный. Работать с ним можно, используя ++. Такой итератор не может двигаться вобратном направлении и не может быть поставлен в произвольное место контейнера. *Двунаправленный итератор*, соответственно, может передвигаться в обоих направлениях и реагирует как на ++, так и на --. *Итератор со случайным доступом* может и двигаться в обоих направлениях, и перескакивать на произвольное место контейнера.

Есть также два специализированных вида итераторов. Это *входной итератор*, который может «указывать» на устройство ввода (cin или даже просто входной файл) и считывать последовательно элементы данных в контейнер, и *выходной итератор*, который, соответственно, указывает на устройство вывода (cout или выходной файл) и выводит элементы из контейнера.

В то время как значения прямых, двунаправленных итераторов и итераторов со случайным доступом могут быть сохранены, значения входных и выходных итераторов сохраняться не могут. Это обусловлено тем, что первые три итератора указывают на некоторый адрес в памяти, тогда как входные и выходные итераторы указывают на устройства ввода или вывода, для которых хранить какой-то «указатель» невозможно.

В STL также поддерживаются *обратные итераторы*. Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Все итераторы можно назначать и копировать. Они считаются простыми объектами и поэтому часто передаются и возвращаются по значению, а не по ссылке.  Интересно заметить, что указатель на объект всегда можно использовать как итератор произвольного доступа, поэтому он может относиться к любой категории итераторов, если он поддерживает необходимый уровень доступа для чтения и записи в последовательность, которую он обозначает.

Visual C++ теперь предоставляет *проверяемые* и *отладочные итераторы* для предотвращения перезаписи границ контейнера.   
Проверяемые итераторы гарантируют отсутствие перезаписи границ контейнера. Они применяются к сборкам выпуска и отладки.    
Необходимость в отладочных итераторах возникает, когда при выполнении программы отображается диалоговое окно, которое говорит о неверном использовании итераторов. Проблемы такого типа могут возникнуть в следующих ситуациях:

а) физическое удаления элемента из контейнера, что приводит к тому, что итератор указывает на элемент, который стал недоступным;

б) увеличение размера вектора (внедрение или вставка) , что делает итератор недоступным.

1. **Другие возможности STL**

# 5.1 Распределители

Каждый контейнер имеет определенный для него *распределитель*, который управляет выделением памяти для контейнера. Распределителем по умолчанию является объект класса allocator, но существует также возможность определять свои собственные распределители, если это необходимо для специализированных приложений. В C++ 11 и более поздних версиях распределитель по умолчанию обновлен с целью предоставления меньшего по размеру интерфейса. Новый распределитель называется минимальный распределитель. В частности, член construct() минимального распределителя поддерживает семантику перемещения, которая может значительно повысить производительность. В большинстве случаев возможностей этого распределителя по умолчанию должно быть достаточно.

**5.2 Функциональные объекты**

*Функциональные объекты* (или *функтор*) — это любой тип, реализующий operator(). Этот оператор называется *оператором вызова* или иногда *оператором приложения*. Библиотека стандартных шаблонов использует функциональные объекты  главным образом в качестве критериев сортировки для контейнеров и в алгоритмах.

Функциональные объекты  обеспечивают два основных преимущества по сравнению с прямым вызовом функций. Во-первых, функциональный объект может содержать данные. Во-вторых, он является типом и поэтому может использоваться в качестве параметра шаблона.Существует несколько предопределенных функциональных объектов, такие как less(), greater(), plus(), minus(), multiplies() и divides(). Возможно, наиболее широко используемым функциональным объектом является less(), определяющий, меньше ли один объект другого. Функциональные объекты могут использоваться вместо указателей на функции в алгоритмах STL.

* 1. **Предикаты**

Несколько из алгоритмов и контейнеров используют специальный тип функций, именуемый предикатом. Существуют две вариации предикатов: унарные и бинарные. Унарный предикат принимает один аргумент. Бинарный предикат принимает два аргумента. Эти функции возвращают результат true или false, но точные условия, которые заставляют их возвращать true или false, определяются программистом. В бинарном предикате аргументы всегда следуют в порядке первый, второй. Как для унарного, так и бинарного предикатов аргументы будут содержать значения типа объектов, хранящихся в контейнере. Некоторые алгоритмы используют специальный тип бинарного предиката, который сравнивает два элемента. Функции сравнения возвращают true, если их первый аргумент меньше второго. Функции сравнения часто обозначаются типом Comp.

* 1. **Привязки и отрицатели**

Две другие сущности, наполняющие STL — это привязки (binders) и отрицатели (negators). Привязка связывает аргумент с функциональным объектом. Отрицатель возвращает дополнение предиката. То и другое повышают разносторонность STL.

* 1. **Лямбда-выражения**

Одной из новых возможностей STL являются *лямбда-выражения* (или просто *лямбда*). Лямбда-выражения ­— это удобный способ определения анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента. Обычно лямбда-выражения используются для инкапсуляции нескольких строк кода, передаваемых алгоритмам или асинхронным методам.

В следующем примере демонстрируется простое лямбда-выражение, передаваемое функции std::sort() в качестве третьего аргумента:

#include <algorithm>

#include <cmath>

void abssort(float\* x, unsigned n)

{

std::sort(x, x + n,

// начало лямбда-выражения

[](float a, float b)

{

return (std::abs(a) < std::abs(b));

} // конец лямбда-выражения

);

}

**Заключение**

Библиотека STL — это весьма обширная область. Её описанию посвящены целые отдельные книги. STL является бесконечно расширяемой, что означает, что в нее можно дописывать новые контейнеры, алгоритмы, примитивы и всё будет совершенно монолитно и бесшовно интегрировано с STL. Также стандартная библиотека шаблонов совместима с различными платформами C++, что является её бесспорным плюсом. Она является самой эффективной библиотекой для C++, существующей на сегодняшний день.

Таким образом, STL является удобным инструментом любого программиста, с помощью которого можно упростить код программы, не причиняя ущерба ее эффективности .

**Литература**

Майерс С. Д. Эффективное использование STL. — СПб: Питер, 2002. — 224 с. — (Библиотека программиста).

Лафоре Р. Объектно-ориентированное программирование в C++. — СПб: Питер, 2013. — 928 с.

Леен Аммерааль. STL для программистов на C++. — ДМК Пресс, 1999. — С. 240.

Дэвид Р. Мюссер, Жилмер Дж. Дердж, Атул Сейни. C++ и STL: справочное руководство. — 2-е издание. — М.: «Вильямс», 2010. — С. 432. — (серия C++ in Depth).