Предисловие

Когда Дэйв Мюссер попросил меня написать предисловие ко второму изданию этой книги, я тут же откликнулся на его просьбу. Во-первых, Дэйв мой коллега — мы сотрудничали более 20 лет, и без Дэйва не было бы STL. Так что я рассматриваю его просьбу как привилегию. Кроме того, он дал мне возможность сказать несколько слов о том, о чем я думал при разработке STL.

Чтобы использовать инструмент, полезно понимать не только инструкции по его применению, но и принципы, которыми руководствовались его разработчики. Основная цель этого предисловия — познакомить вас с принципами , лежащими в основе STL.

STL разработана с учетом четырех фундаментальных идей:

- обобщенное программирование;
- абстрактность без потери эффективности;
- вычислительная модель фон Неймана;
- семантика значений.

Обобщенное программирование. Некоторые из вас могли слышать, что STL представляет собой пример технологии программирования, именуемой “обобщенным программированием”. Это так. Некоторые из вас могли слышать, что обобщенное программирование — это стиль программирования с применением шаблонов C++. Это не так. Обобщенное программирование не имеет никакого отношения к C++ и шаблонам. Обобщенное программирование — это предмет, изучающий систематическую организацию полезных программных компонентов. Его целью является разработка систематики алгоритмов, структур данных, механизмов распределения памяти и прочих программных артефактов таким образом, чтобы обеспечить максимальный уровень повторного использования, модульности и удобства.

Чтобы обеспечить наивысшую степень повторного использования, следует пытаться анализировать все возможные расширения. Например, когда известный ученый-кибернетик увидел мою реализацию алгоритма Евклида для поиска наибольшего общего делителя двух величин,

```
template <typename T> T gcd(T m, T n) {
    while (n != 0) {
        T t = m % n;
        m = n;
        n = t;
    }
    return m;
}
```

он возразил, что данный алгоритм некорректен, так как возвращает -1 при передаче ему аргументов 1 и -1, так что наибольший общий делитель не оказывается наибольшим. Он предложил исправить ситуацию, заменив последнюю строку на

```
return m < 0 ? -m : m;
```

К сожалению, в этом случае алгоритм не будет работать для многих важных расширений, таких как полиномы, комплексные целые числа и т.д. Требуется, чтобы множество элементов, с которыми мы работаем, было вполне упорядоченным. Проблема исчезает, если мы используем более абстрактное (и алгоритмически более осмысленное) определение наибольшего общего делителя: это делитель, который делится на любой другой общий делитель. Такое определение допускает неоднозначные решения: в случае целых чисел 24 и 30 наибольшими общими делителями будут 6 и -6. Это соответствует тому, что математики делали несколько последних столетий.

Классификация компонентов программного обеспечения должна работать только с полезными компонентами. Было бы нелепо вводить концепцию полупоследовательности — последовательности, у которой несколько начал, но только один конец — поскольку мы не знаем ни каких-либо структур данных, имеющих подобный вид, ни алгоритмов, способных работать с ними.

После систематизации вещей и явлений мы можем обеспечить согласованность их интерфейсов, т.е. интерфейсы двух компонентов должны быть одинаковы при одинаковом их поведении. Это позволит нам реализовать алгоритмы, которые смогут работать с несколькими компонентами — обобщенные алгоритмы. Это также обеспечит возможность использования библиотеки. Если программист овладел шаблоном STL `vector`, для него будет несложно изучить шаблон STL `list`, а научиться пользоваться шаблоном `deque` — еще проще. Я считаю, что интерфейсы, обеспечивающие наибольшую возможную степень абстрактного программирования, наиболее просты в изучении (в предположении, что человек начинает изучение с нуля. Трудно убедить программиста на Lisp, что сравнение с итератором за концом последовательности лучше проверки на равенство `nil`).

Во многих отношениях идеи обобщенного программирования подобны идеям абстрактной алгебры. Те из вас, кто прослушал курс, посвященный группам, кольцам и полям, должны увидеть происхождение классификации итераторов.[1]

Как математика организует теоремы вокруг разных абстрактных теорий, так и обобщенное программирование организует алгоритмы вокруг различных абстрактных концепций. Так что задача проектировщика библиотеки заключается в том, чтобы определить все интересующие алгоритмы и минимальные требования, которые надо удовлетворить для их работы. В общем случае эти требования описываются посредством множества допустимых выражений и их семантики. Например, STL не указывает, что `++` для итератора должно быть определено как функция-член класса. Она просто указывает, что если `i` — итератор и если он может быть разыменован, то `++i` — корректное выражение.

Абстрактность без потери эффективности. Математикам, бывает, приходится работать с объектами, которые не могут быть сконструированы вообще или могут быть сконструированы только за произвольно большое время. В области же вычислительной техники эффективность играет важную роль. Недостаточно знать, что некоторая операция может быть выполнена — важно знать, что она может быть выполнена за разумное время. Чтобы это гарантировать, STL предпринимает ряд мер.

Во-первых, она делает требование сложности частью каждого интерфейса. Когда определяется некоторая концепция, такая как итераторы, задаются некоторые требования к сложно-

[1] Вообще говоря, я считаю, что для хорошего программиста математическая культура совершенно необходима. Достаточно грустно, что в наши дни программисты заканчивают колледжи (а часто и магистратуру), так и не столкнувшись с реальной математикой. Я бы хотел посоветовать вам продолжать изучение математики на всем протяжении вашей карьеры. Что касается конкретных книг, то я бы рекомендовал трехтомник Джона Стивелла (John Stillwell) *Numbers and Geometry*, *Mathematics and Its History*, и *Elements of Algebra*; после этого можно обратиться к *Geometry: Euclid and Beyond* Робина Хартшорна (Robin Hartshorne) и *Visual Complex Analysis* Тристана Нидхэма (Tristan Needham).

сти. Программист может быть уверен, что применение операции ++ к итератору существенно от его положения в последовательности не зависит. Разыменование должно выполняться одинаково быстро — некорректно реализовывать итераторы списка при помощи структуры, содержащей указатель на заголовок списка и целочисленный индекс. (Следует заметить, что в то время как операционная семантика операций может быть строго определена при помощи множества корректных выражений и их семантики, сложность определяется неформально; требуется принципиально новое решение, чтобы найти способ для строгого, но практичного указания требований сложности.)

Во-вторых, STL принимает особые меры к тому, чтобы не скрывать никакую часть структуры данных, которая обеспечивает эффективный доступ. Вместо предоставления методов доступа для чтения и записи для работы с контейнером — любимый метод авторов учебников — используется указатель на значение, так что поля могут модифицироваться непосредственно. Можно написать

```
i->second = 5;
```

вместо

```
pair<int, int> tmp = my_vector.get(i);
tmp.second = 5;
my_vector.put(i, tmp);
```

Известно, что итераторы элементов вектора становятся некорректными после периодических перераспределений памяти, так что предполагается, что пользователи STL знают, как быть — заранее выделить достаточно памяти или хранить индексы, а не итераторы.

Были приняты меры для того, чтобы обобщенные алгоритмы в STL были технически современны и эффективны, как и закодированные вручную (говоря точнее, эффективны, как и закодированные вручную при наличии хорошего оптимизирующего компилятора).

Вычислительная модель фон Неймана. Хотя абстрактная математика использует простые числовые факты в качестве основы для своих абстракций (не забывайте, что математика — наука экспериментальная), *что* в качестве основы для абстракций должны использовать мы? По мнению фирмы, где я работаю, единственной основой является архитектура реальных компьютеров. Важно помнить, что архитектуры современных компьютеров являются результатом многолетней эволюции, руководимой необходимостью решать все более и более разнообразные задачи. Память с адресацией байтов и указатели не унаследованы нами от некоего архаического аппаратного обеспечения (в нем вообще не было ни байтов, ни указателей; циклы писались при помощи самомодифицирующегося кода), а явились результатом "погони" архитектуры за потребностями приложений.[2] Если вас интересует проектирование обобщенных схем для числовых типов, важно не только знать математическую теорию целых и действительных чисел, но и понимать, как работают встроенные числовые типы.

Наиболее важная концепция вычислительной техники, которой нет в математике, — это концепция адреса. То, что не только значения, но и адреса стали частью вычислительной модели, стало революционным шагом, обеспечившим путь от 72 адресов в ЭВМ Mark I к миллионам Интернет-адресов. Во многих аспектах наиболее спорной частью STL является тот факт, что "краеугольным камнем" всей ее доктрины стали адреса и их концептуальная клас-

[2] Для того чтобы быть хорошим программистом, важно понимать, что в действительности происходит "за кулисами" высокоуровневого языка программирования. Следует знать по крайней мере пару разных архитектур. Я уже рекомендовал несколько математических книг; позвольте мне теперь предложить вам пару компьютерных книг: книга Джона Хеннесси (John Hennessy) и Дэвида Паттерсона (David Patterson) *Computer Architecture: A Quantitative Approach* является, по моему мнению, наиболее важной книгой по вычислительной технике; она становится еще чудеснее, если сопровождается книгой *Computer Architecture: Concepts and Evolution* Геррита Блаау (Gerrit Blaauw) и Фреда Брукса (Fred Brooks), в особенности ее второй частью, освещающей вопросы исторического характера.

сификация. (Это предложение может показаться странным практикующему программисту, но академическая общественность затратила десятилетия на то, чтобы полностью удалить адреса из того, что называется "функциональным программированием".) В математической терминологии идея в основе STL состоит в том, что различные структуры данных соответствуют различным адресным алгебрам, различным способам связывания адресов. Множество операций, которые выполняют перемещение от одного адреса в структуре данных к следующему, соответствуют итераторам. Множество операций, которые добавляют адреса в структуру данных и удаляют их оттуда, соответствует контейнерам.

Классификации итераторов в STL (входные, выходные, однонаправленные, двунаправленные, произвольного доступа) достаточно для всех фундаментальных алгоритмов, работающих с последовательностями, но чтобы STL работала с многомерными структурами, следует определить новые категории итераторов. (Неоспоримым фактом является то, что даже для многих фундаментальных алгоритмов для последовательностей требуются двумерные итераторы для ускорения этих алгоритмов в случаях (1) неравномерного доступа, как, например, при итераторах дека или строках кэша, и (2) многопроцессорных реализаций.)

Семантика значений. STL рассматривает контейнеры как обобщение структур. Контейнер, как и структура, владеет своими компонентами. При копировании структур копируются все их компоненты. При уничтожении структуры уничтожаются все ее компоненты. То же самое происходит и с контейнерами. Это важнейшие свойства, которые позволяют структурам и контейнерам моделировать ключевой атрибут объектов реального мира — взаимоотношения между целым и частью. Конечно, это не единственное взаимоотношение в реальном мире, и остальные отношения должны моделироваться итераторами.[3] По моему мнению, путаница между частью и отношением, которая так распространена в объектно-ориентированных языках и библиотеках, является основным источником неприятностей в моделировании реального мира, а также основной причиной, по которой требуется сборка мусора. STL не является объектно-ориентированной — не только по способу использования глобальных обобщенных алгоритмов, но и по тому, что она разделяет понятия "иметь объект в качестве части" и "указывать на объект". Она предполагает, что строка

```
T a = b;
```

создает копию объекта со всеми индивидуальными частями, а не просто указатель на тот же объект. Спецификации алгоритмов STL, использующих присваивание (`sort`, `partition`, `remove` и т.д.), требуют такой семантики значений. Во вселенной STL объекты никогда не используют свои части совместно с другими объектами (исключая, конечно, ситуацию, когда один объект является частью другого).

В общем случае STL предполагает, что для любого типа, с которым он работает, семантики копирующих конструкторов, деструкторов, присваивания и равенства и их взаимосвязи те же, что и для встроенных типов. Кроме того, STL предполагает, что для этих объектов определены операторы `<`, `>`, `<=` и `>=`, что их семантика та же, что и для встроенных типов, или, говоря математически, они определяют полное упорядочение. (Один из недостатков C++, на мой взгляд, заключается в том, что C++ не требует согласованности семантики фундаментальных операций с семантикой встроенных типов; ничто не мешает определить оператор `=` для выполнения умножения. Перегрузка операторов хороша только при высокой самодисциплине; в противном случае она способна вызвать огромные неприятности.)

[3] Например, моя нога является частью меня, а мой адвокат — нет. Если меня уничтожить, будет уничтожена и моя нога; если меня скопировать — будет скопирована и моя нога. Мой адвокат — это другой человек, и моя смерть может повлиять на него различными путями (например, он окажется указателем на мертвого клиента, известным в программировании как висящий указатель), но не означает его автоматического уничтожения.

Развитие. STL не проектировалась в качестве части стандартной библиотеки C++. Она задумывалась как первая библиотека обобщенных алгоритмов и структур данных. Случилось так, что C++ был единственным языком, на котором я мог реализовать такую библиотеку, — тогда еще просто для собственного удовольствия. В течение пяти лет после того, как STL стала широко доступна, многие люди заявили, что они могут сделать то же, что делает STL, на своих любимых языках программирования: Ada-95, ML, Dylan, Eiffel, Java и т.д. Может быть, так оно и есть. Насколько же я мог видеть, это не так. Я бы хотел, чтобы они смогли это сделать. Я бы хотел, чтобы кто-то создал язык, в большей степени подходящий для обобщенного программирования, чем C++. В конце концов, на C++ это получилось чудом. Фундаментальные концепции STL, наподобие итераторов и контейнеров, нельзя описать на C++, поскольку STL зависит от строго определенного множества требований, которые не имеют никакого лингвистического представления на C++. (Они, конечно, определены в стандарте, но это определение на естественном языке.)

Самым главным в STL является то, что это расширяемая структура. Пока широко применяется STL, мои надежды на создание нескольких библиотек обобщенных компонентов неосуществимы. Насколько я могу определить, причина, по которой такие библиотеки не созданы, — в отсутствии механизмов финансирования для поддержки такой работы. Невозможно сделать деньги на фундаментальных алгоритмах. Такие библиотеки должны быть разработаны для всей промышленности небольшими командами разработчиков компонентов. Мне везло, и я пару раз получал средства от больших компьютерных компаний для работы над STL. Такая работа не может быть выполнена без надежного способа ее финансирования, и я надеюсь, что правительство США или ЕС создаст маленькую, но эффективную организацию для работы над обобщенными программными компонентами (я имею в виду не исследования, а реальное производство хорошо организованных, документированных, обобщенных и эффективных компонентов). Не хотите ли вы обратиться с этим предложением к своим парламентариям?

STL предполагает существенно отличный способ изучения. 99% программистов должны знать, как использовать компоненты, и не должны знать, как они работают. STL предполагает существенно отличный способ работы программистских компаний. Программисты, которые пишут собственный код вместо использования стандартных компонентов, должны быть похожи на людей, разрабатывающих собственные, нестандартные процессоры. Но сможет ли программирование войти в промышленную стадию? Я сомневаюсь…

Александр Степанов (Alexander Stepanov),
январь, 2001 год

Предисловие к первому изданию

Что такое STL? STL, или Standard Template Library (стандартная библиотека шаблонов), представляет собой библиотеку обобщенных алгоритмов и структур данных общего назначения. Она делает работу программиста более производительной двумя способами: во-первых, она содержит множество различных компонентов, которые могут быть собраны вместе и использоваться в приложениях, а во-вторых, что более важно, она предоставляет способ декомпозиции различных программных задач.

Схема, определяемая STL, очень проста: два ее фундаментальных измерения — это алгоритмы и структуры данных. Причина успешной совместной работы структур данных и алгоритмов достаточно парадоксальна: она заключается в том, что они ничего не знают друг о друге. Алгоритмы написаны в терминах итераторов, которые представляет собой абстрактные методы обращения к данным. Чтобы различные алгоритмы работали в терминах этих концептуальных категорий, STL устанавливает строгие правила, которые управляют поведением итераторов. Например, если любые два итератора равны, то и результаты их разыменования должны быть равны. Только благодаря тому, что все эти правила явно указаны в STL, можно написать код, который ничего не знает о конкретной реализации структуры данных.

Хотя мой опыт говорит о том, что применение STL может существенно повысить производительность программирования, такое повышение возможно, только если программист полностью осведомлен о структуре библиотеки и знак*ом* с поддерживаемым ею стилем программирования. Как программист может ознакомиться с этим стилем? Единственный способ состоит в том, чтобы использовать и расширять ее. Однако для этого требуется с чего-то начать, — например, с данной книги.

Авторы имеют квалификацию, достаточно высокую для написания такой книги. Дэйв Мюссер более пятнадцати лет проводил исследования, которые привели к созданию STL. Цитирую из руководства по STL: “Дэйв Мюссер … внес свой вклад во все аспекты STL: проектирование структуры, семантические требования, проектирование алгоритмов, анализ сложности и измерения производительности”. Атул Сейни был первым, кто разглядел коммерческий потенциал STL и предложил свою компанию для продажи библиотеки, еще до того как она была принята Комитетом по стандартизации C++.

Я надеюсь, что публикация этой книги поможет программистам получить от использования STL такое же удовольствие, какое получаю я.

Александр Степанов (Alexander Stepanov),
октябрь, 1995 год