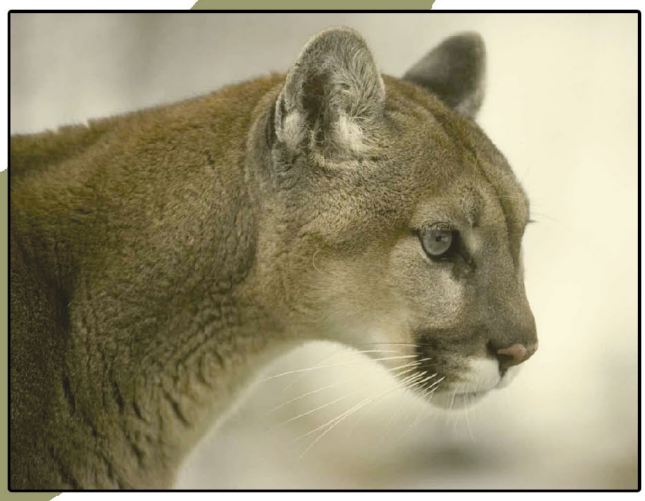
STL
для программистов на C++



Леен Аммерааль

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

STL для программистов на C++

Леен Аммерааль
Высшая школа Утрехта, Нидерланды

Москва

Леен Аммерааль
STL для программистов на C++. Пер. с англ./Леен Аммерааль – М.: ДМК. – 240 с., ил.

ISBN 5-89818-027-3

Книга Леена Аммерааля посвящена стандартной библиотеке шаблонов (STL) – мощному инструменту повышения эффективности труда программистов, пишущих на C++.

Умелое использование STL позволяет повысить надежность, переносимость и универсальность программ, а также снизить расходы на их разработку. В книге описана стандартизованная версия STL. Дается введение в предмет, которое позволяет быстро освоить библиотеку шаблонов. Приведен исчерпывающий справочный материал, в том числе по новому классу STL, string. Изложение сопровождается многочисленными примерами небольших, но законченных программ, иллюстрирующих ключевые понятия STL. Особое внимание уделено разъяснению сложных понятий библиотеки шаблонов, например, функциональных объектов и адаптеров функций.

Книга предназначена как для профессиональных программистов и тех, кто углубленно изучает C++, так и для тех, кто только начинает осваивать этот язык программирования, без преувеличения самый популярный в мире.

ISBN 0-471-97181-2

ISBN 5-89818-027-3

Содержание

Предисловие 7
1. STL для начинающих 8
1.1. Шаблоны, пространства имен и тип bool 8
1.2. Знакомство с STL 14
1.3. Векторы, списки и двусторонние очереди 20
1.4. Сортировка 24
1.5. Алгоритм find 29
1.6. Алгоритм copy и итератор вставки 30
1.7. Алгоритм merge 32
1.8. Типы, определенные пользователем 34
1.9. Категории итераторов 35
1.10. Алгоритмы replace и reverse 41
1.11. Возвращаясь к алгоритму sort 42
1.12. Введение в функциональные объекты 43
1.13. Использование find_if, remove и remove_if 45
1.14. Класс auto_ptr 49
2. Другие алгоритмы и контейнеры 52
2.1. Алгоритм accumulate 52
2.2. Алгоритм for_each 54
2.3. Подсчет 55
2.4. Функциональные объекты, определенные в STL 57
2.5. Введение в ассоциативные контейнеры 59
2.6. Множества и множества с дубликатами 60
2.7. Словари и словари с дубликатами 62
2.8. Пары и сравнения 65
2.9. Снова словари 67
2.10. Функции insert 72
2.11. Удаление элементов словаря 74
2.12. Более удобные строки 74
3. Последовательные контейнеры 81
3.1. Векторы и связанные с ними типы 81
3.2. Функции capacity и reserve 85
3.3. Обзор функций-членов класса vector 88
3.4. Двусторонние очереди 92
3.5. Списки 94
3.6. Векторы векторов 101
3.7. Как избавиться от явного выделения памяти 103
4. **Ассоциативные контейнеры** 107
4.1. Введение 107
4.2. Функции-члены множеств 111
4.3. Объединение и пересечение множеств 117
4.4. Отличия множеств с дубликатами от просто множеств 119
4.5. Словари 120
4.6. Словари с дубликатами 124
4.7. Сводный указатель 127
5. Адаптеры контейнеров 131
5.1. Стеки 131
5.2. Очереди 134
5.3. Очереди с приоритетами 135

6. Функциональные объекты и адаптеры ... 138
6.2. Функциональные объекты ... 138
6.2. Унарные предикаты и привязки ... 142
6.3. Отрицатели ... 143
6.4. Два полезных базовых класса STL ... 145
6.5. Функциональные объекты и алгоритм transform ... 147
6.6. Адаптеры итераторов ... 150
7. Обобщенные алгоритмы ... 155
7.1. Немодифицирующие последовательные алгоритмы ... 156
7.1.1. Алгоритмы find, count, for_each, find_first_of и find_end ... 156
7.1.2. Алгоритм adjacent_find ... 158
7.1.3. Отличие ... 160
7.1.4. Сравнение на равенство ... 161
7.1.5. Поиск подпоследовательности ... 162
7.2. Модифицирующие последовательные алгоритмы ... 163
7.2.1. Преобразовать ... 163
7.2.2. Копировать ... 164
7.2.3. Переместить по кругу ... 166
7.2.4. Обменять ... 168
7.2.5. Заменить ... 170
7.2.6. Удалить ... 172
7.2.7. Заполнить ... 172
7.2.8. Породить ... 173
7.2.9. Убрать повторы ... 175
7.2.10. Расположить в обратном порядке ... 178
7.2.11. Перетасовать ... 178
7.2.12. Разделить ... 180
7.3. Алгоритмы, связанные с сортировкой ... 181
7.3.1. «Меньше» и другие операции сравнения ... 182
7.3.2. Сортировка ... 182
7.3.3. Стабильная сортировка ... 182
7.3.4. Частичная сортировка ... 184
7.3.5. N-й элемент ... 186
7.3.6. Двоичный поиск ... 187
7.3.7. Объединение ... 189
7.3.8. Операции над множествами для сортированных контейнеров ... 191
7.3.9. Операции над пирамидами ... 194
7.3.10. Минимум и максимум ... 197
7.3.11. Лексикографическое сравнение ... 199
7.3.12. Генераторы перестановок ... 200
7.4. Обобщенные численные алгоритмы ... 202
7.4.1. Накопление ... 202
7.4.2. Скалярное произведение ... 202
7.4.3. Частичная сумма ... 204
7.4.4. Разность между смежными элементами ... 205
7.5. Прикладная программа: метод наименьших квадратов ... 206
8. Прикладная программа: очень большие числа ... 211
8.1. Введение ... 211
8.2. Реализация класса large ... 215
8.3. Вычисление числа π ... 229
Библиография ... 235
Указатель идентификаторов и английских названий ... 236
Предметный указатель ... 238

Предисловие

Когда несколько лет назад в языке C++ появились шаблоны, лишь немногие из программистов на C++ могли предположить, какое влияние это окажет на стандарт библиотеки языка. Стандартная библиотека шаблонов (Standard Template Library) была первоначально разработана сотрудниками Hewlett-Packard А.А. Степановым и М. Ли совместно с Д.Р. Муссером из Ренсселэровского политехнического института. После внесения незначительных поправок Комитет по стандартизации языка C++ принял STL, сделав ее существенной составной частью стандартной библиотеки.

Использование STL дает возможность создавать более надежные, более переносимые и более универсальные программы, а также сократить расходы на их разработку. Это значит, что ни один профессиональный программист не может себе позволить пройти мимо этой библиотеки. Я написал книгу для таких программистов, а также для людей, в достаточной мере знакомых с C++.

В книге описана стандартная версия STL, а не изначальный вариант, разработанный в Hewlett-Packard. Вы можете загрузить примеры, приводимые в этой книге, из сети Internet не только для того, чтобы сэкономить на набивании кода, но и для упрощения проблем с переносимостью: некоторые из электронных версий примеров сделаны более переносимыми путем добавления условной компиляции, чтобы обойти нестандартное поведение версий STL, поставляемых с компиляторами Borland и Microsoft. Все примеры доступны в виде одного файла, *stlcpp.zip*, на моем Web-сайте по адресу:

http://www.econ.hvu.nl/~ameraal/

или напрямую с одного из следующих ftp-сайтов:

ftp://ftp.expa.fnt.hvu.nl/pub/ameraal

ftp://pitel-lnx.ibk.fnt.hvu.nl/pub/ameraal

Я благодарен Гэйнору Редверс-Маттону из издательства Wiley и Фрэнсису Глассброу из Ассоциации пользователей C и C++, которые убедили меня написать эту книгу и дали полезные рекомендации по ее содержанию.

1

STL для начинающих

1.1. Шаблоны, пространства имен и тип *bool*

Как легко догадаться из названия, стандартная библиотека шаблонов (STL) основывается на относительно новом понятии *шаблона*. Поэтому мы начнем с краткого обсуждения этого предмета.

Шаблонные функции

Предположим, что для некоторого положительного числа *x* нам приходится часто вычислять значение выражения

```
2 * x + (x * x + 1) / (2 * x)
```

где *x* может быть типа *double* или *int*. В последнем случае оператор деления / обозначает целочисленное деление, дающее целый результат. Например, если *x* имеет тип *double* и равен 5.0, тогда значение приведенного выражения составляет 12.6, но если *x* имеет тип *int* и равен 5, то значение выражения будет 12. Вместо того чтобы писать две функции, такие как

```
double f(double x)
{  double x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}
```

```
int f(int x)
{  int x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}
```

нам достаточно создать один *шаблон*, как показано в следующем примере, который представляет собой законченную программу:

```
// ftempl.cpp: Шаблонная функция.
#include <iostream.h>

template <class T>
T f(T x)
{  T x2 = 2 * x;
   return x2 + (x * x + 1)/x2;
}

int main()
{  cout << f(5.0) << endl << f(5) << endl;
   return 0;
}
```

Программа выведет

```
12.6
12
```

В этом шаблоне *T* – тип, задаваемый аргументом при вызове *f*. При вызове *f*(5.0) T будет обозначать *double* (это тип константы 5.0), так что, к примеру, в выражении (*x* * *x* + 1)*/x2* выполнится деление с плавающей точкой. Напротив, при исполнении вызова *f* (5) T будет обозначать тип *int*, что приведет к целочисленному делению.

При разборе программы *ftempl.cpp* компилятор создает две различные функции, весьма похожие на функции *f*(*double*) и *f*(*int*), с которых мы начинали наш пример. Следовательно, компилятор должен одновременно «видеть» как определения, так и вызовы шаблонов. Это делает шаблоны плохими кандидатами на раздельную компиляцию; вместо этого мы, как правило, помещаем шаблоны в файлы заголовка. Когда мы используем файлы заголовка, написанные кем-то другим, мы не видим определения шаблонов и вызываем их как обычные функции, что показано на примере вызовов *f*(5.0) и *f*(5) в нашей программе. Поэтому, применяя шаблоны функций STL, мы можем и не знать, что вызываем функции, созданные из шаблонов.

Что в имени?

Шаблон, который, подобно рассмотренному выше, начинается со слова *template*, а заканчивается закрывающей фигурной скобкой, следующей

за оператором *return*, создателем языка Бьерном Страуструпом был изначально назван *шаблоном функции*. Данный термин отражает, что мы имеем дело с определенным видом шаблона, отличающимся от шаблонов классов, о которых речь пойдет ниже. Сегодня многие авторы используют вместо этого выражение *шаблонная функция (template funcion)*, потому что шаблоны указанного типа очень похожи на обычные функции. В этой книге мы также будем использовать термин *шаблонная функция*, а иногда даже просто *функция* для обозначения этих шаблонов. То же относится и к обсуждаемому ниже понятию, которое изначально получило наименование *шаблон класса*, а в книге зовется *шаблонным классом (template class)* или просто *классом*.

Шаблонные классы

Мы можем использовать тип как параметр (*T* в предыдущем примере) для классов почти так же, как и для функций. Предположим, нам нужен класс *Pair*, чтобы хранить пары значений. Иногда оба значения принадлежат к типу *double*, иногда к типу *int*. Тогда вместо двух новых классов, к примеру,

```
class PairDouble {
public:
   PairDouble(double x1, double y1): x(x1), y(y1) {}
   void showQ();
private:
   double x, y;
};

void PairDouble::showQ()
{  cout << x/y << endl;
}
```

после чего следует аналогичный фрагмент с классом *PairInt*, нам достаточно написать один *шаблонный класс*:

```
// cltempl.cpp: Шаблонный класс.
#include <iostream.h>

template <class T>
class Pair {
public:
   Pair(T x1, T y1): x(x1), y(y1){}
   void showQ();
private:
   T x, y;
};

template <class T>
```

```
void Pair<T>::showQ()
{  cout << x/y << endl;
}

int main()
{  Pair<double> a(37.0, 5.0);
   Pair<int> u(37, 5);
   a.showQ();
   u.showQ();
   return 0;
}
```

Способ, каким функции-члены шаблонного класса, в этом примере *showQ*, определены вне класса, может с первого взгляда показаться сложным. Но на самом деле эта конструкция весьма логична: имя любой функции-члена, когда эта функция определяется вне класса, должно быть предварено выражением

```
type::
```

и вполне резонно, что вместо *type* в нашем случае мы пишем *Pair<T>*. Кроме того, как пользователи STL мы можем не беспокоиться об определениях, так как шаблонные классы STL доступны в виде файлов заголовков, которые можно использовать, не вдаваясь в подробности их программирования. Единственный аспект применения шаблонов, который мы увидим в наших программах,– это обозначение фактического типа с помощью конструкции наподобие *Pair<double>*.

Пространства имен

Существует другой новый элемент языка, который мы обязаны принять во внимание. Если программа состоит из многих файлов, мы должны принять меры во избежание *конфликта имен*. Концепция пространства имен может быть хорошим способом решения этой задачи. В нижеприведенной программе определены две глобальные переменные *i*, которые не находятся в конфликте, потому что существуют в различных пространствах имен:

```
// namespac.cpp: Концепция пространства имен.
#include <iostream.h>

namespace A
{ int i = 10;
}

namespace B
{ int i = 20;
}
```

```
void fA()
{  using namespace A;
   cout << "In fA:   " <<
      A::i << " " << B::i << " " << i << endl;
}

void fB()
{  using namespace B;
   cout << "In fB:   " <<
      A::i << " " << B::i << " " << i << endl;
}

int main()
{  fA(); fB();
   cout << "In main: " << A::i << " " << B::i << endl;
   // cout << i << endl; Здесь это недопустимо.
   using A::i;
   cout << i << endl; // А это разрешено.
   return 0;
}
```

Эта программа на выходе даст:

```
In fA:   10 20 10
In fB:   10 20 20
In main: 10 20
10
```

Благодаря идентификаторам *A* и *B* мы впоследствии можем ссылаться на эти пространства имен. Для пространства имен *A* можем написать либо что-нибудь вроде

```
A:: ...
```

либо одно из выражений:

```
using namespace A;
using A::i;
```

Только после использования одного из двух последних выражений неквалифицированный идентификатор *i* будет относиться к переменной *i* (со значением 10), определенной в пространстве имен *A*. Результат работы программы наглядно демонстрирует это.

Тип *bool*: синоним для *int* или встроенный тип?

Тип *bool* и два его возможных значения, *true* и *false,* определены в файлах заголовка первоначальной версии STL с помощью приема, который часто можно встретить в программах на C:

```
#define bool int
#define true 1
#define false 0
```

Однако в соответствии с проектом стандарта C++ *bool* является встроенным типом, что подразумевает: следующая программа, вообще не использующая файлы заголовков, должна компилироваться без ошибок:

```
int main()
{  bool b;
   return 0;
}
```

Некоторые старые компиляторы отвергнут эту программу, поскольку они не распознают *bool* в качестве встроенного типа.

В соответствии с проектом стандарта C++ типы *bool* и *int* не являются идентичными, что проиллюстрируем следующими (типичными) выходными результатами программы *boolint.cpp*:

```
sizeof(bool) = 1
sizeof(int) = 4
With B defined as bool B[100], we have
sizeof(B) = 100
```

Программа, которая выводит такие результаты, приведена ниже:

```
// boolint.cpp: Типы bool и int различны.
#include <iostream.h>

int main()
{  cout << "sizeof(bool) = " << sizeof(bool) << endl;
   cout << "sizeof(int) = " << sizeof(int) << endl;
   bool B[100];
   cout << "With B defined as bool B[100], we have\n";
   cout << "sizeof(B) = " << sizeof(B) << endl;
   return 0;
}
```

Очевидно, каждый элемент массива значений типа *bool* занимает один байт, тогда как при размере машинного слова в 32 бита он занимал бы 4 байта, если бы типы *bool* и *int* не различались. Можно представить еще более экономную реализацию, когда восемь булевских значений размещаются в одном байте, но это замедлило бы операции с этими элементами.

В следующем разделе мы обсудим дополнительные различия между версиями C++ (а также STL).

1.2. Знакомство с STL

Установив современный компилятор C++, мы можем сразу начать использовать STL, например откомпилировать и запустить следующую программу. Эта программа читает с клавиатуры переменное количество ненулевых целых чисел и печатает их в том же порядке после того, как введен 0. Данная задача может показаться слишком простой, но на самом деле это не так, потому что нет ограничения на количество вводимых чисел:

```
// readwr.cpp: Чтение и вывод переменного количества
//               ненулевых целых (ввод завершается нулем).
#include <iostream>
#include <vector>
using namespace std;

int main()
{  vector<int> v;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      v.push_back(x);
   vector<int>::iterator i;
   for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
}
```

Мы можем использовать шаблон *vector* как массив переменной длины. Сначала эта длина равна нулю. Поскольку мы хотим, чтобы элементы вектора были целого типа, то всегда пишем *vector<int>*, чтобы обозначить класс, с которым работаем. Выражение

```
v.push_back(x);
```

добавляет значение *x* типа *int* в конец вектора *v*.

Оператор *for* в этой программе используется аналогично тому, как это сделано в следующем фрагменте кода, который выводит массив *a* вместо вектора *v*:

```
int a[N], *p;
...
for (p=a; p != a+N; p++)
   cout << *p << " ";
```

Напомним, что выражения *&a*[0] и *a* эквивалентны, равно как и выражения *&a*[*N*] и *a* + *N*. Начав с первого элемента, мы проходим массив, пока

не оказываемся за его концом: хотя указываем на адрес *a*[*N*], последний элемент массива – *a*[*N*–1]. Это может выглядеть опасным, но поскольку мы не используем значение *a*[*N*], а только его адрес, такой стиль абсолютно безопасен. Переменная *i*, определенная как

```
vector<int>::iterator i;
```

называется *итератором*. Она используется таким же образом, как указатель в вышеприведенном фрагменте. Значение итератора для первого элемента вектора *v* обозначается выражением *v.begin*(), а значение итератора для элемента, следующего за конечным, обозначается как *v.end*(). Значение элемента вектора, на который ссылается допустимый итератор *i*, обозначается выражением * *i*, как если бы *i* был указателем. Для этого итератора *i* определены также операторы ++ и ––, как в префиксном, так и в суффиксном варианте. Это объясняет смысл следующего цикла *for*:

```
for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
```

В приведенном цикле лучше *не* заменять != на <. Хотя в примере это сработает, но оператор < неприменим к некоторым другим типам, отличными от *vector*<*int*> (см. раздел 1.9), в то время как оператор != работает во всех случаях.

Обычно в математике запись [*a, b*] используется для обозначения закрытого интервала $a \leqslant x \leqslant b$, а запись (*a, b*) – для открытого интервала $a < x < b$. Это объясняет запись

```
[a, b)
```

для интервала

$$a \leqslant x < b$$

Подобным же образом мы иногда будем писать

```
[ia, ib)
```

для диапазона значений итератора в следующем фрагменте кода:

```
vector<int>::iterator i, ia, ib;
...
for (i = ia; i != ib; ++i) ...
```

Ошибка выделения памяти

Поскольку все числа, которые читает программа *readwr.cpp,* хранятся в динамически распределяемой памяти, используемая нами компьютерная система наложит ограничение на размер ввода. Вопрос усложняется из-за

реализации некоторыми операционными системами виртуальной памяти, что предполагает использование жесткого диска для расширения оперативной памяти. Хотя этот подход предоставляет в наше распоряжение огромное количество памяти (за счет уменьшения скорости вычислений), ясно, что рано или поздно фрагмент кода

```
vector<int> v;
for (;;) v.push_back(0);
```

приведет к ошибке выделения памяти. То же самое наблюдается при исполнении

```
int *p;
for (;;) p = new int;
```

В последнем фрагменте обычная проверка *if* (*p* != *NULL*) не будет работать с современными компиляторами C++. Согласно проекту стандарта C++ ошибка выделения памяти будет приводить не к возвращению значения *NULL,* а к «выбросу исключения». Несмотря на то что ошибка распределения памяти относится к C++, а не к STL, это была бы достаточно важная тема для обсуждения в настоящей книге, если бы существовало простое, переносимое решение, совместимое с большинством популярных компиляторов и соответствующее принятому стандарту C++. Поскольку различные компиляторы требуют разных подходов, а стандарт языка находится в стадии проекта, мы опустим обсуждение этой темы в данной книге, которая все-таки посвящена STL, а не C++.

Возвращаясь назад

Рекомендация использовать != (или ==) для сравнения значений итераторов вместо <, <=, >, >= усложняет прохождение элементов вектора в обратном порядке с помощью только что обсужденных средств. В программе *readwr.cpp* мы могли бы добиться этого, заменив цикл *for* на следующий фрагмент:

```
i = v.end();
if (i != v.begin())
   do cout << *--i << " "; while (i != v.begin());
```

В начале приведенного решения требуется дополнительное сравнение для определения пустого вектора, в случае если число 0 было единственным числом, введенным пользователем программы. Однако существует более простой путь прохождения вектора (и других структур данных) задом наперед. Он требует использования двух других функций-членов, *rbegin* и *rend,* вместе с другим типом итератора, *reverse_iterator*, как демонстрирует следующий фрагмент:

```
vector<int>::reverse_iterator i;
for (i=v.rbegin; i != v.rend(); ++i)
   cout << * i << " ";
```

Заметьте, что в этом случае мы пишем *++i* вместо *––i*.

Новые элементы языка и проблемы переносимости

В программе *readwr.cpp* способ указания файлов заголовка в строках *#include* может показаться вам непривычным (и отличающимся от того, как мы писали в разделе 1.1). Раньше мы всегда использовали *iostream.h* и *vector.h* вместо просто *iostream* и *vector*. Эти короткие формы приняты в последней версии C++, которая называется проектом стандарта C++. Во многих случаях можно использовать любой из вариантов, например, указывая как *<iostream.h>*, так и *<iostream>*. Далее мы везде будем пользоваться последним, более новым вариантом.

Интересно, что, хотя мы пишем *<iostream>*, на самом деле файл может называться *iostream.h* (например, для компилятора Borland C++ 5.2). В связи с этим возникают сомнения, правомерно ли называть *iostream* файлом заголовка. С этого раздела начнем использовать термин *заголовок*, а не *файл заголовка*. В соответствии с принятым употреблением мы будем применять этот короткий термин не только к именам, заключенным в угловые скобки, как *<iostream>*, но и к самим файлам, таким как *iostream.h*.

Другой новый аспект языка заставит нас добавлять конструкцию *using namespace std* в начале программы. Если опустим эту строчку, может появиться сообщение об ошибке типа «*Не определен символ 'vector'*». Преимущества использования концепции пространства имен, как мы обсуждали в разделе 1.1, заключаются в том, что имена наподобие *vector* не «загрязняют глобальное пространство имен». Иными словами, мы вправе использовать слово *vector* на глобальном уровне для любой другой цели. Если мы поступим так, то при необходимости разрешить неоднозначность напишем *::vector* для глобальной версии и *std::vector* – для версии STL.

Два популярных компилятора C++

Как Visual C++ 5.0 (VC5), так и Borland C++ 5.2 (BC5) поддерживают пространства имен. STL полностью интегрирована в их библиотеки, так что не требуется никаких специальных действий, чтобы начать ее использование. Многие используют эти компиляторы из интегрированной среды разработки, с помощью которой возможно редактировать программные файлы, компилировать их и т. д. Вместо этого мы будем компилировать и компоновать программы из командной строки с помощью команды *cl* или *bcc32*.

Для этого в командную строку PATH-файла *autoexec.bat* должен быть вставлен путь

```
c:\progra~1\devstudio\vc\bin
```

или

```
c:\progra~1\borland\cbuilder\bin
```

Для работы с VC5 необходимо также ввести команду

```
vcvars32
```

чтобы указать компилятору, где следует искать включаемые файлы и библиотеки.

Версии STL

Изначальная версия STL от компании Hewlett-Packard составила основу значительной части проекта стандарта C++. Эта версия доступна в сети Internet и может быть бесплатно использована и модифицирована при соблюдении нестрогих условий, указанных в следующем уведомлении о копирайте:

```
Copyright (c) 1994, Hewlett-Packard
Company.
Разрешение использовать, делать копии,
модифицировать, распространять
и продавать это программное обеспечение
и сопутствующую документацию в любых целях
предоставляется настоящим уведомлением
при условии, что как вышеприведенное
уведомление о копирайте, так и настоящее
уведомление об использовании будет
приведено в сопутствующей документации.
Компания Hewlett-Packard не несет
ответственности за пригодность
к использованию этого программного
обеспечения в каких бы то ни было целях.
Это программное обеспечение
предоставляется «как есть», без явных либо
неявных гарантий.
```

Версия STL, включенная в проект стандарта C++, отличается от изначальной (которую мы будем называть HP STL) количеством и именами заголовков. Вместо 48 файлов в HP STL в проекте стандарта C++ используется только 13 (см. табл.).

Заголовки HP STL		Стандартная STL	
algo.h	algobase.h	bool.h	algorithm
bvector.h	defalloc.h	deque.h	deque
faralloc.h	fdeque.h	flist.h	functional
fmap.h	fmultmap.h	fmultset.h	iterator
fset.h	function.h	hdeque.h	list
heap.h	hlist.h	hmap.h	map
hmultmap.h	hmultset.h	hset.h	memory
hugalloc.h	hvector.h	iterator.h	numeric
lbvector.h	ldeque.h	list.h	queue
llist.h	lmap.h	lmultmap.h	set
lmultset.h	lngalloc.h	lset.h	stack
map.h	multimap.h	multiset.h	utility
neralloc.h	nmap.h	nmultmap.h	vector
nmultset.h	nset.h	pair.h	
projectn.h	set.h	stack.h	
tempbuf.h	tree.h	vector.h	

Как VC5, так и BC5 соответствуют проекту стандарта C++ по названиям заголовков. Если у вас более старый компилятор (вместе с HP STL), вероятно, он потребует, чтобы имена заголовков содержали в себе *.h*, так что вам придется писать *<vector.h>* и *<iostream.h>* вместо *<vector>* и *<iostream>*. В этом случае будет также необходимо убрать операторы *using*, вроде того, что предшествует функции *main* в программе *readwr.cpp*.

Стандартная библиотека шаблонов фирмы SGI, адаптированная для VC++ 5.0

Хотя VC5 очень хороший компилятор, в версии STL, которая с ним поставляется, имеются некоторые проблемы. Поскольку они могут быть разрешены в следующем релизе этого компилятора, не будем их подробно обсуждать. В то же время пользователи Visual C++ могут выбрать версию STL, написанную проектировщиком STL Александром Степановым для фирмы Silicon Graphics (SGI) и адаптированную для VC5 Уэйном Учида (Wayne Ouchida). Следующая Web-страничка содержит подробную информацию по этому вопросу:

```
http://www.sirius.com/~ouchida/
```

Версия SGI STL часто упоминается в связи с ее высоким качеством, и эта адаптация поможет программистам, работающим с Visual C++, воспользоваться этим преимуществом.

В тех случаях, где современные версии STL отличаются друг от друга, за основу для примеров в этой книге берется проект стандарта C++ (на декабрь 1996 года). Если у вас возникнут проблемы с переносимостью примеров для компиляторов Microsoft или Borland, попробуйте использовать электронную версию этих программ, доступную в виде файла *stdcpp.zip*. Некоторые из них сделаны более переносимыми с помощью условной компиляции.

Кроме того, существуют также коммерческие версии STL, которые мы не будем здесь обсуждать. Поскольку STL состоит только из файлов заголовка, замена одной версии на другую является легко решаемой задачей.

1.3. Векторы, списки и двусторонние очереди

В программе *readwr.cpp* три раза встречается слово *vector*:

```
#include <vector>
...
vector<int> v;
...
vector<int>::iterator i;
```

Применение концепции вектора обеспечивает выделение непрерывной памяти, для чего программисты на C обычно пользуются функциями *malloc*, *realloc* и *free*. В качестве альтернативы можно употребить связный список, как рекомендуется в книгах по структурам данных. С помощью STL мы можем использовать (двойные) связные списки, не программируя их самостоятельно. Все, что нам требуется для программы *readwr.cpp*,– заменить всюду слово *vector* на *list*, как показано в следующей программе:

```
// readwr1.cpp: Чтение и вывод переменного количества
//              ненулевых целых (ввод завершается нулем).
//              Использует список.
#include <iostream>
#include <list>
using namespace std;

int main()
{  list<int> v;
   int x;
   cout << "Enter positive integers, followed by 0:\n";
   while (cin >> x, x != 0)
      v.push_back(x);
   list<int>::iterator i;
   for (i=v.begin(); i != v.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
}
```