STL

Собственно сам механизм шаблонов был встроен в компилятор C++ с целью дать возможность программистам C++ создавать эффективные и компактные библиотеки. Естественно, через некоторое время была создана одна из библиотек, которая впоследствии и стала стандартной частью C++. STL это самая эффективная библиотека для C++, существующая на сегодняшний день.

Сегодня существует великое множество реализаций стандартной библиотеки шаблонов, которые следуют стандарту, но при этом предлагают свои расширения, что является с одной стороны плюсом, но, с другой, не очень хорошо, поскольку не всегда можно использовать код повторно с другим компилятором. Поэтому я рекомендую вам все же оставаться в рамках стандарта, даже если вы в дальнейшем очень хорошо разберетесь с реализацией вашей библиотеки.

Начнем рассмотрение с краткого обзора основных коллекций. Каждая STL коллекция имеет собственный набор шаблонных параметров, который необходим ей для того, чтобы на базе шаблона реализовать тот или иной класс, максимально приспособленный для решения конкретных задач. Какой тип коллекции вы будете использовать, зависит от ваших задач, поэтому необходимо знать их внутреннее устройство для наиболее эффективного использования. Рассмотрим наиболее часто используемые типы коллекций. Реально в STL существует несколько большее количество коллекций, но, как показывает практика, нельзя объять необъятное сразу. Поэтому, для начала, рассмотрим наиболее популярные из них, которые с большой вероятностью могут встретиться в чужом коде. Тем более, что этих коллекций более чем достаточно для того, чтобы решить 99% реально возникающих задач.

* **vector** - коллекция элементов Т, сохраненных в массиве, увеличиваемом по мере необходимости. Для того, чтобы начать использование данной коллекции, включите #include <vector>.
* **list** - коллекция элементов Т, сохраненных, как двунаправленный связанный список. Для того, чтобы начать использование данной коллекции, включите #include <list>.
* **map** - это коллекция, сохраняющая пары значений pair<const Key, T>. Эта коллекция предназначена для быстрого поиска значения T по ключу const Key. В качестве ключа может быть использовано все, что угодно, например, строка или int но при этом необходимо помнить, что главной особенностью ключа является возможность применить к нему операцию сравнения. Быстрый поиск значения по ключу осуществляется благодаря тому, что пары хранятся в отсортированном виде. Эта коллекция имеет соответственно и недостаток - скорость вставки новой пары обратно пропорциональна количеству элементов, сохраненных в коллекции, поскольку просто добавить новое значение в конец коллекции не получится. Еще одна важная вещь, которую необходимо помнить при использовании данной коллекции - ключ должен быть уникальным. Для того, чтобы начать использование данной коллекции, включите #include <map>. Если вы хотите использовать данную коллекцию, чтобы избежать дубликатов, то вы избежите их только по ключу.
* **set** - это коллекция уникальных значений const Key - каждое из которых является также и ключом - то есть, проще говоря, это отсортированная коллекция, предназначенная для быстрого поиска необходимого значения. К ключу предъявляются те же требования, что и в случае ключа для map. Естественно, использовать ее для этой цели нет смысла, если вы хотите сохранить в ней простые типы данных, по меньшей мере вам необходимо определить свой класс, хранящий пару ключ - значение и определяющий операцию сравнения по ключу. Очень удобно использовать данную коллекцию, если вы хотите избежать повторного сохранения одного и того же значения. Для того, чтобы начать использование данной коллекции, включите #include <set>.
* **multimap** - это модифицированный map, в котором отсутствует требования уникальности ключа - то есть, если вы произведете поиск по ключу, то вам вернется не одно значение, а набор значений, сохраненных с данным ключом. Для того, чтобы начать использование данной коллекции включите #include <map>.
* **multiset** - то же самое относится и к этой коллекции, требования уникальности ключа в ней не существует, что приводит к возможности хранения дубликатов значений. Тем не менее, существует возможность быстрого нахождения значений по ключу в случае, если вы определили свой класс. Поскольку все значения в map и set хранятся в отсортированном виде, то получается, что в этих коллекциях мы можем очень быстро отыскать необходимое нам значение по ключу, но при этом операция вставки нового элемента T будет стоить нам несколько дороже, чем например в vector. Для того, чтобы начать использование данной коллекции, включите #include <set>.

STL Строки

Не существует серьезной библиотеки, которая бы не включала в себя свой класс для представления строк или даже несколько подобных классов. STL - строки поддерживают как формат ASCII, так и формат Unicode.

* **string** - представляет из себя коллекцию, хранящую символы char в формате ASCII. Для того, чтобы использовать данную коллекцию, вам необходимо включить #include <string>.
* **wstring** - это коллекция для хранения двухбайтных символов wchar\_t, которые используются для представления всего набора символов в формате Unicode. Для того, чтобы использовать данную коллекцию, вам необходимо включить #include <xstring>.

Строковые потоки

Используются для организации сохранения простых типов данных в STL строки в стиле C++. Практическое знакомство с STL мы начнем именно с этого класса. Ниже приведена простая программа, демонстрирующая возможности использования строковых потоков:

//stl.cpp: Defines the entry point for the console application

//

#include "stdafx.h"

#include <iostream>

#include <strstream>

#include <string>

using namespace std;

int \_tmain (int argc, \_TCHAR\* argv [])

{

strstream xstr;

for (int i = 0; i < 10; i++)

{

xstr << "Demo " << i << endl;

}

cout << xstr.str ();

string str;

str.assign (xstr.str (), xstr.pcount ());

cout << str.c\_str ();

return 0;

}

Строковый поток - это просто буфер, в конце которого установлен нуль терминатор, поэтому мы наблюдаем в конце строки мусор при первой распечатке, то есть реальный конец строки определен не посредством нуль терминатора, а с помощью счетчика, и его размер мы можем получить с помощью метода: pcount ().

Далее мы производим копирование содержимого буфера в строку и печатаем строку второй раз. На этот раз она печатается без мусора.

Основные методы, которые присутствуют почти во всех STL коллекциях, приведены ниже.

* empty - определяет, является ли коллекция пустой.
* size - определяет размер коллекции.
* begin - возвращает прямой итератор, указывающий на начало коллекции.
* end - возвращает прямой итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.
* rbegin - возвращает обратный итератор, указывающий на начало коллекции.
* rend - возвращает обратный итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.
* clear - удаляет все элементы коллекции, при этом, если в вашей коллекции сохранены указатели, то вы должны не забыть удалить все элементы вручную посредством вызова delete для каждого указателя.
* erase - удаляет элемент или несколько элементов из коллекции.
* capacity - вместимость коллекции определяет реальный размер - то есть размер буфера коллекции, а не то, сколько в нем хранится элементов. Когда вы создаете коллекцию, то выделяется некоторое количество памяти. Как только размер буфера оказывается меньшим, чем размер, необходимый для хранения всех элементов коллекции, происходит выделение памяти для нового буфера, а все элементы старого копируются в новый буфер. При этом размер нового буфера будет в два раза большим, чем размер буфера, выделенного перед этим - такая стратегия позволяет уменьшить количество операций перераспределения памяти, но при этом очень расточительно расходуется память. Причем в некоторых реализациях STL первое выделение памяти происходит не в конструкторе, а как ни странно, при добавлении первого элемента коллекции. Фрагмент программы ниже демонстрирует, что размер и вместимость коллекции - две разные сущности:

vector<int> vec;

cout << "Real size of array in vector: " << vec.capacity ()

<< endl;

for (int j = 0; j < 10; j++)

{

vec.push\_back (10);

}

cout << "Real size of array in vector: " << vec.capacity ()

<< endl;

return 0;

При использовании микрософтовской реализации STL библиотеки (Visual C++ 7.0) у автора получилось 0 и 13 соответственно до и после заполнения вектора.

vector

Наиболее часто используемая коллекция - это вектор. Как уже было отмечено выше, внутренняя реализация этой коллекции представляет из себя массив и счетчик элементов, сохраненных в нем. Ниже приведена программа, демонстрирующая все основные методы этой коллекции:

Предположим, нам необходимо написать логику клиент - серверного приложения. Администратор сети посылает сообщения на сервер с определенным интервалом, где они сохраняются в одном общем массиве common, при этом каждое сообщение имеет поле To, однозначно идентифицирующее каждого клиента.

Каждый клиент также подключается к серверу, но с гораздо большим интервалом, чем приход сообщений от администратора, чтобы просмотреть сообщения, адресованные ему. При этом нам также необходимо знать хронологию прихода сообщений, адресованных разным пользователям (какое сообщение пришло раньше, а какое позже в любой момент времени). Для того, чтобы получить сообщения, клиент должен подключиться к серверу, просмотреть массив common для того, чтобы выбрать сообщения, адресованные ему, и после отключиться.

В нашем случае, три клиента подключаются к серверу и каждый просматривает общий массив сообщений, при этом мы должны сделать наше приложение поточно - безопасным, поэтому должны использовать код внутри критической секции. Все это рано или поздно приведет к тому, что с увеличением числа клиентов приложение станет очень медленным.

Для того, чтобы избежать этой ситуации, мы заведем массив сообщений для каждого клиента и вместо того, чтобы просматривать общий массив сообщений три раза, мы будем просматривать его всего лишь один раз с интервалом времени, адекватным периоду подключения одного клиента. При этом скопируем все сообщения в соответствующие массивы. Клиенты же будут просто забирать данные из своих массивов при подключении.

На самом деле это немного неправильный подход для решения этой задачи. Скорее всего, нам надо было бы сохранять сообщения в момент их прихода в оба массива, но наша цель - посмотреть возможности использования коллекции vector, поэтому воспользуемся этим подходом и представим упрощенную логику такого приложения:

//stl.cpp: Defines the entry point for the console application

//

#include "stdafx.h"

#include <iostream>

#include <strstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

class MyMessage

{

private:

string from;

string to;

string message;

int id;

public:

MyMessage (string from, string to, string message)

{

this - >from = from;

this - >to = to;

this - >message = message;

}

int GetId ()

{

return this - >id;

}

void SetId (int id)

{

this - >id = id;

}

string GetMessage ()

{

return this - >message;

}

string GetFrom ()

{

return this - >from;

}

string GetTo ()

{

return this - >to;

}

};

int \_tmain (int argc, \_TCHAR\* argv [])

{

vector<MyMessage> common;

// create pool of messages for 3 users:

for (int user = 0; user < 3; user++)

{

for (int i = 0; i < 10; i++)

{

strstream messagex;

messagex << "Message " << i;

string smessage;

smessage.assign (messagex.str (), messagex.pcount ());

strstream userx;

userx << "User " << user;

string suser;

suser.assign (userx.str (), userx.pcount ());

MyMessage message ("Administrator", suser, smessage);

message.SetId (user\*10 + i);

common.push\_back (message);

}

}

// create vector for each user:

vector<MyMessage> user0;

vector<MyMessage> user1;

vector<MyMessage> user2;

for (int x = 0; x < (int) common.size (); x++)

{

MyMessage message = common [x];

if (message.GetTo () == "User 0")

{

user0.push\_back (message);

}

else

if (message.GetTo () == "User 1")

{

user1.push\_back (message);

}

else

if (message.GetTo () == "User 2")

{

user2.push\_back (message);

}

}

cout << "Messages for user 2: " << endl;

for (int i = 0; i < (int) user2.size (); i++)

{

MyMessage message = user2[i];

cout << message.GetTo () << endl;

}

cout << "Messages for user 1: " << endl;

for (int i = 0; i < (int) user1.size (); i++)

{

MyMessage message = user1[i];

cout << message.GetTo () << endl;

}

cout << "Messages for user 0: " << endl;

for (int i = 0; i < (int) user0.size (); i++)

{

MyMessage message = user0[i];

cout << message.GetTo () << endl;

}

cout << "Size of common vector: " << (int) common.size ()

<< endl;

return 0;

}

Теперь у вас есть некоторое представление о том, каким образом писать бизнес - логику приложений с использованием STL. Из этого приложения видно, что кроме перечисленных выше методов, у вектора есть оператор operator [], который позволяет нам пользоваться вектором так же, как обычным массивом. Этот оператор используется также в map, deque, string и wstring.

Итераторы

При перечислении основных методов коллекций упоминались итераторы, при этом не было дано определение этой сущности. Итератор - это абстракция, которая ведет себя, как указатель с некоторыми ограничениями или без них, то есть, сохраняет все свойства своего прародителя. Указатель - это тоже итератор. В действительности, итераторы, в большинстве случаев, это объектные обертки указателей. Вот как примерно может выглядеть внутреннее устройство итератора:

class Iterator

{

T\* pointer;

public:

T\* GetPointer ()

{

return this - >pointer;

}

void SetPointer (T\* pointer)

{

this - >pointer = pointer;

}

:

};

Но итератор представляет собой более высокий уровень абстракции, чем указатель, поэтому утверждение, что итератор - это указатель в некоторых случаях может быть неверно. А вот обратное будет верно всегда. Вот несколько формализованных определений для итератора:

Итераторы обеспечивают доступ к элементам в коллекции

Итераторы для конкретного класса коллекции определяются внутри класса этой коллекции. В STL существует три типа итераторов: iterator, reverse\_iterator, и random access iterator. Для обхода коллекции от меньшего индекса к большему, используются обычные или forward итераторы. Для обхода коллекции в обратном направлении используются reverse итераторы. Random access iterator являются итераторами, которые могут обходить коллекцию как вперед, так и назад. Ниже приведен пример использования итераторов для удаления половины элементов вектора:

#include "stdafx.h"

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void printInt (int number);

int \_tmain (int argc, \_TCHAR\* argv [])

{

vector<int> myVec;

vector<int>::iterator first, last;

for (long i=0; i<10; i++)

{

myVec.push\_back (i);

}

first = myVec.begin ();

last = myVec.begin () + 5;

if (last >= myVec.end ())

{

return - 1;

}

myVec.erase (first, last);

for\_each (myVec.begin (), myVec.end (), printInt);

return 0;

}

void printInt (int number)

{

cout << number << endl;

}

Важно помнить, что когда вы получаете итератор к коллекции, а после этого модифицируете коллекцию, то этот итератор становится уже непригодным к использованию. Естественно, не все изменения приводят к непригодности итератора для дальнейшего использования, а только изменения структуры коллекции. В случае же, если вы просто измените значения, сохраненные в коллекции, то ничего страшного не произойдет и итератор не испортится.

Итерация по коллекции вперед происходит так:

for (iterator element = begin (); element < end (); element++)

{

t = (\*element);

}

Итерация по коллекции назад происходит так:

for (reverse\_iterator element = rbegin (); element < rend ();

element++)

{

t = (\*element);

}

Если вы работаете и с random access iterator итератором, то синтаксис конструкции может быть, например, таким:

for (iterator element = begin (); element < end ();

element+=2)

{

t = (\*element);

}

Для более эффективного использования контейнеров используйте typedef или наследуйте свой класс от класса коллекции.

Сделать это можно так:

typedef vector<int> myVector

typedef map<string, int> myMap

typedef deque<string> myQue

Или вот такая техника в случае наследования:

class myVector: public vector<int> {};

В случае с итератором применима предыдущая техника:

typedef myVector::iterator vectorIterator

typedef myVector::reverse\_iterator revVectorIterator

Алгоритмы

До этого мы посмотрели основные приемы использования STL коллекций на примере использования вектора. Это основа STL, но для того, чтобы по - настоящему использовать всю мощь этой библиотеки, придется расширить наши знания. С использованием алгоритмов возможно создание очень мощных и эффективных программ. По компактности такой код превосходит код, написанный на таких современных языках, как Java и С#, и в значительной степени эффективнее последнего.

STL - алгоритмы представляют набор готовых функций, которые могут быть применены к STL коллекциям и могут быть подразделены на три основных группы:

Функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них:

**count, count\_if, find, find\_if, adjacent\_find, for\_each, mismatch, equal, search copy, copy\_backward, swap, iter\_swap, swap\_ranges, fill, fill\_n, generate, generate\_n, replace, replace\_if, transform, remove, remove\_if, remove\_copy, remove\_copy\_if, unique, unique\_copy, reverse, reverse\_copy, rotate, rotate\_copy, random\_shuffle, partition, stable\_partition**

Функции для сортировки членов коллекции:

**Sort, stable\_sort, partial\_sort, partial\_sort\_copy, nth\_element, binary\_search, lower\_bound, upper\_bound, equal\_range, merge, inplace\_merge, includes, set\_union, set\_intersection, set\_difference, set\_symmetric\_difference, make\_heap, push\_heap, pop\_heap, sort\_heap, min, max, min\_element, max\_element, lexographical\_compare, next\_permutation, prev\_permutation**

Функции для выполнения определенных арифметических действий над членами коллекции:

**Accumulate, inner\_product, partial\_sum, adjacent\_difference**

Для того, чтобы использовать все это разнообразие, у вас под рукой должна быть соответствующая документация. Microsoft предлагает достаточно подробную документацию, как часть MSDN для своей реализации STL. Достаточно подробную и обстоятельную документацию предлагает так же SGI. Для того, чтобы использовать ее, вам придется загрузить STLPort библиотеку, представляющую из себя набор из документации и хедер - файлов. Эта библиотека заслуженно считается одной из лучших, Borland уже включил ее как часть своего продукта, так что если вы используете C++ Builder 6.0, то делать это необязательно. При этом вы можете использовать эту библиотеку практически с любыми компиляторами, так что, если вы действительно стремитесь к переносимости своего кода, то это хороший выбор.

Ранее мы уже использовали один из алгоритмов: for\_each () для того, чтобы распечатать все значения из вектора. Я думаю, не требует дополнительных объяснений то, что произошло при этом. Единственное, что бы хотелось отметить, что, кроме указателя на функцию в этом случае мы могли бы передать функтор - специальный класс с перегруженным оператором operator (). Для того, чтобы показать, как это делается, ниже приведена простая программа.

#include "stdafx.h"

#include <iostream>

#include <strstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

class MyFunctor

{

string comment;

public:

MyFunctor ()

{

comment = "My comment";

};

MyFunctor (string comment)

{

this - >comment = comment;

}

void operator ()(int test)

{

cout << test << comment << endl;

};

};

int \_tmain (int argc, \_TCHAR\* argv [])

{

vector<int> test;

// fill vector:

for (int i = 0; i < 5; i++)

{

test.push\_back (i);

}

// now use our functor:

MyFunctor functor (" Test comment");

for\_each (test.begin (), test.end (), functor);

return 0;

}

Преимущество такого подхода заключается в том, что если нам необходимо передать какие - либо параметры для того, чтобы произвести обработку каждого члена коллекции, то мы имеем возможность сделать это в конструкторе функтора или определить дополнительные функции в классе - функторе. Это позволит нам сократить количество переменных в области видимости функции - обработчика членов коллекции для хранения значений, которые впоследствии будут использованы внутри тела этой функции. Если же для работы над членами коллекции нам не нужно передавать параметры, то целесообразнее определить просто функцию.

Еще один небольшой пример использования алгоритмов приведен ниже, мы создаем две коллекции: женскую и мужскую, после чего заполняем каждую из них соответствующими именами мужчин и женщин. Если мы захотим поменять местонахождение членов обоих коллекций - то есть, женщин поместить в мужскую коллекцию и наоборот, то сделать это с использованием алгоритмов очень просто:

#include "stdafx.h"

#include <iostream>

#include <strstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

void printMan (string user);

int \_tmain (int argc, \_TCHAR\* argv [])

{

vector<string> maleRoom;

vector<string> fimaleRoom;

maleRoom.push\_back ("Vasya");

maleRoom.push\_back ("Petya");

maleRoom.push\_back ("Sasha");

fimaleRoom.push\_back ("Nastya");

fimaleRoom.push\_back ("Alena");

fimaleRoom.push\_back ("Sveta");

for\_each (maleRoom.begin (), maleRoom.end (), printMan);

reverse (maleRoom.begin (), maleRoom.end ());

cout << "Males in reverse order " << endl;

for\_each (maleRoom.begin (), maleRoom.end (), printMan);

maleRoom.swap (fimaleRoom);

cout << "Now in male room are fimales: " << endl;

for\_each (maleRoom.begin (), maleRoom.end (), printMan);

return 0;

}

void printMan (string man)

{

cout << man << endl;

}

Предикаты

Для многих алгоритмов STL необходимо задать условие, посредством которого алгоритм определяет, что ему необходимо делать с тем или иным членом коллекции. По определению, предикат - это функция, принимающая один или более параметров и возвращающая значения истина или ложь. Предикат может быть функцией или функтором. Существует также набор стандартных предикатов. Рассмотрим некоторые способы использования предикатов в библиотеке стандартных шаблонов на примере алгоритмов find\_if и sort:

#include "stdafx.h"

#include <iostream>

#include <strstream>

#include <string>

#include <vector>

#include <algorithm>

#include <functional>

using namespace std;

class Man;

ostream& operator << (ostream& os, Man& man);

class Man

{

string sex;

string name;

int age;

public:

Man ()

{}

Man (string name, string sex, int age)

{

this - >name = name;

this - >sex = sex;

this - >age = age;

}

int GetAge ()

{

return this - >age;

}

void SetAge (int age)

{

this - >age = age;

}

string GetName ()

{

return this - >name;

}

void SetName (string name)

{

this - >name = name;

}

string GetSex ()

{

return this - >sex;

}

void SetSex (string sex)

{

this - >sex = sex;

}

void PrintInfo ()

{

cout << (\*this);

}

};

ostream& operator << (ostream& os, Man& man)

{

os << " - - - - - Info: - - - - - " << endl;

os << "My name is: " << man.GetName () << endl;

os << "I am " << man.GetAge () << " years old " << endl;

os << "I am " << man.GetSex () << endl;

os << " - - - - - End of Info - - - - - " << endl;

return os;

};

class ManLess

{

public:

bool operator ()(Man& man1, Man& man2)

{

if (man1.GetAge () < man2.GetAge ())

{

return false;

}

else

{

return true;

}

};

};

bool ManOlderThan23(Man& man)

{

if (man.GetAge () > 23)

{

return true;

}

else

{

return false;

}

};

class ManOlderThan

{

int m\_age;

public:

ManOlderThan (int age)

{

m\_age = age;

};

bool operator ()(Man& man)

{

if (man.GetAge () > m\_age)

{

return true;

}

else

{

return false;

}

};

};

int \_tmain (int argc, \_TCHAR\* argv [])

{

// create 3 men

Man man1("Dima", "male", 23);

Man man2("Sasha", "male", 30);

Man man3("Sergey", "male", 32);

vector<Man> programmers;

programmers.push\_back (man1);

programmers.push\_back (man2);

programmers.push\_back (man3);

// find and print all programmers older than 23

cout << "Find all programmers older than 23 " << endl;

vector<Man>::iterator p =

find\_if (programmers.begin (), programmers.end (),

ManOlderThan23);

while (p!= programmers.end ())

{

cout << (\*p);

p++;

}

// here is the same in more flexible way:

cout << "Find all programmers older than 23 " << endl;

p = find\_if (programmers.begin (), programmers.end (),

ManOlderThan (23));

for\_each (p, programmers.end (), mem\_fun\_ref

(Man::PrintInfo));

cout << "Sorted list of programmers: " << endl;

sort (programmers.begin (), programmers.end (), ManLess ());

for\_each (programmers.begin (), programmers.end (),

mem\_fun\_ref (Man::PrintInfo));

return 0;

}

На первый взгляд, этот пример выглядит довольно запутанно, но на самом деле все очень просто. Первое, что мы делаем, это включаем упреждающее объявление класса Man, оно необходимо нам для того, чтобы, в свою очередь, использовать его в упреждающем объявлении перегруженного оператора << для нашего класса Man. Теперь мы можем использовать его внутри метода класса Man. Сам класс Man не представляет из себя ничего необычного - это обычный бизнес - класс, описывающий человека.

Далее описывается предикат - функтор LessMan, необходимый для сортировки членов нашего вектора. Он принимает два параметра типа Man. Он будет использован для сортировки в порядке убывания по возрасту программистов. ManOlderThan23 - это предикат - функция, которая отбирает всех программистов старше 23 лет. После этого мы определяем точно такой же предикат - функтор ManOlder с возможностью устанавливать минимальный возраст человека в момент его создания. Такой подход гораздо гибче предыдущего.

После входа в функцию main () мы создаем вектор programmers и заполняем его программистами: Дима, Саша и Сергей. Далее мы находим и распечатываем всех программистов старше 23 лет двумя способами, после этого сортируем и распечатываем весь список наших программистов в порядке убывания по возрасту.

Стоит отметить, что эта программа будет выполнять два первых действия корректно только в случае, если все программисты отсортированы по возрастанию. Подумайте, как нужно изменить алгоритм так, чтобы они выполнялись корректно всегда.

Еще одной особенностью этого кода является то, что мы получаем указатель на функцию класса с помощью mem\_fun\_ref. Как видим, иногда это бывает очень полезно. Для того, чтобы воспользоваться этой возможностью, необходимо включить #include <functional>.

Потокобезапасность

STL не потокобезопасная библиотека, но исправить это очень просто. Предположим, вам необходимо сохранять данные в вашу коллекцию в одном потоке, когда другой поток также сохраняет их туда. Тогда просто используйте критическую секцию или Mutex.

Пример реализации потокобезопасной коллекции для WIN32 с использованием критической секции приведен ниже:

#include "stdafx.h"

#include <windows.h>

#include <iostream>

#include <strstream>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

void printInt (int number);

class MyCriticalSection

{

private:

CRITICAL\_SECTION CriticalSection;

bool success;

public:

MyCriticalSection ()

{

success = true;

// Initialize the critical section.

InitializeCriticalSection (&CriticalSection);

};

bool Lock ()

{

// Request ownership of the critical section.

\_\_try

{

EnterCriticalSection (&CriticalSection);

return true;

}

\_\_except (EXCEPTION\_EXECUTE\_HANDLER)

{

// Release ownership of the critical section.

LeaveCriticalSection (&CriticalSection);

// Release resources used by the critical section object.

DeleteCriticalSection (&CriticalSection);

success = false;

return false;

}

};

void Unlock ()

{

if (success)

{

// Release ownership of the critical section.

LeaveCriticalSection (&CriticalSection);

}

};

~MyCriticalSection ()

{

// Release resources used by the critical section object.

if (success)

{

DeleteCriticalSection (&CriticalSection);

}

};

};

// define thread safe vector of integers

class VectorInt: public vector<int>, MyCriticalSection

{

public:

VectorInt (): vector<int>(), MyCriticalSection ()

{}

void safe\_push\_back (int arg)

{

Lock ();

push\_back (arg);

Unlock ();

}

};

int \_tmain (int argc, \_TCHAR\* argv [])

{

VectorInt vx;

for (int i = 0; i < 5; i++)

{

vx.safe\_push\_back (i);

}

for\_each (vx.begin (), vx.end (), printInt);

return 0;

}

void printInt (int number)

{

cout << number << endl;

}

Для начала мы создаем объектную обертку для критической секции. Далее создаем класс, который будет потомком двух классов: вектора с целочисленным параметром и нашей обертки и добавляем в него нашу потокобезопасную функцию, внутри которой вызываем метод вектора для добавления нового элемента в коллекцию.

Заключение

Достоинство STL - это то, что библиотека действительно является кроссплатформенной. И это не пустая декларация, как происходит со многими другими технологиями. Я думаю, что существует больше платформ, не поддерживающих Java, чем компиляторов C++ для этих же платформ, не поддерживающих STL. Конечно, не существует абсолютной гарантии, что она встроена абсолютно во все компиляторы C++. Например, некоторые компиляторы для мобильных устройств и микроконтроллеров не включают эту библиотеку. Это обусловлено тем, что она является относительно неэффективной в плане использования памяти, поскольку оптимизирована для обеспечения максимальной скорости. В мобильных устройствах, как известно, самый дорогой ресурс - это память, в то время как на вашем PC сегодня его в избытке. Поэтому иногда вам придется писать шаблонные классы, похожие на классы STL, самостоятельно для того, чтобы например перенести приложение, работающее под Windows или Linux на мобильное устройство. Автор этой статьи, например, реализовал класс vector, который более бережно относится к памяти, чем его прототип, именно для такого проекта.

Автор постарался выбрать все наиболее ценное для использования на практике. Все примеры были оттестированы в среде VC7++. При создании использовался тип консольного Win32 приложения без поддержки ATL и MFC. Автор надеется, что они также хорошо будут работать при компиляции на других платформах. Естественно, за исключением приложения, демонстрировавшего создание потокобезапасной коллекции, все их можно собрать, скажем на Linux с использованием GCC. Как это делать, было подробно описано в моей предыдущей статье.