boost

Русанов Алексей. ВМИ — 313

Что за зверь, этот boost ?

Библиотека ***boost C++*** - это собрание множества независимых библиотек, созданных независимыми разработчиками и тщательно проверенными на различных платформах.


Нужно добавить регулярные выражения в **C++11**, ваши предложения?


Пишем сами.
Нафиг они нужны.
В **boost** уже есть, возьмем с него?

Особенности boost.

boost обладает двумя уникальными особенностями:

1. Он имеет тесные связи с комитетом по стандартизации C++ и способен влиять на его решения. boost был основан членами этого комитета, и участники одного часто являются также членами другого.

2. Вдобавок к этому boost всегда провозглашал одной из своих целей служить платформой для тестирования средств, которые могут быть добавлены в Стандарт C++.

Библиотеки, входящие в boost.

boost содержит десятки библиотек, и к ним постоянно добавляются новые. Библиотеки сильно отличаются по размерам и областям применения. С одной стороны находятся библиотеки, концептуально требующие лишь нескольких строк кода.

Другую крайность составляют библиотеки, представляющие настолько широкие возможности, что им можно посвящать целые книги.

Некоторые библиотеки

Обработка строк и текстов. Сюда входят библиотеки для безопасного по отношению к типам форматирования (по аналогии с printf), работы с регулярными выражениями, а также лексического и грамматического анализа.

Контейнеры. Сюда входят библиотеки для работы с массивами фиксированной длины с STL-подобным интерфейсом, битовыми наборами произвольной длины, а также многомерными массивами.

Некоторые библиотеки

Функциональные объекты и высокоуровневое программирование. Эта категория объединяет несколько библиотек, которые лежат в основе функциональности TR1.

Математика и численные методы. Сюда входят библиотеки для работы с рациональными числами, поиска наибольшего общего делителя и наименьшего общего кратного, а также для операций со случайными числами.

Некоторые библиотеки

Структуры данных. Сюда отнесены библиотеки для поддержки безопасных по отношению к типам объединений (то есть «любых» неоднородных типов) и библиотека кортежей, которая нашла применение в TR1.

Память. Сюда входит библиотека Pool для высокопроизводительных распределителей памяти блоками фиксированного размера а также целый ряд «интеллектуальных» указателей включая те, что вошли в TR1 (но не только).

Некоторые библиотеки

boost предлагает библиотеки для решения самых разных задач, но они, конечно, не покрывают всех тем, которыми занимаются программисты. Так, например, нет библиотеки для разработки графических интерфейсов, как нет и библиотек для доступа к базам данных.

boost::regex_iterator

Данный итератор может быть удобен для последовательного поиска вхождений подстроки, соответствующей регулярному выражению.

```
string xStr("AAAA-12222-BBBBB-44455");
regex xRegEx("(\\w|\\d)+");
smatch xResults;
sregex_iterator xIt(xStr.begin(), xStr.end(), xRegEx);
sregex_iterator xInvalidIt;
while(xIt != xInvalidIt)
  std::cout << *xIt++ << “\t";
```

Результат

AAAA 12222 BBBBB 44455

boost::tokenizer

Нередко в практике программистов встречаются задачи, когда нужно разобрать строку Решений этой задачи - множество. Например простейший поиск очередного символа-разделителя.

```
string Str("Hello. I’m boost");
char_separator<char> Sep('.');
smatch xResults;

tokenizer <char_separator<char>> tokens(Str, Sep);
BOOST_FOREACH(const string& t, tokens)
cout<<t<<endl;
```

Результат

```
Hello.
I’m boost
```

boost::any

Шаблонный класс, представляющий собой универсальный контейнер, в котором может храниться всё, что угодно. Но для доступа к данным необходимо точно знать хранимый тип.

```
any a = 10;
a = string("Hello world");
string s = any_cast<string> (a);
```

! В случае, если параметр типа шаблонной функции any_cast не совпадает с типом хранимого значения, будет выброшено исключение. !

boost::bind

Один из ключей к пониманию того, как использовать bind – концепция заменителей. Заменители символизируют аргументы, которые должны передаваться результирующему функциональному объекту. boost::bind поддерживает до девяти таких аргументов. Заменители обозначаются как _1 … _9, и используются в тех местах, где обычно находятся аргументы функции.

```
void show( int i1, int i2) {cout<<i1<<'\t'<< i2<<'\n';}
int i1 = 1, i2 = 2;
(bind(&nine_arguments, _2, _1)) (i1, i2);
```

Результат

```
2	1
```

boost::random

Случайные числа полезны в различных приложениях. boost::random обеспечивает разнообразие генераторов и распределений для генерации случайных чисел, имеющих полезные свойства, такие как равномерное распределение.

```
mt19937 rng;
uniform_int_distribution<> six(1,6);
int x = six(rng);
```

Результат

От 1 до 6

boost::tuple

Тип **tuple** это **коллекция** элементов фиксированного размера. В языке программирования **tuple** это объект данных, содержащий другие объекты как элементы. Эти элементы могут быть разных типов.

```
tuple<string, double> Tuple("Pi", 3.14)
string TupleStr = Tuple.get<0>();
cout << TupleStr;
```

Результат

```
Pi
```

Немножко об **auto_ptr**

Следует обратить внимание на этот умный указатель. Здесь проявляется одна любопытная особенность ***auto_ptr***, незнание которой чревато серьёзными ошибками.

```
auto_ptr<A> a(new A);
auto_ptr<A> b(a);
```

Дело в том, что ***auto_ptr*** реализует так называемое *разрушающее копирование*.

Немножко об **auto_ptr**

Сначала мы создаём экземпляр класса A и передаём его во владение умному указателю a. Затем мы создаём ещё один умный указатель b и передаём владение экземпляром A ему. При этом указатель a должен быть сброшен, иначе одним и тем же экземпляром объекта будут владеть два умных указателя, и в результате мы получим двойное удаление одного и того же экземпляра при выходе из области видимости.

! Разрушающее копирование – это особенность поведения ***auto_ptr***, заложенная в его архитектуре. !

Умные указатели: scoped_ptr

scoped_ptr - не копируемые «автоматические» указатели. Используется для длительного контроля за указателем. Т.е. ***scoped_ptr*** представляет собой облегчённую версию ***auto_ptr***, запрещающую копирование и присваивание. По сути это всего-навсего аналог ***auto_ptr*** из STL.

```
scoped_ptr<SomeClass> ptr(new SomeClass);
ptr->SomeFunc();
```

Умные указатели: shared_ptr

shared_ptr - указатели с подсчетом ссылок. Используется для контроля времени жизни указателя в пределах операторного блока.

shared_ptr обладает во многом схожей с ***auto_ptr*** семантикой, основные отличия заключаются в методике копирования. Класс ***shared_ptr*** реализует разделяемое (shared) владение с подсчётом ссылок, что позволяет использовать более привычные конструкторы копирования и операторы присваивания. Благодаря такой семантике ***shared_ptr*** можно использовать в стандартных контейнерах STL.

В итоге мы имеем реализацию умного указателя, экземпляры которого совместно владеют экземпляром объекта, на который они удерживают ссылку. Такая семантика позволяет использовать ***shared_ptr*** вместо обычного указателя, практически полностью сохраняя поведение.

Умные указатели: weak_ptr

Представим себе случай, когда нам нужно передать указатель, не передавая права владения объектом. Использовать для этой цели ***shared_ptr*** нельзя, т.к. его копирование увеличит счётчик ссылок, и в результате происходит разделение прав владения. Использование обычного указателя также нежелательно, т.к., как уже говорилось в самом начале, невозможно проверить, ссылается указатель на существующий объект или нет.

Специально для такого случая предусмотрен ещё один вид умных указателей: ***weak_ptr***.

weak_ptr – это «слабый», не владеющий экземпляром объекта умный указатель.

Умные указатели: ptr

- ***intrusive_ptr*** представляет собой облегчённую версию ***shared_ptr***, специально предназначенную для классов, имеющих встроенные механизмы подсчёта ссылок. Для таких классов ***intrusive_ptr*** позволяет реализовать эффективный механизм совместного владения с подсчётом ссылок, но без дополнительных затрат. Соответственно, отсутствует и аналог ***weak_ptr***. Во всём остальном семантика ***intrusive_ptr*** повторяет семантику ***shared_ptr***.

Умные указатели: scoped/shared_array

Классы ***scoped_array*** и ***shared_array*** представляют собой семантические аналоги ***scoped_ptr*** и ***shared_ptr***, но применимо к динамически выделяемым массивам объектов. Отличие состоит только в реализации оператора [] вместо операторов * и ->. Кроме того, ***shared_array*** в отличие от ***shared_ptr*** не реализует копирующие конструкторы и операторы присваивания, позволяющие преобразовывать типы значений – такие преобразования опасны и в большинстве случаев приводят к неопределённому поведению, либо к нарушению защиты памяти.

```
scoped_array<int> sca;
sca.reset(new int[10]);

shared_array<int> sha;
sha.reset(new int[10]);
```

Умные указатели: scoped/shared_array

```
class A { };
class B : public A { };

shared_ptr<B> b(new B);
shared_ptr<A> a(b); // OK

shared_array<B> b_arr(new B[10]);
shared_array<A> a_arr(b_arr); // Ошибка
```

Итого

- boost – очень мощная библиотека, содержащая множество библиотечек для различных нужд.
- Порог вхождения очень высок, для использования boost'а нужно обладать определенными знаниями C++.
- Были созданы многие аналогичные конструкции из языков C# и Java, но которых не было в C++.
- Умные указатели – просто чудо.