Реализация новых парадигм программирования в C++: плюсы и минусы

Автор: *Григорьев Вячеслав Владимирович*

Опубликовано: *29.12.2011*
Исправлено: *24.04.2012*
Версия текста: *1.1*

Введение
Идеал и реализация
Пример: парадигма функционального программирования
Качество реализации парадигмы функционального программирования
Диагностика
Заключение
Список литературы

Введение

Язык программирования C++ является мультипарадигменным. В переводе на русский это означает, что в рамках этого языка можно разрабатывать программы, используя совершенно разные парадигмы программирования: процедурную, объектно-ориентированную и другие. Несомненным достоинством языка является возможность реализовывать библиотеки, предоставляющие среды для программирования с совершенно иными парадигмами. Ярким примером таких библиотек являются меташаблонные библиотеки из набора **boost** (www.boost.org): **lambda**, **phoenix**, **spirit** и другие. Например, **lambda** и **phoenix** предлагают разработчику в C++ возможность функционального программирования, изначально языку не присущего. Что, несомненно, очень удобно для тех, кто привык к такой парадигме и в других языках.

Реализация библиотеки, предлагающей пользователю новую парадигму программирования, как правило, связана с некоторым расширением языка, с созданием нового языкового уровня, называемого «языком предметной области» (Domain Specific Language – DSL). Конечному пользователю библиотеки предлагается выражение логики его программы либо в рамках DSL, либо на всём пространстве нового языка [DSL + изначальный язык] (рис.1).

Рис.1

Примечание

Ниже речь пойдёт о версии языка, наиболее широко используемой мировым сообществом до сентября 2011 года – ISO/IEC 14882:1998 / ISO/IEC 14882:2003. В сентябре случилось знаковое событие – был утверждён новый стандарт языка – ISO/IEC 14882:2011, неформально

называемый C++0x.

Идеал и реализация

Как правило, разработчик, владеющий навыками работы в нескольких парадигмах, выбирает ту, которая более удобна для текущей задачи. Например, основная часть кода может быть написана в стиле объектно-ориентированного программирования, а отдельные небольшие части – в стиле функционального программирования.

Рис. 2

Чтобы не быть голословными, рассмотрим весьма распространённый пример использования функции **for_each** из стандартной библиотеки шаблонов (STL) C++:

```
void f1(const std::string &s)
{
    std::cout << s << std::endl;
}

void work()
{
    std::vector<std::string> v;

    // ...
    std::for_each(v.begin(), v.end(), &f1);
}
```

В примере эта функция используется для «пробегания» по всем элементам вектора **v** и вызова для каждого элемента функции **f1**. В результате получается построчный вывод содержимого вектора в стандартный поток вывода **std::cout**. Код вывода содержимого каждого элемента, однако, настолько прост, что здесь напрашивается использование какого-нибудь временного функционального объекта. В самом деле, зачем определять в коде новую вспомогательную функцию **f1**, если её алгоритм можно выразить в том месте, где она используется? Как минимум, это не засоряет пространство имён лишними конструкциями, которые нужны только в одном конкретном месте. Как максимум, делает код более читабельным, так как в реальном исходном тексте функции **f1** и **work** могут находиться далеко друг от друга, и для понимания логики работы придётся воспользоваться текстовым поиском (или различными assistant-средствами IDE) с возможными прыжками по разным файлам.

Используя какую-нибудь библиотеку поддержки функционального программирования, можно преобразовать код к такому виду:

```
void work()
{
    std::vector<std::string> v;

    // ...
    std::for_each(v.begin(), v.end(), std::cout << _1 << std::endl);
}
```

Безусловно, это гораздо нагляднее и лучше передаёт смысл осуществляемого действия. В конечном счёте, это более удобно.

Удобство работы в рамках той или иной парадигмы зависит, помимо её основополагающих свойств и применимости к данной задаче, в том числе и от удобства её реализации. Для большинства людей очевидно, что реализация парадигм средствами самого языка программирования не вызывает вопросов о качестве, если это такой известный язык, как C++. То есть, проще говоря, процедурное и объектно-

ориентированное программирование В нём может быть выполнено настолько качественно, насколько на это способен конкретный разработчик. Однако качество реализации парадигм средствами не языка, а библиотек, может быть абсолютно разным. А потому удобство программирования В рамках этих парадигм будет определяться не только навыками разработчика.

Критерием качества, который можно выделить В первую очередь, В данном случае служит точность Выражения и использования элементов парадигмы В рамках заданного исходного языка программирования (host-языка, C++ В этой статье). Эта точность ограничивается наличием подходящих синтаксических конструкций языка, которые аВтор библиотеки может переопределить и использоВать по сВоему усмотрению. Например, если В реализуемой парадигме есть ключеВой синтаксический элемент **«if»**, то В рамках C++ придётся придумыВать различные заменители типа **«if_»** или **«operator_if»**. Вторым критерием качества яВляется прозрачность использоВания конструкций из новой парадигмы. В идеале для разработчика Всё должно Выглядеть так, как будто он пишет код В новой среде, которая и была написана специально для работы с этой парадигмой. Влияние нюансов и особенностей исходного языка на Выражения программных конструкций из новой парадигмы должно быть сВедено к минимуму.

Пример: парадигма функционального программирования

Наиболее иЗВестными библиотеками, предоставляющими среду для программирования В рамках этой парадигмы, яВляются библиотеки **lambda** и **phoenix** из набора **boost**. Эта парадигма подразумеВает использоВания блоков программы как неименованных Временных функциональных объектоВ. В примере Выше Вместо функции **f1** как раз и был ВВедён такой блок В качестВе третьего параметра В функцию **std::for_each**.

Парадигма функционального программирования предлагает пользоВателю описыВать блоки кода В качестВе Временных объектоВ, используя тот же язык программирования, что и В остальной части программы. То есть В идеале должно получиться что-то Вроде того, что уже реализоВано В последней Версии стандарта языка C++:

```
...
std::for_each(v.begin(), v.end(),
   [](std::string s){ std::cout << s << std::endl; });
```

Выражение Внутри "{" и "}", стоящее третьим параметром, должно быть абсолютно таким же корректным C++-кодом, что и остальная часть программы.

Сложность качественной реализации этой парадигмы определяется обоими критериями качества, что были упомянуты Выше. Во-первых, аВтору библиотеки не удастся просто так использоВать ключеВые слоВа языка типа **«if»**, чтобы реализоВать сВой функциональный объект. Придётся делать какие-то синтаксические заменители. Во-Вторых, привычные глазу пользоВателя C++-конструкции, состоящие из наборов операторов, Выражений и т.п., стоящие Внутри блока, должны разВорачиВаться не В привычный код, который создаётся компилятором В обычном случае, а В искусственные Внутренние конструкции, которые, В конечном итоге, породят Выполняемый объект. Ведь Выражение **«std::cout << s << std::endl»** на языке C++ означает «Вывести В объект **std::cout** строку **s** с переВодом строки» именно В том месте, где оно написано. А Выражение для Временного Выполняемого объекта **«[](std::string s){ std::cout << s << std::endl; }»** означает «создать Выполняемый объект, который, *когда будет Вызван*, Выполнит ВыВод переданного параметра и перевода строки В объект **std::cout**». И поВедение этих самых Внутренних искусственных конструкций, генерируемых библиотекой, значительно отличается от того кода, что создал Бы компилятор, будь это обычная C++-программа.

Чтобы лучше представить, что же на самом деле происходит, и какие Вспомогательные объекты создаются, рассмотрим следующий пример реализации подобной библиотеки. Это очень упрощённый пример, Весьма далёкий от того, чтобы его использоВать на практике, но, тем не менее, работающий.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <string>

// Определим заглушку, которая будет использоваться В Выражении В тех местах,
// где должен подставляться передаВаемый при Вызове параметр. Для простоты
// ограничимся поддержкой ВызоВов только с одним параметром.
template <int N> struct placeholder;

template <> struct placeholder<1>
{
    template <typename T1>
    T1& operator()(T1 &t1) { return t1; }
};
```

```
placeholder<1> _1;

// Определим простейший функтор, единственная задача которого - сохранить ссылку
// на переданный объект произвольного типа и возвращать её независимо от фактических
// параметров, передаваемых при вызове.
template <typename T> struct functor_ref
{
    T &_t;
    functor_ref(T &t) : _t(t) {}

    template <typename T1>
    T& operator()(T1&) { return _t; }
};

// Определим функтор, который инкапсулирует оператор << для любых левого и правого
// выражений. Они должны также представлять собой функторы этой микро-библиотеки.
template <typename Tl, typename Tr> struct functor_out
{
    Tl _l;
    Tr _r;
    functor_out(const Tl &l, const Tr &r) : _l(l), _r(r) {}

    template <typename T1>
    void operator()(T1 &t1)
    {
        _l(t1) << _r(t1);
    }
};

// Определим «свободный» оператор «<<» для любого выражения, у которого справа
// стоит определённая выше заглушка.
template <typename T, int N>
functor_out<functor_ref<T>, placeholder<N> > operator<<(T &l, placeholder<N> &r)
{
    return functor_out<functor_ref<T>, placeholder<N> >(functor_ref<T>(l), r);
}


int main(int argc, char* argv[])
{
  std::vector<std::string> v;
  v.push_back("test1\n");
  v.push_back("test2\n");

  std::for_each(v.begin(), v.end(), std::cout << _1);
  return 0;
}
```

Вывод программы:

```
test1
test2
```

Как всё это работает?

1. Магия начинается при попытке компиляции строчки **std::for_each** в функции **main**. Чтобы сформировать третий параметр, компилятор начинает разбор выражения **std::cout << _1**. В нём осуществляется вывод в поток **std::cout** объекта типа **placeholder<1>**. Если бы где-то в стандартной библиотеке была определёна такая операция, компилятор просто сгенерировал бы соответствующий код вызова непосредственно в этом месте, а результат операции вернул бы в качестве третьего параметра для **std::for_each**. Однако такая перегрузка оператора вывода определена только в нашей программе.
2. В итоге компилятор генерирует вызов к специализации оператора вывода: **functor_out<functor_ref<std::ostream>, placeholder<1> > operator<<(std::ostream &l, placeholder<1> &r)**. В данном операторе вместо реального вызова объекта потока **std::cout** ссылка на него сохраняется в объекте типа **functor_ref<std::ostream>**. И тот передаётся в качестве левого операнда в новый сконструированный возвращаемый объект **functor_out<functor_ref<std::ostream>, placeholder<1> >**. А в качестве правого операнда туда передаётся просто ссылка на заглушку **_1**.

3. Этот функтор, сохранив в себе всю информацию о будущей операции, и передаётся третьим параметром в **std::for_each**.
4. Внутри метода **std::for_each**, где происходит вызов этого функтора с элементом вектора, происходит следующее. Управление передаётся в оператор вызова **functor_out<…>::operator() (std::string &t1)**. Там для выполнения выражения **_1(t1) << _r(t1)** поочерёдно вызываются левый и правый функторы. Левый возвращает ссылку на сохранённый объект **std::cout**. А правый возвращает ссылку на переданный в него первый параметр **(t1)**.
5. После того, как получены левый и правый операнды, осуществляется непосредственно то, ради чего всё и затевалось – операция **std::cout << t1**.

Таким образом, прозрачно для пользователя библиотеки осуществляется как бы операция вывода элемента в поток, хотя на самом деле происходит её эмуляция. Теперь становится очевидно, что за выражением **std::cout << _1** стоит совершенно не тот код, что за выражением реального вывода элемента в поток.

В чём принципиальное отличие настоящей библиотеки функционального программирования от только что рассмотренного кода? Во-первых, это поддержка от одного до N параметров, N обычно задаётся каким-нибудь макроопределением при компиляции библиотеки. Для поддержки переменного числа параметров в каждом функторе делается не один, а N операторов вызова – для 1, 2, 3… N аргументов. Например, так:

```
template <typename T1> void operator()(T1 &t1)
{
    _1(t1) << _r(t1);
}
template <typename T1, typename T2> void operator()(T1 &t1, T2 &t2)
{
    _1(t1, t2) << _r(t1, t2);
}
template <typename T1, typename T2, typename T3> void operator()(T1 &t1, T2 &t2, T3 &t3)
{
    _1(t1, t2, t3) << _r(t1, t2, t3);
}
//...
```

Во-вторых, передача аргументов иногда удобна по ссылке (если они будут меняться сформированным выражением), иногда по константной ссылке (если они участвуют только в выражении справа), или по значению (если эти объекты настолько малы, что эффективнее их передать именно так; например, тип char). Меташаблонные процедуры в библиотеке при компиляции программы должны определить наиболее подходящий тип передачи для каждого параметра.

В-третьих, в реальной библиотеке должен быть перегружен не один оператор «<<», а все, доступные для перегрузки. И их семантика должна быть сохранена, что не всегда удаётся выполнить на 100%.

А в-четвертых, что самое важное, настоящая библиотека должна уметь правильно определять тип возвращаемого значения в эмулируемой операции. В моём простейшем примере я использовал **void** в качестве такого типа, что, несомненно, годится только для примера. Это не позволит эмулировать выражения, где участвует больше двух операндов, например, «**std::cout << _1 << std::endl**». И это для автора библиотеки представляет далеко не тривиальную задачу, так как в языке C++ нет простых способов получить тип возвращаемого значения для произвольного выражения. Только в последней версии языка такой оператор появился.

Качество реализации парадигмы функционального программирования

Кратко рассмотрим, насколько реализации этой парадигмы в библиотеках **lambda** и **phoenix** близки к идеалу. В примерах, приводимых в документации по этим библиотекам, авторы пытаются показать, что их использование практически прозрачно для пользователя. То есть, что описание кодовых блоков почти не отличается от написания обычной программы на C++. Ниже уделяется внимание именно тем моментам, где идеал расходится с реальностью, т.е. местам, которые авторами либо пропускаются, либо дополняются использованием всяческих help'еров и wrapper'ов. Примечание: эксперименты проводились в Visual Studio 2008 на 32-разрядной машине.

1. Вывод строки в поток. Реализация функциональным объектом библиотеки **lambda** (с вызовом в том же месте):

```
(std::cout << boost::lambda::_1 << std::endl)("test");            // Не скомпилируется
```

Этот пример компилироваться не будет из-за ошибки с неоднозначностью выведения шаблона «**boost::lambda::operator<<**». Интересно, что убрав последний оператор вывода «<< **std::endl**» из

приведённого кода, можно заставить пример скомпилироваться и нормально отработать. Реализация через **phoenix** компилируется и работает:

```
(std::cout << boost::phoenix::placeholders::_1 << std::endl)("test");
```

1. Использование произвольного глобального non-copyable объекта вместо **std::cout**. Для теста используем следующий объект:

```
class A
{
    A(const A&);
    A& operator=(const A&);
public:
    A() {}
    const A& operator<<(const char *p) const
    {
        std::cout << "\"A::operator<<\" has been invoked with: " << p << std::endl;
        return *this;
    }
} g_a;

// Естественно, это работает:
g_a << "test";
```

Реализация с использованием **lambda** не компилируется, так как где-то внутри кто-то пытается копировать объект **g_a**. Необходимость передавать по значению объекты, которые сами по себе не могут быть скопированы, возникает достаточно часто при работе с разными библиотеками из boost-набора. Поэтому в boost есть универсальная ref-обёртка (reference wrapper), единственная задача которой – сохранив в себе ссылку на аргумент-объект, позволить внешнему коду её свободное копирование. Но в данном случае и использование вспомогательной ref-обёртки ничего не меняет:

```
(g_a << boost::lambda::_1)("test");                                 // Не скомпилируется
(boost::ref(g_a) << boost::lambda::_1)("test");                     // Не скомпилируется

// Ещё одна параноидальная попытка:
(boost::ref(g_a) << boost::lambda::_1)(boost::lambda::make_const("test"));
                                                                    // Не скомпилируется
```

Попытка сделать то же самое на **phoenix** также не увенчалась успехом:

```
(g_a << boost::phoenix::placeholders::_1)("test");                  // Не скомпилируется
(boost::ref(g_a) << boost::phoenix::placeholders::_1)("test");      // Не скомпилируется
(boost::phoenix::val(g_a) << boost::phoenix::placeholders::_1)("test");
                                                                    // Не скомпилируется
(boost::phoenix::ref(g_a) << boost::phoenix::placeholders::_1)("test");
                                                                    // Не скомпилируется
```

Наверное, есть шанс заставить всё это скомпилироваться, но такие попытки ставят под сомнение качество реализации этой парадигмы в рассматриваемых библиотеках по второму критерию. Проще говоря, быстрее было бы написать внешнюю простейшую функцию на C++, а не на псевдоязыке, который пытается быть похожим на C++.

> **ПРИМЕЧАНИЕ**
>
> Примечание: кстати, выражение (g_a << _1)("test") успешно компилируется и выполняется с использованием тестового кода микро-библиотеки, описанной выше.

Диагностика

Вывод системы обработки ошибок компилятора предназначен, в первую очередь, разработчикам, ведущим работу на языке этого компилятора. Тривиальное утверждение, не так ли? С другой стороны, авторы библиотек, предлагающих новые парадигмы программирования или новые DSL, предлагают как бы трансляторы из DSL в основной язык (рис.1). Но, в силу объективных причин, они не в состоянии

обеспечить выдачу качественных диагностических сообщений пользователям своих DSL. Поэтому любая ошибка на этапе компиляции вырождается в монстрообразные сообщения компилятора, содержание которых имеет мало общего с тем кодом, который написал разработчик. Часто можно считать, что сообщения об ошибке просто нет – мы знаем только результат компиляции: «успешно» / «не успешно».

Я наблюдал два типа попыток авторов библиотек довести до разработчика транслированную информацию об ошибке. Во-первых, это применение статических проверок со специальными именами типов тестовых внутренних объектов. Например, некоторые статические проверки основаны на том, что при некорректной генерации кода в местах статической проверки компилятор пытается вычислить длину массива с отрицательной длиной. Автор библиотеки пытается сделать информативным название такого массива, чтобы хоть как-то передать пользователю, что именно в точке проверки не так. Выглядеть сообщение компилятора будет примерно так: «не удаётся создать массив this_is_wrong_library_using_case_xxx[-1] с отрицательной длиной». Пользователь библиотеки, разумеется, должен обратить внимание не на то, что это там за массив с отрицательной длиной, а что такое «неверный вариант xxx использования библиотеки».

Вторая попытка довести до пользователя информацию о возможной ошибке – это специальные комментарии в определённых местах кода библиотеки. Если автор библиотеки может с уверенностью сказать, что в данном месте ошибка компиляции может возникнуть только по причине определённого некорректного использования, то он оставляет там комментарий типа: « // если компилятор здесь сообщил об ошибке, то это значит, что вы неверно используете библиотеку в варианте xxx».

В целом это неплохие варианты сообщения об ошибке и иногда они помогают. Однако это «детский лепет» по сравнению с той максимально полной информацией, которую бы мог выдавать полноценный компилятор DSL-языка. Работая в рамках классического C++, это можно сравнить с разницей в сообщении «не могу скомпилировать класс A» и реальным сообщением современного C++-компилятора:

Код:

```
class A {
    A() {}
};

int main(int argc, char* argv[])
{
    A a;
    // ...
```

Вывод:

```
sample1.cpp(7) : error C2248: A::A: невозможно обратиться к private член, объявленному в классе "A"
sample1.cpp(2): см. объявление 'A::A'
sample1.cpp(1): см. объявление 'A'
```

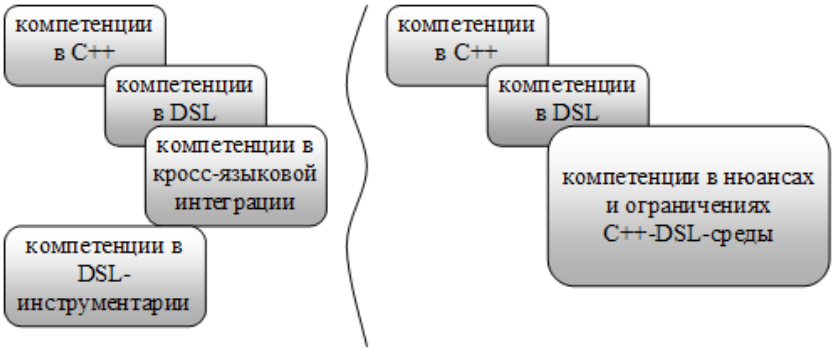
В этом примере приведена диагностика компилятора от Visual Studio 2008 при попытке скомпилировать данный код. В диагностике содержатся сведения о:

- точке в коде, где используется класс, при компиляции которого возникла ошибка;
- причине, почему такое использование класса ошибочно + ссылка на более подробные сведения в справочнике;
- точке, где определён сам класс;
- точке, где определён тот метод, который вызывает эту ошибку.

Заключение

Безусловно, язык C++ предлагает множество способов не только для написания конечных программ, но и для написания метабиблиотек, предлагающих новые парадигмы программирования или просто дополнительные DSL. И мощь таких сред в том, что разработку конечной программы можно вести сразу в нескольких парадигмах, прозрачно переключаясь между ними, не испытывая проблем с кросс-языковой совместимостью.

Однако это не даётся даром. Чем более объёмной является сторонняя парадигма или DSL, тем больше побочных эффектов (или нюансов) имеют созданные таким образом среды. И эти нюансы зачастую делают практически невозможным их прозрачное использование «как будто мы работаем в среде этого DSL». Разработчик должен не просто знать этот DSL, но и знать все эти нюансы, что резко повышает требования к его компетенции. Сложность процесса освоения таких сред на практике часто ставит вопрос о целесообразности их использования. Не говоря уже о сложности их создания.



Что разумнее?

Рис.3

Итак, на данный момент метапрограммирование в C++ имеет ряд существенных недостатков, значительно ограничивающих его широкое применение. И, как следствие, созданные с его применением DSL также весьма ограничены. Мне видятся два варианта дальнейшего пути развития языка.

Первый – это расширение языка средствами, позволяющими выполнять метапрограммирование более естественным образом. У языка должны быть средства, позволяющие свободно расширять его синтаксис. Примером может служить язык Nemerle, где можно с лёгкостью вводить через макроопределения новые операторы, а затем использовать их так же, как и первоначально встроенные в язык. Ещё одно важное для метапрограммирования свойство языка – отражение (reflection). С его помощью в программе можно получать свойства самой же программы, преобразовывать её. Метапрограммирование по сути – это написание программ, преобразующих не данные, как в обычном программировании, а другие программы или самих себя.

Второй вариант дальнейшего развития языка – это внедрение в него новых, но уже широко зарекомендовавших себя на практике, DSL и парадигм программирования. В этом случае инициатива переходит от авторов метабиблиотек к авторам самого языка.

В новой версии языка C++ авторы решили выбрать что-то среднее между этими двумя вариантами. С одной стороны, они позаботились об удобстве метапрограммирования. Например, ввели оператор decltype, с помощью которого можно получить тип, возвращаемый любым выражением языка. С другой стороны, что касается парадигмы функционального программирования, они реализовали её прямо в языке. Что, безусловно, позволит писать более «опрятные», лаконичные и эффективные программы.

Список литературы

1. Документация к библиотекам **lambda** и **phoenix** из набора **boost**. www.boost.org/doc/.