Руководство по написанию плагинов к системе Blue-Sky и правила использования smart_ptr

Blue-Sky представляет собой модульную платформу для создания, управления и обработки объектов данных. Объекты данных, а также функции и команды для их обработки предоставляются модулями, подгружаемыми на этапе загрузки ядра Blue-Sky. При загрузке модуля, у него вызывается специальная функция `bs_register_plugin`, задача которой – зарегистрировать предоставляемые модулем объекты и команды в системе (точнее, зарегистрировать указатели на функции, производящие копии этих объектов). Если указанная функция в модуле не найдена, он не регистрируется.

0. Об объектах и структуре системы

В Blue-Sky (пока) существуют две основных сущности – собс-но объекты данных (далее просто объекты) и команды. Команды – это классы, которые предназначены для выполнения каких-либо действий с объектами данных, их можно ассоциировать со сказуемым, в то время как объекты – с подлежащим. У команды есть два виртуальных метода – execute и unexecute, первый из которых выполняет прямое действие команды, а второй – обратное (если таковое возможно). Производные классы должны перегружать эти методы, чтобы вложить в команду логическое содержание.

Ранее объекты и команды были двумя разными типами, унаследованными от одного предка named_type, который содержал, счетчик ссылок, мьютекс для блокировки объекта, а также символьное имя, возвращаемое по методу bs_name() и др. самые низкоуровневые функции. В настоящее время ядро переделано таким образом, что функциональность named_type разделена между классами bs_refcounter (содержит счетчик ссылок и мьютекс) и objbase (в него перенесена функция bs_name), а класс команды унаследован от objbase. В свою очередь, objbase является наследником bs_refcounter. В результате, objbase является самым низкоуровневой (для плагинов) сущностью BlueSky, от которой должны быть унаследованы все вновь создаваемые в плагинах сущности, включая команды.

Для работы с объектами и командами должен использоваться умный указатель типа smart_ptr< T, **true** >, где второй параметр обозначает, что указываемый объект унаследован от bs_refcounter (или от objbase), т.е. содержит в себе встроенный счетчик ссылок и является типом BlueSky. Об умных указателях разговор будет далее.

Каждый класс объекта (и команды) содержит в себе статический список (std::list) умных указателей на все созданные экземпляры данного и производных типов. Т.е. список базового objbase будет содержать указатели на <u>все</u> созданные в системе объекты и команды, производный от него bs_cube будет содержать указатели на экземпляры bs_cube и производные от него и т.д. Доступ к списку экземпляров можно получить с помощью статической функции instances(), которая вставляется в тело класса с помощью макроса. Возвращаемый список константный, т.е. нет возможности добавлять или удалять из него элементы вручную (возможно в будущем будет возвращаться независимая копия списка для пущей безопасности, но смысла это не меняет). Редактирование этого списка неявно осуществляется ядром (kernel), при вызове методов create_object и release_object.

Таким образом, существует возможность быстро и удобно получать доступ ко всем созданным экземплярам данного типа и производным от него с помощью вызова T::instances(). В объектах данных, кроме того, присутствует еще один список – указателей на команды, которые с этим объектом могут работать. Доступ к этому списку можно получить с помощью функции commands(), которая, в отличие от instances(), возвращает неконстантный список с возможностью редактирования.

Экземпляр объекта BlueSky будет уничтожен автоматически в деструкторе последнего указывающего на него умного указателя. Точнее, деструктор smart_ptr вызовет виртуальную функцию dispose() класса objbase, ответственную за физическое

уничтожение объекта. Это может произойти как минимум после того, как будут очищены все статические списки, содержащие указатели на данный объект. Для этого в каждом типе BlueSky предусмотрена статическая функция free_instance, задача которой как раз аккуратно удалить все умные указатели на некоторый объект данного класса из списков ядра. Для команд надо также чистить списки указателей на данную команду в соответствующих объектах. В дополнение к free_instance существует также виртуальная функция free_this, которая, как нетрудно догадаться, просто вызывает free_instance(this) и удаляет из ядра текущий объект. Ее прелесть в ее вируальности, т.е. free_this может вызываться из базового класса (например, objbase), при этом будет вызвана «правильная» функция free_instance в производном классе.

Еще раз отмечу, что после вызова free_instance ядро уже не «поддерживает жизнь» освобожденного объекта. Он будет уничтожен, как только вызовется деструктор последнего умного указателя на этот объект у вызывающей стороны.

1. Умные указатели (УУ)

Как много в этом слове
Для сердца русского слилось

1.1 Идеология разграничения доступа между потоками в Blue-Sky

При разработке механизма организации многопоточного доступа к объектам Blue-Sky было решено воспользоваться возможностью создания константных функций-членов класса, предоставляемой языком C++. Технически, указывая слово const после объявления функции-члена, вы превращаете указатель this в const this со всеми вытекающими последствиями (становятся константными все переменные-члены, невозможен вызов неконстантных функций и т.п.). В результате этого, константная функция не может изменять никакие данные объекта, т.е. обладает только правом **чтения**. Конечно, это ограничение можно снять, используя оператор const_cast (что в большинстве случаев свидетельствует об очень плохом стиле программирования), либо предваряя объявление переменной-члена класса ключевым словом mutable. В последнем случае переменную можно будет изменять в т.ч. из константных функций. Как пишет Herb Sutter, константная функция не должна изменять видимое (т.е. public) состояние объекта, в то время как скрытые и защищенные переменные ей менять дозволяется. Т.е. решение о том, делать функцию константной или нет, можно и нужно принимать не только на основе тривиального правила о необходимости изменять хотя бы одну внутреннюю переменную класса, а руководствуясь некоторой логикой более высокого порядка. За счет этого механизму константности можно придать смысл, заставив компилятор отслеживать техническую сторону вопроса.

В нашем случае через константность реализуется поддержка блокировок объектов в многопоточной среде. Считается, что если функция-член класса объявлена константной, то она либо **не меняет состояние объекта** (включая скрытые и защищенные переменные), т.е. выполняет только **операцию чтения**, либо она **сама заботится о необходимых блокировках** в случае, если изменения требуются. Если функция объявлена неконстантной, то предполагается, что она серьезно влияет на состояние объекта и требует **его полной блокировки на время своего выполнения**. Для блокировки всего объекта в каждом экземпляре objbase предусмотрен мьютекс. Во время полной блокировки все последующие вызовы неконстантных функций-членов будут ожидать завершения текущего. Вызовы неконстантных функций не требуют никаких блокировок и выполняются напрямую, поэтому, повторюсь еще раз, если неконстантная функция все-таки меняет состояние объекта, она должна сама делать необходимые блокировки.

Как это работает? Очень просто. Все дело в том, что указатель smart_ptr< T, true > на самом деле указывает не на объект типа T, а на объект типа const T (**const не надо писать при объявлении УУ перед T, это делается автоматически!**). В результате, вызов

константного метода с помощью operator-> компилируется и работает без проблем, при попытке же вызова неконстантного метода компилятор немедленно выдаст сообщение об ошибке, главный смысл которого сводится к невозможности преобразования const T в просто T. Ошибка такого рода сразу должна **наталкивать на мысль о том, что пропущена блокировка** перед вызовом неконстантного метода. Таким образом, опасность пропустить блокировку там, где она нужна, сводится к минимуму, т.к. компилятор сам ткнет носом в каждую подобную ошибку.

Простейшая политика при разработке собственного типа BlueSky сводится к следующему: все методы, выполняющие только чтение объекта, объявлять константными (это тривиальное правило должно соблюдаться **всегда**), все остальные методы, хоть как-то изменяющие состояние объекта, объявлять без const в конце. Это очень надежно, но во многих случаях неэффективно. Зачастую достаточно **заблокировать только какую-либо часть объекта** (т.е. одну или несколько изменяемых переменных-членов) вместо блокировки всего объекта. В этом случае метод следует объявить константным, предотвратив тем самым полную блокировку объекта, и внутри функции при необходимости изменения внутренних переменных пользоваться дополнительными мьютексами и различными механизмами снятия константности – mutable-переменные, const_cast, непрозрачные указатели на реализацию (идиома pimpl) и т.п. Либо пользоваться любыми другими гениальными изобретениями, лишь бы они надежно работали (в последнее время, например, активно обсуждаются техники обеспечения многопоточного доступа без блокировок).

Вызов неконстантных функций напрямую из УУ невозможен. Для этого предусмотрено 2 механизма, применять которые надо в зависимости от контекста исполнения.

1) Быстрый однократный вызов неконстантной функции. Реализуется посредством метода lock() в УУ, который возвращает временный объект bs_locker. Он живет лишь в течение вызова функции объекта BlueSky и содержит operator ->, возвращающий неконстантный указатель на этот объект. Таким образом, полный вызов выглядит следующим образом:

```
smart_ptr< T, true > sp(xxx);
…
sp.lock()->xxx_function();
```

Поскольку в течение жизни bs_locker родительский УУ не может умереть, bs_locker **не увеличивает счетчик ссылок объекта**, а лишь выполняет блокировку. Т.е. накладные расходы минимальны: в конструкторе bs_locker выполняется блокировка объекта, в деструкторе – освобождение блокировки.

Для удобства вызова неконстантных функций из константных (с блокировкой), в интерфейсе каждого наследника objbase также предусмотрена функция lock(), которая возвращает такой же bs_locker, существующий только на время вызова функции. Пример использования:

```
my_const_func(int a) {
     //…
     lock()->my_nonconst_func();
     //…
}
```

2) Неоднократный максимально последовательный вызов неконстантных функций объекта (например, в цикле). В этом случае целесообразно делать вызовы в одной «транзакции» - заблокировать объект, сделать все вызовы, разблокировать объект. Для этого создан еще один шаблон УУ – lsmart_ptr< class SP > : public SP (сокращение от locked smart pointer), где SP – тип УУ, который будет эмулировать lsmart_ptr. В качестве SP могут использоваться только УУ, поддерживающие многопоточность (см. следующий параграф 1.2). Он обладает явным (explicit) конструктором для предотвращения случайных захватов объекта. Аналогично bs_locker, lsmart_ptr в конструкторе блокирует объект, а в деструкторе освобождает. Отличие от bs_locker в том, что lsmart_ptr является

отдельным долгоживущим УУ и наследует поведение УУ типа SP (например, работает со счетчиком ссылок).

В деструкторе lsmart_ptr происходит освобождение блокировки. Освободить блокировку можно также вызовом функции release() (которая также «сбрасывает» указатель в NULL), либо с помощью функции unlock(). Пример использования lsmart_ptr:

```
smart_ptr< T, true > sp(xxx);
//transaction begins
lsmart_ptr< T > lsp(sp);
//call nonconstanct xxx_functions
…
//transaction end
lsp.release();
```

Т.к. деструктор lsmart_ptr автоматически освобождает объект, этот УУ можно использовать в «блоке» кода:

```
smart_ptr< T, true > sp(xxx);
if(a > b) {
     //transaction begins
     lsmart_ptr< T > lsp(sp);
     //call nonconstanct xxx_functions
     …
     //transaction end
     //lsp destructor will release the lock
}
```

В целом советы при многопоточной работе с данными таковы: максимально подготовить все необходимое, произвести все возможные вычисления, не захватывая объект. Подготовить минимально возможный блок кода, вносящий изменения в состояние объекта и выполнить его в одной «транзакции». Для выполнения транзакций следует пользоваться lsmart_ptr.

1.2 Типы УУ

Иерархия УУ в Blue-Sky (пока) насчитывает 4 типа, которые различаются по типу указываемого объекта (blue-sky/generic) и по наличию поддержки многопоточности с помощью механизма константности, описанного выше.

Тип указываемого объекта кардинально влияет на реализацию подсчета ссылок на него. Все типы BlueSky унаследованы от bs_refcounter, который, по сути, является счетчиком ссылок, содержащемся прямо в объекте (это наиболее эффективный способ хранения). УУ на типы, производные от bs_refcounter, не содержат внутри себя отдельного счетчика ссылок, вместо этого обращаясь к соответствующим функциям класса bs_refcounter.

УУ на объект произвольного типа (generic type) создают отдельный счетчик ссылок для каждого объекта. УУ такого типа содержат дополнительный указатель на внешний счетчик, увеличивая и уменьшая его напрямую.

Характер работы со счетчиком ссылок определяется вторым шаблонным параметром класса smart_ptr< T, bool has_refcnt >, первый из которых определяет тип указываемого объекта. Если has_refcnt = true, считается, что указываемый объект унаследован от bs_refcounter (является типом BlueSky) и содержит в счетчик ссылок в самом объекте. Значение false обозначает произвольный тип и приводит к использованию внешнего счетчика. Во время компиляции делается попытка определить правильное значение параметра has_refcnt, т.е. является ли тип T унаследованным от bs_refcounter. Поэтому в большинстве случаев можно пользоваться укороченной записью smart_ptr< T >, значение второго параметра будет определено автоматически с помощью специального класса bs_conversion. Однако, этот механизм работает только с **полностью определенным типом T** (предварительного объявления а-ля «class T;» недостаточно). Важным моментом

является тот факт, что внутри объявления класса до закрывающей последовательности «};» тип данного класса считается неопределенным! Т.е. попытка скомпилировать код, подобный следующему, будет неудачна:

```
class A : public blue_sky::objbase {
// …
smart_ptr< A > pa_; // ошибка
smart_ptr< A, true > pa_good_; // правильно
// …
};
```

Однако, компилятор всегда сам сообщит вам о своей беспомощности, выдав строку с ошибкой, внутри которой будет содержаться подстрока «use_with_complete_types». Это верный сигнал к **явному** указанию второго булевого параметра класса smart_ptr.

Обе реализации smart_ptr< T, true > и smart_ptr< T, false > имеют поддержку блокировок в многопоточной среде с использованием механизма константности (в дальнейшем для краткости будем говорить, что они являются **многопоточными**). В этом плане между ними также есть существенная разница. Объекты BlueSky содержат мьютекс внутри себя (внутри класса bs_refcounter), в то время как smart_ptr< T, false > создает для каждого объекта внешний мьютекс так же, как и счетчик ссылок. При создании УУ на произвольный тип существует возможность не создавать этот внешний мьютекс, а использовать предоставленный пользователем (в виде параметра конструктора). За подробностями обращайтесь к исходным текстам.

Таким образом, несмотря на то, что smart_ptr< T, true > и smart_ptr< T, false > отличаются всего лишь вторым шаблонным параметром, они содержат **абсолютно разную реализацию одинакового интерфейса**. Поэтому присваивания между ними не определены и, вообще говоря, практически невозможны. Конечно, всегда существует способ эти ограничения обойти и присваивание таки осуществить (например, через функцию get(), возвращающую простой указатель), однако разработчик должен иметь очень и очень серьезные основания для совершения такой операции. Иначе это грубейшая ошибка, которая неминуемо приведет к непредсказуемому поведению программы.

Помимо описанных, в BlueSky определены еще два УУ, предназначенные для работы с объектами любых типов (generic smart pointers). Первый из них, mt_ptr, является ни чем иным, как многопоточной оберткой над простым указателем. Он не содержит счетчика ссылок и никак не влияет на указываемый объект, т.е. в этом плане ведет себя в точности как обычный указатель. По этой же причине отсутствия подсчета ссылок, mt_ptr может пользоваться для разграничения доступа к объекту только внешним мьютексом, ссылку на который нужно передать в конструктор. Преимущества mt_ptr заключаются в его простоте, компактности и высокой скорости работы (скорее всего, она почти не будет отличаться от скорости работы с простыми указателями) за счет отсутствия накладных расходов на поддержку счетчика ссылок. Этот УУ должен применяться в случаях, когда вам нужно лишь предоставить многопоточный доступ к какому-либо объекту, жизнью которого вы полностью управляете сами.

Последний на данный момент тип УУ называется st_smart_ptr. Приставка st_ расшифровывается как single threaded или «однопоточный». Из названия ясно, что это УУ на произвольный объект с подсчетом ссылок и управлением жизнью объекта, но без поддержки многопоточности. Должен применяться для автоматического удаления памяти после использования **локальных** ресурсов. Т.е. вы можете применять эти указатели для работы с локально создаваемыми данными в своих функциях, но они не должны возвращаться и быть видны вызывающей стороне.

Следующая таблица вкратце обобщает описанные выше свойства имеющихся УУ:

Таблица 1. Свойства умных указателей

Тип УУ	`smart_ptr< T, true >`	`smart_ptr< T, false >`	`mt_ptr< T >`	`st_smart_ptr< T >`
Многопоточность	+	+	+	-
Подсчет ссылок (управление жизнью объекта)	+	+	-	+
Область применения	Работа со всеми типами BlueSky	Управление и многопоточный доступ к любым типам	Эффективный указатель для многопоточного доступ к любым типам	Управление создаваемыми локальными ресурсами внутри функций

Как видно из таблицы, УУ, содержащие подсчет ссылок, содержат в названии слово smart. Префикс mt_ означает multithreaded - поддержка многопоточности, st_ - single threaded – наоборот, ее отсутствие. Такой схемы именования предполагается придерживаться и в дальнейшем, чтобы по имени типа УУ можно было легко определить его свойства.

В данном параграфе не перечислены блокирующий указатель lsmart_ptr и bs_locker, описанные в п. 1.1.

1.3 Политики приведения типов при использовании smart_ptr< T, true >

Политика приведения типов указателей должна отвечать на вопрос, что будет при попытке присвоения указателей на разные типы, т.е. A* = B*. Обычно в этом вопросе полагаются на возможности автоматического неявного приведения типов, предоставляемые компилятором. Однако при работе с УУ на объекты BlueSky детали процесса приведения можно более тонко контролировать.

Всего существует три политики – bs_static_cast, bs_semi_dynamic_cast и bs_dynamic_cast. Детальное их описание на английском языке содержится в заголовочном файле smart_ptr.h. Здесь приводится краткий перевод на русский.

1) bs_static_cast – все приведения выполняются с помощью static_cast, т.е. с максимальной скоростью. Допустим, у нас есть класс A и его наследник, класс B. Все присваивания типа smart_ptr< A > = smart_ptr< B >, smart_ptr< B > = smart_ptr< A >, smart_ptr< A > = B* и т.п. в конечном итоге сводятся к присваиванию A* = B* или B* = A*. И то, и другое в данном случае будет выполнено с помощью static_cast, если **A и B находятся на одной ветви классовой иерархии, т.е. A является наследником B или наоборот**. Проверка этого условия будет осуществлена во время компиляции и, если оно не выполнено, будет выдана ошибка. Чтобы такая проверка работала, необходимо, чтобы типы A и B были полностью определены, иначе также будет выдана ошибка компиляции. Это одно из ограничений. Другое состоит в опасности следующего кода:
```
smart_ptr< A > p_a = new A;
smart_ptr< B > p_b = p_a; //p_b указывает на экземпляр A
//неопределенное выполнение – в боль-ве случаев runtime error,
//т.к. p_b на самом деле указывает на A, а не B
if(p_b) p_b->B_specific_member();
```
2) bs_semi_dynamic_cast – приведения вверх по классовой иерархии выполняются с помощью static_cast (A* = B*), для приведений вниз используется dynamic_cast (B* = A*). Как и в предыдущем случае, используется проверка наследуемости B от A во время компиляции, что требует полного определения типов. Однако, пример выше теперь работает нормально, т.к. dynamic_cast позволяет отловить ошибочное преобразование последующей проверкой указателя на NULL. Такая политика медленнее, чем предыдущая, из-за применения dynamic_cast, но не содержит потенциальных дыр в безопасности.
   Используется по умолчанию.

3) bs_dynamic_cast – все приведения делаются с помощью dynamic cast. Эта политика может работать с неопределенными типами, т.к. никаких попыток определить конвертируемость в процессе компиляции не производится. Любые приведения легальны, об успешности можно узнать только последующей проверкой на NULL.

Текущая политика по умолчанию определяется с помощью переменной препроцессора `BS_TYPE_CAST_BEHAVIOR`, которую можно переопределить в своем коде. Кроме того существует возможность явного задания политики при создании нового экземпляра УУ или при присваивании через функцию assign. Присваивание через переопределенный operator= всегда использует глобальную политику BS_TYPE_CAST_BEHAVIOUR.

Пример:

```
smart_ptr< A > p_a = new A;
smart_ptr< B > p_b(p_a, bs_dynamic_cast());
p_b.assign(p_a, bs_static_cast());
```

Наибольший практический интерес представляет использование bs_dynamic_cast(), т.к. в этом случае вы можете работать с не определенными до конца типами, отключив проверки во время компиляции.

2. Создание плагина

2.1 Загрузка плагина

Плагин BlueSky – это динамически подключаемая библиотека (.dll в Windows и .so в Linux). Загрузка библиотек осуществляется ядром (его подсистемой plugin manager) и, если данная библиотека поддерживает протокол работы с BlueSky, она регистрируется в системе.

Регистрация состоит из двух основных этапов. Первый этап состоит из поиска и вызова функции bs_get_plugin_descriptor(), задача которой – вернуть указатель на описатель плагина (plugin_descriptor). В каждом плагине должен содержаться **один и только один** такой описатель. В данный момент в описателе плагина содержится строковая информация о версии плагина, его имени, а также о коротком и длинном описании. При разработке плагина структуру plugin_descriptor и функцию bs_get_plugin_descriptor() нужно вставлять с помощью макроса BLUE_SKY_PLUGIN_DESCRIPTOR, принимающего соответствующие параметры. Этот макрос следует помещать где-нибудь в main.cpp вашего плагина.

После успешного получения описателя, плагин регистрируется в системе и начинается вторая фаза регистрации – регистрация типов. Она выполняется с помощью вызова функции bs_register_plugin(const plugin_initializer& bs_init), которая тоже должна находиться в плагине. В качестве параметра в эту функцию подается структура plugin_initializer, которая на данный момент содержит ссылку на ядро (kernel&) и указатель на описатель данного плагина. В теле функции bs_register_plugin все обращения к функциям ядра следует выполнять **только через ссылку, содержащуюся в plugin_initializer**! Дело в том, что в момент регистрации плагинов экземпляр ядра может быть еще не создан (не завершен вызов конструктора), поэтому стандартный способ получения ссылки на ядро через синглтон (give_kernel::instance()) может приводить к зависанию системы (бесконечный цикл вызова конструктора). Однако, на данный момент с помощью пары хитростей опасность зависания ядра исключена в любом случае и вызов give_kernel::instance() из функции bs_register_plugin безопасен. Тем не менее, правилом хорошего тона будет обращение к ядру во время регистрации типов только через ссылку plugin_initializer.k_.

Регистрация типов объектов и команд выполняется через вызов функции ядра register_instance. В качестве одного из параметров в эту функцию нужно передать

описатель плагина, который удобно получить из plugin_initializer. Дело в том, что plugin_descriptor является скрытым объектом в одном из объектных файлов плагина, и попытки получить к нему доступ из других объектных файлов (например, из функции bs_register_plugin) потребует дополнительных усилий со стороны программиста. Поэтому указатель на описатель текущего загружаемого плагина был добавлен в plugin_initializer и к нему можно легко получить доступ там, где это нужно.

Вторым параметром функции kernel::register_type является **описатель типа** BlueSky – type_descriptor. Этот описатель можно получить с помощью статической функции bs_type(), которая есть в каждом объекте (добавляется с помощью макросов, см. след. параграф). Описатель типа, содержит **уникальное строковое имя типа**, его краткое и полное описание, версию, адрес структуры type_info, соответствующей данному типу (предоставляется механизмом RTTI C++), адрес статической функции bs_create_instance(), выполняющей создание экземпляров типа, адрес функции bs_create_copy(), создающей копии экземпляров данного типа, адрес функции bs_instances(), возвращающей список указателей на все созданные экземпляры этого и производных типов. Создание структуры type_descriptor осуществляется с помощью макроса BS_TYPE_DECL (объявления) и семейства макросов BS_TYPE_IMPL*. Эти макросы, в свою очередь, входят в состав соответственно макросов BLUE_SKY_OBJ_DECL/ BLUE_SKY_COM_DECL и BLUE_SKY_OBJ_IMPL/BLUE_SKY_COM_IMPL, поэтому явно их прописывать нигде не нужно.

Т.о. type_descriptor хранит всю необходимую информацию для поиска, создания экземпляров и копий типов, а также для других задач управления. Отдельное внимание стоит уделить необходимости дать типу уникальное строковое имя. Внутри ядра поиск типов везде, где только возможно, осуществляется с помощью указателей на структуры type_info. Эти указатели уникальны, т.к. type_info по требованию стандарта выделяются в статической памяти. Поиск через сравнение указателей осуществляется намного быстрее, чем по строковым именам типов. Однако, эти имена становятся необходимыми для реализации механизма сериализации (сохранения/загрузки) объектов, т.к. сохранять указатели в файл бессмысленно. Кроме того, имена типов полезны для предоставления информации о загруженных типах в понятной человеку форме. Поэтому в ядре существует еще один дублирующий способ поиска типов по именам, который должен использоваться только в случае крайней необходимости. Везде, где есть возможность вызвать bs_type() или bs_resolve_type(), нужно пользоваться функциями ядра, принимающими в качестве параметра type_descriptor&, тем самым задействуя стандартный механизм поиска типов по указателям. Ядро следит за уникальностью строковых имен и не позволит зарегистрировать два типа с одинаковыми именами.

Итак, вторая фаза регистрации плагина заключается в регистрации каждого предоставляемого плагином типа через вызов функции kernel::register_type(). Сгенерировать этот вызов можно с помощью макроса `BLUE_SKY_REGISTER_TYPE(plugin_descr, T)`. В случае успешной регистрации всех типов, bs_register_plugin() должна вернуть true. В случае каких-либо ошибок и прочих непредвиденных ситуаций нужно вернуть false, что вызовет отмену регистрации вашего плагина и он будет выгружен из памяти.

Сигнатуру функции bs_register_plugin() лучше всего получать с помощью макроса `BLUE_SKY_REGISTER_PLUGIN_FUN`. Таким образом, опять же где-то в main.cpp вашего плагина должен располагаться код, подобный следующему:

```
BLUE_SKY_REGISTER_PLUGIN_FUN
{
	bool res = BLUE_SKY_REGISTER_TYPE(*bs_init.pd_, bs_cube_t< int >);
	res &= BLUE_SKY_REGISTER_TYPE(*bs_init.pd_, bs_cube_t< float >);
	res &= BLUE_SKY_REGISTER_TYPE(*bs_init.pd_, bs_cube_t< double >);

	res &= BLUE_SKY_REGISTER_TYPE(*bs_init.pd_, bs_cube);
	res &= BLUE_SKY_REGISTER_TYPE(*bs_init.pd_, cube_command);
```

```
        return res;
}
```

2.2 Разработка объекта BlueSky

2.2.1 Маркосы для экспорта/импорта имен плагина

Для того, чтобы разработанные вами классы и функции были видны ядру и другим плагинам, необходимо их правильно экспортировать, а клиентскому коду, соответственно их правильно импортировать. Для выполнения этой задачи в BlueSky предусмотрены специальные макросы, которыми следует помечать все имена, которые вы предполагаете видимыми извне. Причем один и тот же макрос выполняет одновременно экспорт и импорт символов в зависимости от контекста использования. Кроме того, эти макросы будут автоматически распознавать все поддерживаемые компиляторы и корректно работать с ними (на данный момент это компилятор MS Visual Studio для Windows и gcc для Linux/UNIX).

Определены следующие макросы:

- BS_API_PLUGIN – осуществляет полный экспорт/импорт имен в формате C++. Должен использоваться при экспорте классов, шаблонов и других сложных имен. Импорт, скорее всего, можно осуществить только в коде, скомпилированном с использованием того же компилятора, т.к. не существует стандарта на формат имен C++, в отличие от чистого C. Также такие имена «не видны» системным функциям загрузки символов из библиотеки (`GetProcAddress/dlsym`). Тем не менее это основной макрос для экспорта/импорта, т.к. основной язык BlueSky – C++, изобилующий сложными классами, шаблонами и т.п. Поэтому с зависимостью от компилятора пока придется смириться.
- BS_C_API_PLUGIN – осуществляет экспорт/импорт имен в переносимом формате C. Может применяться в основном только для функций, при этом экспортируется только имя функции. С помощью данного макроса описываются функции bs_get_plugin_descriptor() и bs_register_plugin(). Скорее всего, разработчикам плагинов применять этот макрос не понадобится вообще.
- BS_HIDDEN_API_PLUGIN – макрос для явного предотвращения экспорта имени. Реальную работу выполняет, например, при использовании gcc, который по умолчанию экспортирует все. При создании dll под Windows наоборот, все символы по умолчанию скрыты. Последние версии gcc (особенно начиная с 4.2) могут полноценно эмулировать правила экспорта имен, принятые для dll (очевидно, что они предпочтительны, т.к. уменьшают размер библиотеки и время поиска имен за счет экспорта только необходимого).

Для ядра существуют аналогичные макросы, только без постфикса _PLUGIN. Для того, чтобы система автоматического экспорта/импорта работала корректно, каждый **плагин должен определять** с помощью командной строки компилятора **переменную препроцессора BS_EXPORTING_PLUGIN**. Ядро BlueSky определят переменную BS_EXPORTING. Исполняемые файлы, использующие плагины и ядро не должны определять ни одну из этих переменных.

Существует возможность одному плагину импортировать символы из другого. Для этого директивы include, включающие заголовочные файлы другого плагина должны быть заключены между парой следующих директив:

```
#include BS_FORCE_PLUGIN_IMPORT()
//… include your cross-plugin headers here
#include BS_STOP_PLUGIN_IMPORT()
```

Главная задача этих макросов – временно скрыть переменную BS_EXPORTING_PLUGIN и снова определить ее по окончании импорта заголовков другого плагина. При этом корректно отрабатываются множественные вложения таких макросов.

2.2.2 Оформление заголовочного файла класса

В заголовочном файле вашего класса должно содержаться как можно меньше директив #include. Включайте только самые необходимые файлы, помните, например, о существовании файла с опережающими определениями <iosfwd>, никогда не включайте в заголовочный файл вашего класса <iostream>. Если вы планируете в будущем часто изменять реализацию вашего класса или просто хотите скрыть реализацию, серьезно задумайтесь об идиоме PIMPL (Pointer To Implementation). Применение pimpl позволит вам свободно менять детали реализации без необходимости пересобирать все плагины, которые используют ваш плагин. Но цена вопроса – уменьшенное быстродействие. Настоятельно рекомендую почитать по этому вопросу соответствующий раздел в книге Herb Sutter []. Если планируете использовать в своем коде заголовочные файлы из других плагинов, не забывайте пользоваться макросами BS_FORCE_PLUGIN_IMPORT и BS_STOP_PLUGIN_IMPORT (см. предыдущий параграф).

Определившись с импортируемыми файлами, поместите далее определение вашего класса в **пространстве имен blue_sky**, открыто (public) унаследовав его от класса objbase. Для этого вы должны не забыть импортировать файл bs_object_base.h. Описав интерфейс своего класса, в самом конце, перед закрывающей фигурной скобкой поместите макрос BLUE_SKY_OBJ_DECL(T), где вместо T нужно подставить имя вашего класса. Данный макрос вставляет в интерфейс вашего класса следующие функции:

- bs_instances – статическая функция для получения списка указателей на созданные экземпляры этого и производных типов;
- bs_register_instance – статическая функция, выполняющая регистрацию экземпляра типа в вышеописанном списке;
- bs_register_this – виртуальная функция, вызывающая T::bs_register_instance(this);
- bs_free_instance – статическая функция, выполняющая очистку вышеприведенных списков, т.е. «удаление» объекта из ядра;
- bs_free_this – виртуальная функция, вызывающая bs_free_instance(this);
- bs_type – статическая функция, возвращающая type_descriptor;
- bs_resolve_type – виртуальная функция, вызывающая T::bs_type();
- bs_create_instance – фабричная функция, ответственная за создание новых экземпляров данного типа;
- bs_create_copy – фабричная функция для создания копий объекта данного типа.

Помимо этого, данный макрос добавляет объявление защищенного конструктора по умолчанию вида T(bs_type_ctor_param& param = NULL) и защищенного конструктора копий вида T(const T&). Реализацию объявленного конструктора нужно будет написать в любом случае, конструктор копий можно не реализовывать в зависимости от выбранного вами способа поддержки механизма создания копий (или ее отутствия). Все остальные пользовательские конструкторы следует объявлять только **защищенными, либо закрытыми**, т.к. создаваться и копироваться объекты должны только через ядро.

2.2.3 Оформление файла реализации

Здесь все немного сложнее. Имеет значение, какой класс вы разрабатываете – простой или шаблонный, в зависимости от этого применяются разные макросы.

В первую очередь напишите реализации всех функций вашего класса. В самом конце .cpp файла в пространстве имен blue_sky поместите макросы с реализациями bs_* функций.

I. С помощью макросов (или самостоятельно) вставить тело функции bs_create_instance(). Макросом можно реализовать стандартный способ создания объектов – с помощью оператора new. При этом используется конструктор по умолчанию. Виртуальный метод dispose() в базовом классе objbase соответствует именно этому способу создания объектов и выполняет удаление экземпляра с помощью вызова delete this. Как обычно, стандартный способ создания годится в 99% случаев.

Для того, чтобы сгенерировать тело функции bs_create_instance(), реализующей создание экземпляров с помощью new и конструктора по умолчанию используйте макрос BLUE_SKY_OBJ_STD_CREATE(T), где вместо T нужно подставить имя своего класса. Соответственно, шаблонная версия имеет на конце постфикс _T: BLUE_SKY_OBJ_STD_CREATE_T(T, templ_param_prefix). Данный макрос может быть использован только для шаблонных классов с одним шаблонным параметром и генерирует общий шаблон функции bs_create_instance без создания специализаций. Вместо templ_param_prefix нужно подставить все, что стоит перед T в определении шаблонного параметра. В большинстве простейших случаев это “class” или “typename” без кавычек. Рассмотрим более сложный пример:

```
// A.h
template< template< class > class T >
class A : public objbase {
//…
};

// A.cpp
namespace blue_sky {
// …
// templ_param_prefix = template< class > class
BLUE_SKY_OBJ_STD_CREATE_T(A, template< class > class)
}
```

BLUE_SKY_OBJ_STD_CREATE_T(A, template< class > class) генерирует шаблон:

```
template< template< class > class D > \
blue_sky::objbase* T< D >::bs_create_instance(bs_type_ctor_param param) {
  return new T< D >(param);
}
```

Специализации bs_create_instance для конкретных типов будут созданы автоматически при создании описателей типа для специализаций вашего класса. Эти описатели (type_descriptor) создаются макросом BLUE_SKY_OBJ_IMPL_T, о котором речь пойдет далее. Поэтому важно, чтобы вышеупомянутые макросы находились в одном .cpp файле с реализацией вашего шаблонного класса, причем BLUE_SKY_OBJ_STD_CREATE_T должен стоять перед BLUE_SKY_OBJ_IMPL_T. Об еще одном способе определения шаблонных классов для BlueSky читайте в следующем параграфе.

Помимо макроса, создающего общий шаблон функции bs_create_instance, существует макрос для создания конкретных специализаций этой функции: `BLUE_SKY_TYPE_STD_INSTANTIATE_T(T, type)`, где вместо type нужно подставить конкретный тип, для которого будет создана специализация, например:

```
BLUE_SKY_TYPE_STD_INSTANTIATE_T(A, int)
```

Развернется в:

```
template< > BS_API_PLUGIN
blue_sky::objbase* T< type >::bs_create_instance(bs_type_ctor_param param) {
  return new T< int >(param);
}
```

Если вы как разработчик планируете использовать какой-то другой способ создания/удаления экземпляров вашего класса, вам следует вручную написать тело функции bs_create_instance, а также переопределить функцию dispose(), ответственную за удаление данного экземпляра, так, чтобы она соответствовала выбранному способу создания объектов. Макросы при этом использовать, конечно же, не нужно.

II. Если ваш класс поддерживает копирование экземпляров, определить тело функции bs_create_copy() опять же, с помощью макросов, либо вручную. Стандартным способом является создание копии с помощью new и вызов конструктора копий вашего класса. Поэтому необходимо написать тело конструктора копий. Аналогично созданию экземпляров, существует 3 макроса для генерации тела bs_create_copy: BLUE_SKY_TYPE_STD_COPY для обычных классов, BLUE_SKY_TYPE_STD_COPY_T и BLUE_SKY_TYPE_STD_INSTANTIATE_COPY_T для шаблонных классов. Смысл последних двух макросов аналогичен пункту I.

Можно определить свой способ создания копий, вручную написав функцию bs_create_copy вместо использования макроса.

Естественно, если поддержки копирования нет, ничего из вышеперечисленного делать не нужно.

III. Поместите макрос BLUE_SKY_OBJ_IMPL для обычного класса или BLUE_SKY_OBJ_IMPL_T для шаблонного класса, указав все необходимые параметры. Учтите, что для шаблонного класса макрос BLUE_SKY_OBJ_IMPL_T следует написать для каждой специализации шаблона, например:

```
BLUE_SKY_OBJ_IMPL_T(bs_cube_t< int >, objbase, "int_bs_cube_t", "Short test int_bs_cube_t description", "")
BLUE_SKY_OBJ_IMPL_T(bs_cube_t< float >, objbase, "float_bs_cube_t", "Short test int_bs_cube_t description", "")
// etc
```

Существуют укороченные версии макросов BLUE_SKY_OBJ_IMPL_SHORT и BLUE_SKY_OBJ_IMPL_T_SHORT, которые вместо строкового имени типа подставляют его C++ тип, переданный в качестве первого параметра. Т.е. для `bs_cube_t< int >` строковый тип будет “`bs_cube_t< int >`”. Однако, пользоваться этими макросами в реальных библиотеках не рекомендуется, вместо этого постарайтесь сами написать осмысленное имя типа.

Если вы не предполагаете поддержку копирования экземпляров вашего класса, то следует воспользоваться макросом BLUE_SKY_OBJ_IMPL_NOCOPY для обычных классов и, соответственно BLUE_SKY_OBJ_IMPL_NOCOPY_T для шаблонных. В этом случае писать реализацию конструктора копий не нужно. Существуют версии этих макросов с постфиксом _SHORT, который имеют тот же смысл, что и описанные выше.

Подытожив все вышесказанное, приведем маленький пример из cpp-файла шаблонного класса bs_cube_t:

```
namespace blue_sky {
	// body of bs_create_instance
	BLUE_SKY_TYPE_STD_CREATE_T(bs_cube_t, class);

	// body of bs_create_copy
	BLUE_SKY_TYPE_STD_COPY_T(bs_cube_t, class);

	//specializations
	BLUE_SKY_OBJ_IMPL_T_SHORT(bs_cube_t< int >, objbase, "Short test int_bs_cube_t description")
	BLUE_SKY_OBJ_IMPL_T_SHORT(bs_cube_t< float >, objbase, "Short test float_bs_cube_t description")
	BLUE_SKY_OBJ_IMPL_T_SHORT(bs_cube_t< double >, objbase, "Short test double_bs_cube_t description")
}
```

2.2.4 Замечания относительно создания шаблонных объектов BlueSky

Все, что было сказано в п. 2.2.3 относительно шаблонных классов описывает первый и главный способ регистрации подобных типов в BlueSky. На самом деле нужно понимать, что регистрируется не абстрактный шаблон, а его конкретные реализации, генерируемые с помощью макросов BLUE_SKY_OBJ_IMPL_T. Помимо всего прочего, данный макрос содержит в конце явное указание для компилятора сгенерировать специализацию шаблонного класса для данного типа. Поэтому, например, для gcc важно, чтобы блок макросов находился в самом конце файла с реализацией, т.к. к моменту специализации макросу BLUE_SKY_OBJ_IMPL_T необходимо «видеть» все функции класса, в противном случае есть риск, что часть из них будет отсутствовать в получившейся библиотеке и впоследствии не будут найдены.

Грубо говоря, этот способ описания шаблонных классов предполагает, что вы будете регистрировать только конкретные, заранее вами заданные специализации. Специализации для других типов пользователь создать не сможет. Также одна из особенностей – перенос реализации класса в cpp-файл, тогда как в заголовочном фале находится лишь объявление класса. Подобной модель описания шаблонов более естественна для компилятора MS Visual Studio, gcc наоборот, зачастую для корректной

работы требует наличия реализации в одном заголовочном фале с объявлением или прямо в теле класса. Однако, за счет ограничения количества поддерживаемых специализаций, описанный способ полностью переносим, и все перечисленные выше макросы правильно работают и в Windows, и в Linux (при соблюдении всех оговоренных условий).

Еще одна важная деталь – при применении этого способа необходимо использовать макрос BS_API_PLUGIN при объявлении вашего шаблонного класса. Это как минимум имеет значение для компилятора M$ и заставляет его искать все специализации любых функций вашего класса в предоставленной библиотеке. Если требуемый символ не найден, будет выдано сообщение об ошибке (сборщиком) вместо попыток сгенерировать отсутствующую в библиотеке специализацию здесь и сейчас.

Однако, существует другой, «классический» способ описания шаблонных классов. Вышеприведенная модель хороша всем, кроме ограничения количества доступных специализаций разработчиком плагина. Иногда нужно разработать класс, который предположительно будет использоваться с большим количеством типов и при этом должен создаваться и учитываться через BlueSky

3. Создание команды

Немногим отличается от создания модуля.

В заголовочном файле:

1) включить заголовочный файл bs_command.h, содержащий объявление базового класса команды;

2) открыто унаследовать свой класс от класса blue_sky::command;

3) добавить в тело объявления класса макрос BLUE_SKY_COM_DECL(T), где вместо T подставить имя команды;

4) объявить **закрытые (private) или защищенные** (если планируете наследование от вашего класса) конструкторы;

5) если используете какой-нибудь экзотический способ создания своих объектов (например, это просто статические переменные и их не нужно уничтожать), отличный от выделения в куче через new, то переопределить виртуальную функцию `void dispose() const`;

6) написать весь остальной интерфейс.

В файле реализации:

1) поместить макрос BLUE_SKY_COM_IMPL(T, obj, base_of_T), вместо T подставить имя команды, вместо base_of_T – имя класса-родителя, вместо obj – имя соответствующего объекта;

2) если используете стандартную стратегию создания/уничтожения объектов, поместить макрос BLUE_SKY_COM_STD_CREATE(T). В противном случае самостоятельно написать тело функции create_T() (вместо T подставить имя класса);

Примечание: Макросы из пунктов 1 и 2 нужно размещать в пространстве имен blue-sky.

3) в теле каждого конструктора прописать макросы INSTLIST_PUSH и INSTLIST_PUSH_COM2OBJ(obj) – последний добавляет команду в статический список команд объекта obj.

В каждом модуле необходимо также написать и проэкспортировать функцию bs_register_plugin, которая должна регистрировать все функции создания объектов и команд из п. 2. Пример:

```
_LIBAPI_PLUGIN_ void bs_register_plugin()
{
      kernel& k = give_kernel::Instance();
      k.register_object(bs_cube::type(), create_bs_cube);
```

```
      k.register_command(cube_command::type(), create_cube_command);
}
```