

**Компилятор полного стандарта
языка C++
как ядро систем разработки
программного обеспечения
(сборник статей компании «Интерстрон»)**

Приложение к журналу «КомпьюЛог» № 3 '2000

Журнал зарегистрирован Министерством печати
и массовой информации России
Свидетельство о регистрации № 0110310

**Главный редактор сборника
А.Г. Сергеев**

Компания «Интерстрон»:
Генеральный директор
Антон Георгиевич Сергеев
Телефон: (095) 977-6021
Факс: (095) 976-7572
E-mail: info@mlc.ru
Сеть: <http://www.mlc.ru>

© Компания «Интерстрон», 2000
© Издательство «КомпьюЛог», 2000

В предлагаемом сборнике компания «Интерстрон» рассказывает о своем компиляторе переднего плана языка программирования Си++, в полном объеме реализующем действующий международный стандарт ISO/IEC 14882:1998.

Особые возможности компилятора позволяют создавать на его основе специализированные программные комплексы - системы анализа, верификации и сертификации программ, комплексы высокоуровневой отладки и тестирования программ, полностью русифицированные системы программирования, в том числе для ответственных применений, и т.д.

О компании

В 1998 году под названием «МедиаЛингва: Компиляторы» при содействии компании «МедиаЛингва» объединились две группы специалистов – из НИВЦ МГУ и из отдела компиляторов фирмы «Фитон» (программные инструменты для однокристальных микроконтроллеров). Два года спустя процесс получил логическое развитие, и весной 2000 года на этой основе была образована самостоятельная компания «Интерстрон».

Компания занимается разработкой и маркетингом продуктов и технологий высокого уровня сложности, ориентированных прежде всего на ответственных пользователей. На сегодняшний день центр тяжести нашей деятельности находится в области инструментальных средств разработки ПО на языке Си++.

Компания стремится постоянно быть в курсе того, какие продукты и технологии нужны в будущем, и учитывать это в своей текущей деятельности. Отметим, что мы особенно заинтересованы в сотрудничестве, затрагивающем следующие перспективные направления:

- Технология исполнения программ на виртуальной машине Си++.
- Полные компиляторы Си++ для различных программно-аппаратных архитектур.
- Сертификация компилятора Си++ на соответствие требованиям МО России.
- Технология использования Си++ в качестве внутреннего языка программирования для встраивания в ПО третьих фирм («Scripting C++»).
- Технология использования Си++ в качестве языка программирования для Интернет («Internet C++»).
- Технология для визуального проектирования, анализа и разработки ПО, основанная на системе обозначений UML.
- Инструментальные средства разработки ПО для встроенных систем управления, в том числе для ответственных применений.
- ОС реального времени для встроенных систем управления с жесткими временными требованиями.
- Компиляторы для Embedded C++, Java, Ada95 и других языков программирования.

В мае 2000 года Центр независимой комплексной экспертизы и сертификации систем и технологий (ЦНКЭС) проверил эффективность системы качества, разработанной и действующей в компании «Интерстрон». Многолетние традиции аккуратного системного подхода к созданию программных продуктов, сложившиеся в нашем коллективе, получили теперь официальное признание. Заключение ЦНКЭС по результатам проверки удостоверяет, в частности, наличие «условий, необходимых для обеспечения выполнения государственного оборонного заказа».

Контактная информация

Генеральный директор:

Антон Георгиевич Сергеев

Дополнительная информация
и консультации:

info@mlc.ru

Сеть:

http://www.mlc.ru

Телефон:

(095) 977-6021

Факс:

(095) 976-7572

Введение

И достоинства, и недостатки языка Си++ общеизвестны, но все же реально в настоящее время это практически единственное средство, позволяющее решать промышленные задачи проектирования, разработки и сопровождения больших программных комплексов. Системы на Си++ будут создаваться и работать еще долгие годы, и значит, еще долго будет сохраняться потребность в разнообразных инструментах, поддерживающих разработку новых и сопровождение существующих программ.

Возможности инструмента, связанного с обработкой исходных текстов на языке программирования, определяет в первую очередь компонент, отвечающий за начальный этап обработки, на который возложен анализ (лексический, синтаксический, семантический) исходного текста.

Компания «Интерстрон» разработала самостоятельный продукт, компилятор переднего плана языка программирования Си++, который выполняет упомянутый анализ начального этапа и фиксирует его результаты для дальнейшего использования, создавая так называемое промежуточное представление.

Как будет показано далее, использование компилятора переднего плана и промежуточного представления в качестве основы для построения инструментальных систем, усиленное тем обстоятельством, что компилятор переднего плана, являясь собственной разработкой, доступен для практически любых усовершенствований, – все это способствует решению самых разных проблем, связанных с применением Си++.

К таким проблемам относятся, например:

- взаимодействие пользователя со средой разработки – язык и качество диагностических сообщений;
- понимание, анализ существующих программных комплексов (с целью подготовить модификацию, добавить новый компонент, чтобы использовать в качестве прототипа и т.п.);
- проверка корректности выполненных изменений;
- перенос программ на новую платформу;
- сертификация программ на соответствие специфическим требованиям и т.д.

В определенном смысле сюда же относятся и обучение, подготовка новых программистов Си++.

Статьи, представленные в этом сборнике, рассказывают о нашем компиляторе Си++, о возможностях и преимуществах, которые дает его применение. Статьи могут изучаться независимо друг от друга, так как практически не содержат взаимных ссылок.

Назначение и основные особенности компилятора

Евгений Александрович ЗУЕВ

Ведущий специалист отдела разработки компиляторов

В настоящей статье после вступительных слов о Си++ как об основном языке промышленного программирования, за которыми следует краткая характеристика важнейших задач, стоящих в сфере разработки и анализа больших программных систем, рассмотрен подробнее компилятор переднего плана Си++, играющий роль ядра различных инструментальных средств.

Язык программирования Си++

Основой реализации любой программной системы являются инструментальные средства и прежде всего языки и системы программирования. Несмотря на большое количество разнообразных языков и систем программирования, всевозможных систем автоматизированного построения программ, можно уверенно говорить о безусловном преобладании сегодня языка программирования Си++, как основного средства разработки промышленных программных систем.

Среди основных причин такого положения можно указать следующие.

Во-первых, Си++ объективно является в настоящее время наиболее мощным и развитым языком программирования. Он вобрал в себя все достижения науки и практики программирования, полученные за последние десять-пятнадцать лет. В нем получили отражение принципы строгой типизации и абстрактных типов данных, объектно-ориентированный подход, механизмы программирования с защитой от ошибок и обобщенного программирования и т.д. Во-вторых, этот язык возник не на пустом месте – он основан на популярнейшем языке Си, включает его как подмножество и тем самым обеспечивает совместимость с огромным количеством программ, уже написанных на Си.

Язык Си++ специально ориентирован на создание больших программных систем. Имеющиеся в нем средства раздельной компиляции и (с некоторыми оговорками) модульности дают возможность организовывать долговременную разработку силами больших коллективов программистов. Си++ допускает создание повторно используемых компонент, что особенно существенно для долгоживущих и развивающихся систем. Далее, этот язык дает возможность создания высокоэффективных программ, что особенно важно для разработок специального назначения, в частности, систем реального времени.

Наконец, существенной причиной доминирования Си++ служит наличие международного стандарта на этот язык (ISO/IEC 14882). Наличие авторитетного документа, единообразно интерпретирующего синтаксис и семантику языка, послужило мощным стимулом к его дальнейшему распространению и утверждению в качестве наиболее предпочтительного инструмента программирования.

Сказанное не должно восприниматься как апологетика Си++. Хорошо известны многие недостатки этого языка, связанные как с архаичными чертами, сохранившимися от языка-предшественника, так и с попытками совместить в нем максимальное количество новых возможностей, часто плохо сочетающихся друг с другом.

Мы не собираемся проводить анализ и сравнение Си++ с другими языками. Для нас достоянием является очевидный факт преобладания Си++, несмотря на острое соперничество со стороны других серьезных и мощных языков, как, например, Ada или Java. Безусловное доминирование Си++ не рассматривается нами как повод для ликования или, наоборот, уныния. Основной вывод, который наша компания делает из этого обстоятельства, – сохраняется острая необходимость в разнообразных инструментальных системах, ориентированных на язык Си++ и основанных на нем. Эта необходимость уже проявилась на Западе, и мы уверены, что в самое ближайшее время потребность в таких системах будет осознана и в России.

Разработка и анализ больших программных систем

Кратко рассмотрим наиболее типичные задачи, связанные с обработкой программных текстов.

1. **Компиляция** (в узком смысле – как получение исполняемого кода) остается основной функцией систем программирования. При этом особое значение придается таким задачам, как межъязыковое связывание (конструирование программ, компоненты которых написаны на различных языках программирования) и межмодульная и глобальная оптимизация получаемого кода.

2. Имеется целый комплекс задач, связанных с пониманием программ человеком. Несмотря на существенные усилия, предпринятые для повышения читабельности программ, анализ больших программных текстов, приобретая очень важное значение в современной индустрии программного обеспечения (ПО), остается одной из самых трудоемких и тяжелых задач. В связи с этим за последние годы был предложен ряд методик, призванных отобразить различные характеристики программ (структуру, связи между подпрограммами и модулями, информационные потоки, потоки управления и т.п.) в форме, облегчающей их восприятие человеком. Такие методики, как правило, являются обратимыми, то есть могут использоваться как для анализа существующих программ, так и для проектирования нового ПО. С некоторой долей огрубления указанный комплекс задач можно назвать **визуализацией**.

3. Относительно известной, но остающейся принципиально важной является задача **верификации** программ. Под этим понимается деятельность, имеющая своей целью как выявление тонких ошибок (наподобие случаев использования потенциально опасных свойств языка или присутствия сомнительных с точки зрения эффективности фрагментов программ), так и проверку соблюдения формальных стандартов и ограничений.

4. Сохраняет свою актуальность задача **статического анализа** программ. С момента появления первых исследований, связанных с формальной оценкой программ по различным параметрам, было предложено много различных метрик и систем метрик. Во многих случаях статический анализ является необходимым элементом жизненного цикла разработки ПО и потому нуждается в соответствующей программной поддержке.

5. В последнее время, прежде всего в связи с успехом языка Java, вновь возник интерес к **интерпретационной схеме** исполнения программ. Существенно возросшая производительность вычислительных систем поставила интерпретацию в ряд практических подходов для многих реальных применений, не требующих предельной эффективности. Для некоторых ЯП, например, для **Си++** интерпретационная схема исполнения, не отменяя очевидных достоинств непосредственного выполнения, потенциально может дать ряд преимуществ. Основной выгодой интерпретационного подхода является **повышенный диагностический сервис** периода исполнения, недостижимый в полном объеме для традиционного исполнения на процессоре. Тем самым становится возможным адекватно выявлять (и, следовательно, преодолевать) множество типичных для Си++ динамических ошибок (некорректная работа с указателями, утечки памяти при динамическом управлении, неинициализированные переменные и т.д.). Можно сказать, что интерпретация позволяет превратить Си++ из мощного, но ненадежного языка в мощный и надежный.

Модель исполнения, разработанная для Java, основана на концепции виртуальной машины. Для повышения уровня и возможностей отладки, а также для целей повышения надежности при исполнении программ на Си++ вполне практичным и актуальным становится создание интерпретирующих и отладочных систем на основе **виртуальной машины Си++**.

Решение перечисленных задач базируется на совокупности алгоритмов лексического, синтаксического и семантического анализа исходных программ, то есть на алгоритмах, составляющих существо традиционного понимания компиляции. Часть компилятора, выполняющую функции анализа, часто называют компилятором переднего плана. Информационное взаимодействие между таким компилятором и другими компонентами системы осуществляется по-

средством информационных структур (основные из которых – дерево программы и таблица символов), которые в данном случае перестают быть внутренними структурами компилятора и логически выносятся вовне его, образуя промежуточное представление программы. Далее мы рассмотрим это подробнее.

Компилятор переднего плана языка Си++

Итак, с точки зрения анализа исходных программ, центральной частью (ядром) инструментальных систем и средств, ориентированных на Си++, следует считать **компилятор переднего плана Си++**. Необходимо специально отметить отличия такой схемы от традиционной. Эти отличия показаны на рис.1 и 2.



Рис.1 Традиционный компилятор



Рис.2 Компилятор переднего плана

Традиционный компилятор схематически изображен на рис. 1. Такой компилятор, воспринимая на входе исходный текст программы или ее части, проводит лексический, синтаксический и семантический анализ этого текста и, в случае отсутствия в нем ошибок, порождает соответствующий объектный код в формате конкретной программно-аппаратной платформы. В процессе обработки программы компилятор формирует ряд внутренних структур, в которых хранится временная информация о программе (дерево программы, таблицы имен и т.п.). После завершения компиляции эти структуры удаляются.

В отличие от традиционной схемы, компилятор переднего плана, разработанный нашей компанией (рис.2), выполнив полный анализ исходной единицы трансляции, на выходе формирует так называемое **промежуточное представление** (ПП) программы. Существенно, что ПП образуется из структур, которые компилятор создает и наполняет в процессе обработки ис-

ходной программы. Иными словами, компилятор переднего плана выводит наружу, делает своим интерфейсом те структуры, которые обычно были скрыты внутри компилятора и которые в совокупности содержат полную информацию об исходной программе.

Название «компилятор переднего плана» (front end compiler) отражает тот факт, что он реализует первый этап обработки исходных программ, связанный с особенностями входного языка программирования и слабо зависящий от целевой платформы. Компонента системы программирования, реализующая генерацию кода для конкретного процессора, иногда называется «компилятором заднего плана» (back end compiler).

Поясним преимущества реализованной схемы.

1. Разнообразие задач, связанных с разработкой и анализом программ на Си++, было показано выше. Как видно, собственно генерация кода из исходных текстов является не единственной и во многих случаях даже не самой важной задачей. С другой стороны, все перечисленные задачи требуют возможно более полной информации о программах. Промежуточное представление как раз и предоставляет такую информацию, так как в процессе работы компилятор переднего плана производит детальный и глубокий анализ исходных текстов, выявляя все его семантические особенности, включая скрытую (не заданную явно в исходной программе) семантику.

Благодаря своему устройству промежуточное представление, генерируемое компилятором переднего плана, может служить универсальным интерфейсом для всевозможных инструментальных средств. Схема использования компилятора переднего плана иллюстрируется рис.3.



Рис.3 Компилятор переднего плана как ядро систем программирования

Как видно, генератор кода выступает таким же клиентом промежуточного представления, что и другие компоненты.

2. Предлагаемая схема с использованием компилятора переднего плана обеспечивает заметные преимущества и для случая традиционной компиляции с генерацией объектного кода. Речь идет о кросс-платформной и вообще о многоплатформной компиляции (см. также рис.3). В

силу независимости от целевой платформы, компилятор переднего плана может одинаковым образом выполнять начальный этап компиляции программ для различных платформ. Совмещая его с генераторами кода, ориентированными на различные программно-аппаратные платформы, можно получать традиционные компиляторы, порождающие код той или иной платформы.

Промежуточное представление

Несколько подробнее остановимся на структуре промежуточного представления (ПП). В свете вышеизложенных соображений понятно, что промежуточное представление должно удовлетворять нескольким важным требованиям. Во-первых, компоненты ПП должны быть определены в терминах, эквивалентных или близких понятиям входного языка, то есть Си++. Во-вторых, структура ПП должна прямо соответствовать структуре исходной программы: элементы ПП и их взаимосвязи должны повторять характер отношений между сущностями программы. В-третьих, ПП должно быть устроено таким образом, чтобы обеспечивать эффективный доступ к его элементам и в целом удобную и естественную навигацию.

Промежуточное представление, формируемое обсуждаемым компилятором переднего плана, полностью удовлетворяет сформулированным требованиям. Структура промежуточного представления показывалась на рис.2.

Три основные компоненты ПП соответствуют трем категориям сущностей языка Си++.

Дерево программы содержит образ общей структуры исходного текста на Си++, включая блоки программы (пространства имен, функции и составные операторы), заключенные в них объявления и операторы, а также выражения.

Таблица символов содержит полную информацию обо всех именованных сущностях, объявленных в программе: классах, функциях, переменных, шаблонах, типах и т.д. Для каждой сущности таблица хранит большое число семантических атрибутов, в совокупности полностью характеризующих эту сущность. Таблица символов имеет вид иерархии локальных таблиц, каждая из которых соответствует определенной области действия исходной программы (классу, функции, блоку, пространству имен и т.д.).

Заметим, что все компоненты ПП (узлы дерева и элементы таблиц символов) содержат атрибуты привязки сущности к месту ее задания в исходном тексте.

Наконец, третьим компонентом ПП является **таблица типов**. Наличие этого компонента обусловлено наличием в языке Си++ развитой системы типов, возможностью задания сколь угодно сложных пользовательских типов и разнообразными правилами преобразований типов. Таблица типов устроена таким образом, что один и тот же тип хранится в ней в единственном экземпляре, независимо от количества его упоминаний в исходной программе. Это обеспечивает высокую эффективность наиболее распространенных операций доступа к таблице типов (поиск типа, сравнение типов и т.п.).

Все алгоритмы, работающие с промежуточным представлением, реализованы в виде отдельной, независимой от компилятора, библиотеки классов и функций (см. также рис.3). Заметим, что в этой схеме сам компилятор переднего плана является одним из равноправных клиентов библиотеки доступа к ПП.

Отличительные характеристики компилятора переднего плана компании «Интерстрон»

В заключение кратко перечислим некоторые существенные характеристики представляемого компилятора переднего плана.

1. Компилятор реализует **международный стандарт** языка Си++ ISO/IEC 14882 практически в полном объеме. Степень соответствия стандарту соответствует наиболее известным зарубежным реализациям Си++ (Microsoft, IBM), а в некоторых аспектах превосходит их.

2. Компилятор разработан как **мобильная программа**. С одной стороны, его исходный текст, написанный на языках ANSI C и C++, допускает компиляцию и сборку такими известными

ми инструментами, как IBM Visual Age C++, Microsoft Visual C++, GNU C++. С другой стороны, работоспособность компилятора проверена в различных программно-аппаратных средах, в частности, на персональных компьютерах PC IBM под управлением ОС Windows NT, OS/2, Windows 9X, FreeBSD, на рабочих станциях SPARC под управлением ОС Solaris, а также на Sun 3/60 под управлением SunOS.

3. Компилятор является **полностью локализованной программой**. Это означает, во-первых, что все диагностические сообщения компилятора выводятся на русском языке (при этом имеется возможность вывода англоязычных эквивалентов сообщений). Во-вторых, документация по компилятору и промежуточному представлению написана по-русски. В-третьих, компилятор обеспечивает возможность использовать при программировании русскоязычные идентификаторы и русскоязычные эквиваленты служебных слов Си++. Последняя возможность является некоторым отступлением от стандарта языка (расширением стандарта), но представляется весьма важной для отечественных разработчиков.

4. Компилятор является полностью **оригинальной** программой, созданной без использования каких-либо разработок третьих компаний или аналогичных свободно-доступных систем (типа компилятора GNU).

Если принять во внимание еще и качество технической поддержки, получаемой нашими пользователями, то не будет большим преувеличением сказать, что применение компилятора Си++ компании «Интерстрон» открывает перед разработчиками возможность создавать на его основе системы с практически неограниченной степенью соответствия требованиям заказчика.

Показатели качества: технология разработки и тестирования компилятора

Евгений Александрович ЗУЕВ

Ведущий специалист отдела разработки компиляторов

Введение

1. Язык Си++ представляет собой чрезвычайно сложный объект как для изучения, так и для реализации. Это касается практически всех сторон и аспектов этого языка

- Это **большой** язык. Его эталонное описание (международный стандарт ISO/IEC 14882) содержит около 750 страниц насыщенного текста.

- Такая характеристика, как **сложность** языка, является относительно неформальной, однако есть основания утверждать, что объем синтаксиса Си++ и его семантическая насыщенность превосходят аналогичные характеристики любого другого известного языка программирования.

- Это **не** слишком **удачно спроектированный** язык. С одной стороны, вынужденный учет многих устаревших и откровенно неудачных свойств языка-предшественника и, с другой стороны, стремление видеть в языке возможности, отражающие современные достижения науки и практики программирования, в результате привели к нагромождению разнородных, избыточных и неортогональных свойств.

В качестве частичного объяснения такой ситуации можно высказать следующее соображение. Сложность языка вызвана, прежде всего, сложностью решаемых задач. Требования, предъявляемые к программному обеспечению (ПО), часто имеют предельно жесткий характер по быстродействию, надежности и степени адекватности решаемым задачам. Инструменты создания программных систем, а язык программирования является основным таким инструментом, не могут поэтому не быть сложными.

2. Компилятор такого языка неизбежно будет представлять собой, во-первых, весьма объемную и, во-вторых, логически чрезвычайно сложную программу. Поэтому совершенно естественными и неизбежными являются специальные усилия, направленные на достижение приемлемого качества компилятора. Это касается, по крайней мере, следующих аспектов:

- Надежность компилятора (отсутствие программных сбоев)
- Поддержка всех свойств и возможностей входного языка
- Правильность интерпретации статической семантики языка
- Адекватность сформированного результата компиляции динамической семантике языка

Начиная с первых шагов проектирования и разработки компилятора, особое внимание было обращено на обеспечение достаточного уровня указанных характеристик. Основным средством для этого стал систематический и комплексный подход к **тестированию компилятора**. Однако безусловно необходимой предпосылкой для успеха столь масштабного проекта является вообще четкая организация работ, иными словами, **технология разработки**. Данная статья освещает некоторые меры, предпринятые для достижения приемлемого качества создаваемого продукта.

Технология разработки

Будучи не в состоянии затронуть в рамках одного сообщения все аспекты жизненного цикла ПО, связанные с технологией, ограничимся здесь упоминанием о двух средствах, отчетливо показавших свою полезность при создании компилятора.

Среди принципов, на которых строит свою работу компания, следует особо отметить такие, как **документирование** деятельности и **формализация** рабочих процессов.

В качестве среды, обеспечивающей как текущее повседневное взаимодействие сотрудников, так и накопление информации долгосрочного характера, уже на протяжении ряда лет используется Lotus Notes – система коллективной работы с документами.

Текущий рабочий процесс собственно программирования организован с применением продукта Team Connection фирмы IBM. Этот продукт позволяет автоматизировать управление команд-

ной разработкой, отслеживать версии создаваемого продукта, хранить все наработки всех участников в общей базе данных, контролировать доступ к ним.

На основании реального опыта в коллективе утвердилось мнение, что формализованный подход действительно поддерживает повышение производительности труда, объективно экономит силы и время, снижая количество корректирующих действий. Понятие «технологическая дисциплина» привычно сотрудникам нашей компании и наполнено для них практическим смыслом.

В мае 2000 года Центр независимой комплексной экспертизы и сертификации систем и технологий (ЦНКЭС) проверил эффективность системы качества, разработанной и действующей в компании «Интерстрон». Наше традиционное внимание к технологии создания программных продуктов получило теперь и официальное признание. Заключение ЦНКЭС по результатам проверки удостоверяет, в частности, наличие «условий, необходимых для обеспечения выполнения государственного оборонного заказа».

Тестирование компилятора

К основным стратегическим и проектным решениям, связанным с тестированием, можно отнести следующие.

Тестовый набор

В качестве основного инструмента тестирования принят **тестовый набор**, в котором около 7 тысяч тестовых программ, в совокупности покрывающих все свойства языка Си++, согласно его стандарту. Подробнее о тестовом наборе и научно-методических основах его создания будет сказано ниже.

Организационные меры

Разработка тестов и собственно тестирование рассматриваются нами как деятельность, по своей природе существенно отличная от процесса разработки и отладки самого компилятора. Кроме того, мы считаем обоснованным методологическое требование, согласно которому тестирование не должно проводиться самими разработчиками. Поэтому на протяжении всего цикла разработки вопросы тестирования занимался отдельный, организационно **независимый коллектив**.

Среди других организационных решений следует упомянуть **регулярность** тестирования. В процессе разработки и отладки программы ее текст подвергается различным модификациям, которые, будучи направлены на улучшение, могут, помимо этого, привести к внесению дополнительных ошибок. При этом риск внесения ошибок в процессе разработки возрастает по мере возрастания объема текста, увеличения числа логических связей в программе и ее общей сложности. Поэтому принципиальное значение имеет поддержание достигнутого качества и предотвращение деградации компилятора, когда уже разработанные и оттестированные компоненты оказываются не работоспособными в результате появления новых ошибок. Поэтому было принято безусловное требование, согласно которому любое изменение исходного текста компилятора сопровождается его регулярным тестированием. Такое тестирование, как правило, проводится ежедневно (в частичном объеме) и еженедельно (в полном объеме тестового набора).

Полнота тестовых проверок

Тот факт, что компилятор смог откомпилировать очередной тест, то есть сгенерировал свой выходной результат - промежуточное представление (ПП) – еще недостаточен, чтобы считать тест успешно выполненным. Для каждого откомпилированного теста необходимо удостовериться, что сгенерированное ПП статически и динамически корректно.

Для статической проверки была разработана специальная программа-**верификатор**, которая проводит формальный анализ сгенерированного ПП на предмет внутренней непротиворечивости (взаимной согласованности ссылок, наличия необходимой информации и ее правдоподобности и т.д.) и на предмет соответствия описанию промежуточного представления.

Для динамической проверки была предпринята опережающая разработка **виртуальной машины Си++**, предназначенной для непосредственного исполнения (интерпретации) промежуточного представления. В итоге успешными считаются тесты, прошедшие компиляцию, верификацию сгенерированного ПП, а затем исполненные на виртуальной машине.

Так же, как и тестовый набор, верификатор и виртуальная машина создавались отдельным коллективом разработчиков. «Интерфейсом» этой группы с разработчиками компилятора являлись только текст Стандарта Си++ и описание структуры промежуточного представления.

Автоматизация процесса тестирования

Тестирование компилятора на столь большом тестовом наборе, естественно, не может осуществляться вручную. Поэтому для целей автоматизации таких работ был разработан специальный **монитор** тестирования, который обеспечивает ряд удобных средств для организации различных видов и режимов тестирования.

Тестовый монитор обеспечивает следующие возможности:

- Автоматический прогон тестов как в полном объеме, так и по отдельным разделам тестового набора, а также по специальному заданию.
- Различные режимы тестирования («только компиляция», «компиляция и верификация», полный цикл «компиляция-верификация-исполнение»).
- Формирование отчетов по тестированию различной степени подробности, а также ведение истории тестирования.
- Автоматическое сравнение последних результатов тестирования с полученными на предыдущих прогонах, с выдачей соответствующих отчетов.

Тестовый монитор можно использовать для запуска самых разных программ с интерфейсом командной строки. На практике, помимо собственно тестирования, этот монитор был использован для сравнительных прогонов нашего компилятора и компиляторов других фирм.

Тестирование на массиве реальных программных текстов

Практика показала чрезвычайную полезность и в то же время недостаточность тестового набора для создания полноценного компилятора. Дело в том, что тестовый набор ориентирован на проверку соответствия стандарту, но не предназначен для тестирования других важных характеристик компилятора, в частности, устойчивости (способности восстановления работы в случае ошибок в исходных программах), надежности в «краевых» случаях (предельные размеры исходного текста и отдельных конструкций языка), быстродействия и т.п. Кроме того, никакой тестовый набор не в состоянии охватить всех возможных сочетаний свойств и возможностей языка.

Поэтому наряду с тестовым набором нами используются реальные программы на Си++. В результате анализа доступных программ было принято решение в качестве «эталонной» программы основываться на исходных текстах **стандартной библиотеки Си++**, которую также реализует наша компания. Во-первых, в ней используется практически весь спектр возможностей языка (механизм классов, шаблоны в самых сложных вариантах, нетривиальные случаи преобразования типов, исключительные ситуации и многое другое). Во-вторых, эта библиотека весьма велика по объему: около 100 тысяч строк исходного текста. Кроме того, нам представляется, что совместная разработка (когда компилятор тестируется на библиотеке, а библиотека проверяется компилятором и виртуальной машиной) приводит к повышению совокупного качества всех этих систем.

Отметим, что в свою очередь, реализация стандартной библиотеки привела к разработке дополнительного тестового набора для ее проверки, и в настоящее время тестовый набор для стандартной библиотеки насчитывает более 500 тестов.

Разработка тестового набора

Научные исследования, связанные с тестированием компиляторов, в настоящее время ведутся достаточно активно. В частности, в Московском государственном университете коллектив под руководством проф. В.А.Сухомлина в течение ряда лет изучает проблематику тестирования конформности компиляторов стандартам входных языков (аттестационного тестирования). Одним из конкретных результатов этих исследований является тестовый набор, который использован для тестирования нашего компилятора Си++.

Тесты, входящие в тестовый набор, разрабатывались согласно следующим **базовым принципам**:

- набор аттестационных тестов должен покрывать весь стандарт языка;
- тесты должны быть независимыми в том смысле, что результаты исполнения любого из них не должны влиять на условия или результаты исполнения любого другого;

- один тест должен проверять только одну языковую конструкцию;
- тест не должен проверять что-либо за рамками стандарта языка.

Опыт практического тестирования показывает, что для создания аттестационных наборов, отвечающих указанным принципам, достаточным является использование следующих тестов:

- «правильные» тесты. Тест считается успешным, если он откомпилирован, и полученный код выполнен без ошибок (самопроверка теста подтвердила его правильное исполнение);
- «неправильные» тесты. Тест считается успешным, если все ошибки, которые заложены в нем, определены на этапе компиляции;
- дополнительно, тесты межмодульных связей между различными единицами трансляции.

Другие требования к тестам, в частности, таковы:

- каждый тест должен быть по возможности прост и краток;
- кроме специальных случаев, «неправильный» тест должен содержать только одну ошибку;
- все тесты, кроме «неправильных», должны быть самопроверяющимися;
- тесты, группы тестов и комплекты тестов должны быть самодокументированными и иметь ссылки на соответствующие им разделы исходных спецификаций.

В конечном итоге удалось получить хорошо отлаженный тестовый набор, обладающий устойчивой структурой и достаточно легко сопровождаемый, несмотря на большое число (около 7 тысяч) входящих в его состав программных тестов.

В заключение заметим, что тестовый набор для проверки компиляторов Си++ представляет собой «штучный» и весьма дорогой продукт. Так, на Западе насчитывается всего 3-4 компании, поставляющие подобные наборы, и их цена доходит до 40 тыс. долл.

Инструментальная поддержка сертификации и тестирования ПО

Имеется достаточно обоснованная, как мы надеемся, уверенность, что компилятор, оттестированный согласно методике, принятой в нашей компании, приобретает некоторое новое качество. Речь идет о том, что такой компилятор, помимо выполнения своих непосредственных задач, может использоваться как средство сертификации программ на Си++.

С помощью компилятора могут быть выполнены:

- проверка на соответствие Стандарту Си++;
- проверка на наличие в программе фрагментов, которые представляют собой формально допустимые, но потенциально опасные конструкции (например, произвольные преобразования типов), либо ограничивают переносимость программы на другие платформы;
- проверка (после минимальных доработок) на соответствие отраслевым или внутрифирменным стандартам программирования.

Однако в целом возможный спектр проверок, производимых компилятором, неизбежно ограничен сферой статической семантики Си++; проблемы и особенности, проявляющиеся только на стадии выполнения программы, за редкими исключениями, принципиально не могут быть выявлены при ее обработке компилятором.

Как будет показано в других материалах этого сборника, наличие такого компонента, как виртуальная машина, делает возможным решение гораздо более широкого круга задач, связанных с сертификацией ПО по результатам динамического анализа программ.

Построение программных систем на основе компилятора переднего плана

Антон Георгиевич СЕРГЕЕВ

Генеральный директор

В этой статье показаны предпосылки к построению программных систем на основе компилятора переднего плана и рассказывается о компонентах такой системы.

Предпосылки применения компилятора для построения программных систем

Компилятор переднего плана языка Си++

Продукт, о котором идет основная речь в этой статье, это компилятор **переднего плана Си++**. Название «компилятор переднего плана» (front end compiler) отражает тот факт, что его назначением является первый этап обработки исходных программ, связанный с особенностями входного языка программирования и слабо зависящий от целевой программно-аппаратной платформы.

Традиционный компилятор, воспринимая на входе исходный текст программы или ее части, проводит лексический, синтаксический и семантический анализ этого текста и, в случае отсутствия в нем ошибок, порождает соответствующий объектный код в формате конкретной программно-аппаратной платформы. В процессе обработки компилятор формирует ряд внутренних структур, в которых временно хранится информация о программе (дерево программы, таблицы имен и т.п.), а после завершения компиляции эти структуры удаляются.

Компилятор переднего плана также проводит анализ исходного текста, но в отличие от традиционного компилятора выводит наружу, делает своим интерфейсом внутренние структуры, которые обычно оставались скрыты. Полученное на выходе так называемое **промежуточное представление** (ПП) содержит в совокупности полную информацию об исходной программе и не привязано ни к какой конкретной платформе.

Таким образом, промежуточное представление, генерируемое компилятором переднего плана, может служить универсальным интерфейсом для всевозможных инструментальных систем и средств, ориентированных на язык Си++. Любой компонент, и в том числе, генератор кода для конкретной платформы, выступает таким же клиентом промежуточного представления, что и другие компоненты.

О структуре промежуточного представления

Промежуточное представление должно удовлетворять нескольким важным требованиям.

Во-первых, компоненты ПП должны быть определены в терминах, эквивалентных или близких понятиям входного языка, то есть Си++.

Во-вторых, структура ПП должна прямо соответствовать структуре исходной программы: элементы ПП и их взаимосвязи должны повторять характер отношений между сущностями программы.

В-третьих, ПП должно быть устроено таким образом, чтобы обеспечивать эффективный доступ к его элементам и в целом удобную и естественную навигацию.

Три основные компоненты ПП соответствуют трем категориям сущностей языка Си++.

Дерево программы содержит образ общей структуры исходного текста на Си++, включая блоки программы (пространства имен, функции и составные операторы), заключенные в них объявления и операторы, а также выражения.

Таблица символов содержит полную информацию обо всех именованных сущностях, объявленных в программе: классов, функций, переменных, шаблонов, типов и т.д. Для каждой сущности таблица хранит семантические атрибуты, в совокупности полностью характеризующих эту сущность. Таблица символов имеет вид иерархии локальных таблиц, каждая из которых соответствует определенной области действия исходной программы (классу, функции, блоку, пространству имен и т.д.).

Таблица типов – присутствие этого компонента обусловлено наличием в языке Си++ развитой системы типов, возможностью задания сколь угодно сложных пользовательских типов и разнообразными правилами преобразований типов. Таблица типов устроена таким образом, что один и тот же тип хранится в ней в единственном экземпляре, независимо от количества его упоминаний в исход-

ной программе. Это обеспечивает высокую эффективность наиболее распространенных операций доступа к таблице типов (поиск, сравнение типов и т.п.).

Заметим, что все компоненты ПП содержат атрибуты привязки сущности к месту ее задания в исходном тексте.

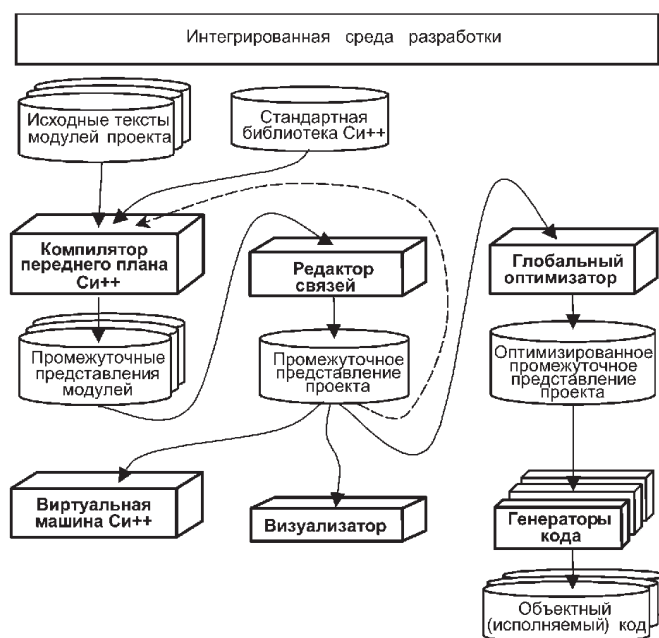
Возможности внесения изменений

Следует сделать замечание, что в нашем случае ядро системы – компилятор переднего плана – это полностью оригинальная разработка с доступными исходными текстами, поэтому для доработок с целью решения таких проблем, как, например, локализация системы или предоставление детальной диагностики, открыты практически неограниченные возможности.

Программные системы на основе компилятора переднего плана

После изложенного выше обоснованной и естественной выглядит задача построения системы, интегрирующей на основе единого ядра – компилятора переднего плана – различные средства, направленные на поддержку проектирования и анализа Си++-программ. Создание таких систем можно считать одним из магистральных направлений в деятельности нашей компании.

Принципиальная компонентная схема такой системы может выглядеть следующим образом:



Рассмотрим теперь компоненты системы подробнее.

Интеллектуальный редактор связей

Особенности построения компилятора переднего плана, а также характер промежуточного представления, формируемого компилятором, позволяет принципиально повысить возможности редактора связей по сравнению с традиционными компоновщиками.

В традиционной схеме генерация объектного кода происходит **перед** связыванием модулей. Наш подход предусматривает предварительное связывание промежуточных представлений отдельных единиц трансляции Си++ в единое промежуточное представление всей программы. Генерация объектного кода производится **после** этого, по объединенному ПП.

Сохранение в объединенном ПП полной информации об исходной программе дает ряд значительных преимуществ перед традиционной схемой:

- поскольку при прохождении через генератор кода часть исходной информации неизбежно теряется, традиционный редактор связей не в состоянии полностью (в соответствии с

требованиями стандарта) диагностировать ошибки, связанные с межмодульными связями. Часть таких ошибок он даже не в состоянии выявить;

- располагая информацией обо всех единицах трансляции, усовершенствованный линкер может устранять сущности, повторяющиеся в разных модулях (для программ на Си++ эта проблема «разрастания кода», связанная с наличием inline-функций и механизмом настроек шаблонов, весьма характерна; однако устранить дублирование кода без доступа к информации из исходных текстов крайне трудно);
- наличие в распоряжении компоновщика информации о всей программе (объединенного ПП) дает возможность глобальной оптимизации кода. Очевидно, что традиционный подход к компоновке практически исключает такую возможность.

Заметим также, что известная проблема, связанная с поддержкой механизма межмодульного экспортирования шаблонов Си++, полностью решается усовершенствованным линкером «на ходу».

Наконец, в усовершенствованной схеме компоновки при обработке каждого следующего модуля проекта компилятор располагает информацией об уже скомпилированных модулях, в частности, о настроенных шаблонах. Учет уже выполненных настроек дает существенное ускорение при компиляции очередного модуля, тем более что механизм шаблонов является существенной и активно используемой на практике частью языка (на них построена вся стандартная библиотека).

Линейка унифицированных генераторов кода

Компилятор переднего плана «единообразно» выполняет начальный этап компиляции программ независимо от целевой платформы. Совмещая его с генераторами кода, ориентированными на разные программно-аппаратные платформы, можно получать традиционные компиляторы, порождающие код для той или иной платформы.

Учитывая практическую мобильность компилятора переднего плана, следует ставить вопрос о разработке целой **линейки генераторов кода** для различных платформ. Эта работа может проводиться как нашей компанией, так и в сотрудничестве с другими разработчиками (например, можно предложить **конструктор генераторов кода**, который бы максимально облегчил стороннему производителю создание конкретного генератора кода на основе нашего компилятора).

Отметим также, что на основе объединенного промежуточного представления всего проекта, помимо общей глобальной оптимизации на начальном этапе, выполненной линкером, может быть также выполнена дополнительная глобальная оптимизация генератором кода, учитывающая особенности конкретной платформы.

Виртуальная машина Си ++

Концептуально виртуальная машина Си++ (VM) представляет собой идеализированную модель некоторого замкнутого самодостаточного исполнителя (компьютера), архитектура которого адекватна динамической семантике языка Си++. В нашем случае это означает, что виртуальная машина позволяет непосредственно исполнять сгенерированное компилятором переднего плана промежуточное представление.

Требования, накладываемые на архитектуру и реализацию виртуальной машины, несколько противоречивы и включают четыре основные группы:

- **адекватность языку Си++.** VM должна включать элементы (структуры данных и алгоритмы), прямо реализующие важнейшие особенности семантики Си++;
- **поддержка отладки.** VM должна предоставлять полный сервис для отладочного исполнения в соответствии с современными представлениями об отладке, а также обеспечивать высокий уровень контроля исполнения программ;
- **интерфейс с операционной системой.** VM должна обеспечивать для прикладных программ доступ ко всем ресурсам программного интерфейса базовой операционной системы;
- **эффективность.** VM должна быть разработана так, чтобы не допустить существенного падения производительности программ, выполняемых под ее управлением по сравнению с непосредственным выполнением откомпилированного кода в обычной среде.

В рамках рассматриваемого проекта виртуальная машина создавалась прежде всего как средство тестирования ПП, порождаемого компилятором, однако опыт реального использования показал ряд ее существенных преимуществ в самых разных приложениях.

Во-первых, виртуальная машина может обеспечить максимально полную **диагностику ошибок** периода выполнения, которую зачастую в принципе невозможно получить иным способом. В частности, VM позволяет контролировать:

- инициализированность объектов;
- обращение за границу области памяти, отведенной под объект;
- наличие указателей на уничтоженные объекты (как автоматические, так и в динамической памяти);
- наличие объектов в динамической памяти, на которые нет указателей;
- корректность значений указателей; в частности, обращение по абсолютному физическому адресу;
- допустимость значений фактических аргументов вызова функций стандартной библиотеки;
- допустимость (корректность) преобразований типов.

Во-вторых, виртуальная машина дает возможность исполнения программ в некоторой **модельной среде**, исключающей потенциальное влияние исполняемой программы на системное окружение. Тем самым, VM пригодна в качестве «испытательного стенда» для тестирования, верификации и сертификации программ.

В-третьих, виртуальная машина может быть использована напрямую, как платформно-независимая **система выполнения программ** в тех случаях, когда относительное падение эффективности вполне допустимо.

Визуализатор программ на Си++

Методика наглядного представления программ, базирующаяся на графическом языке UML, в настоящее время является мировым стандартом де-факто. Две отдельные статьи настоящего сборника посвящены проблемам отображения программ на Си++ в виде UML-диаграмм.

Инструментальная поддержка тестирования и сертификации ПО

Имеется достаточно обоснованная, как мы надеемся, уверенность, что компилятор, разработанный и оттестированный согласно методике, принятой в нашей компании, приобретает некоторое новое качество. Речь идет о том, что на основе такого компилятора могут быть созданы средства **сертификации программ** на Си++.

Вопросы статического и динамического анализа программ на Си++ рассматриваются в данном сборнике также в статьях, посвященных проблемам показателей качества и локализации.

Среда разработки

Для поддержки всех этапов разработки ПО - от подготовки исходного текста до документирования - предназначена **интегрированная среда разработки**, которая представляет собой, с одной стороны, законченную систему и содержит все необходимые для работы средства, и, с другой стороны, - открыта для легкого добавления в нее новых инструментов.

Стандартная библиотека языка Си++ представляет собой весьма значительную (более половины общего объема) часть Стандарта языка Си++. Стандартный комплекс библиотечных ресурсов языка Си++ воплощает большое количество важных принципов и подходов, отражающих современное состояние программистской теории и практики.

Стандартная библиотека вместе с исходными текстами проекта используется для компиляции промежуточного представления. Она дает в распоряжение программиста большое количество готовых хорошо оттестированных программных компонентов общего назначения.

Однако стандартная библиотека не покрывает все основные потребности программиста в компонентах. Можно упомянуть по крайней мере три области, для которых нужны готовые **проблемно-ориентированные библиотеки**:

- поддержка хранения объектов (Persistence);
- создание графических интерфейсов;
- сетевой доступ.

На сегодняшний день эти компоненты используются в очень многих прикладных программах. Поэтому нам кажется обоснованным и разумным включать в состав инструментальных систем некоторые, сравнительно простые версии данных библиотек.

Визуализация C++ программ с помощью унифицированного языка моделирования UML

Владимир Юрьевич РОМАНОВ

Ведущий специалист

В статье описываются способы визуализации основных конструкций и понятий языка C++ с помощью графической нотации языка моделирования UML. Показывается, что существует взаимное соответствие конструкций и понятий в текстовом языке C++ и графическом языке UML. Даются пример визуализации C++ программ с помощью инструментов, входящих в состав компилятора C++, разработанного фирмой «Интерстрон».

Введение

При разработке больших систем на языке C++ по мере роста системы возникает вопрос об управляемости процессом разработки. Для того, чтобы оценить текущее состояние разрабатываемой системы, предстоящие изменения и дополнения в этой системе, становятся необходимы средства, позволяющие представить в компактном виде многие из важных свойств этой системы. Каким образом устроена иерархия классов, как связаны друг с другом подсистемы, где и как используются ключевые классы системы? Возможность легко и быстро получить ответы на эти и многие другие вопросы позволяет держать процесс разработки под постоянным контролем, предотвращая таким образом возможные осложнения.

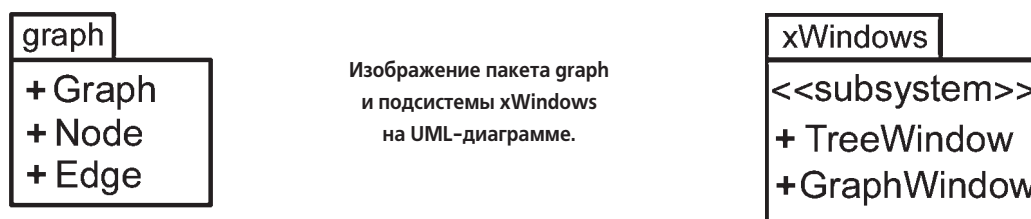
Видимо, достаточно понятно, что наиболее компактно и наглядно ответы на такие вопросы могут быть представлены в графическом виде. Перефразируя крылатое выражение «маленький чертеж стоит большой речи», можно сказать, что одна картинка может стоить тысячи строк текста программы на C++. Действительно, результирующая картинка может быть результатом анализа огромного количества строк программы, к тому же расположенных во множестве различных файлов. Если такой поиск, анализ и отображение в графическом виде необходимой информации о программе выполняется инструментальной программой, а не человеком, можно гарантировать, что при этом потери информации не произошло.

Возникает, однако, вопрос: можно ли графически адекватно представить конструкции и понятия такого сложного языка как C++? Консорциум Object Management Group создал специальную группу по анализу и проектированию, в результате многолетней работы которой был разработан язык Unified Modeling Language (UML). Этот язык был предложен в качестве стандартного языка анализа и проектирования сложных программных систем. Одна из его особенностей заключается в том, что он имеет достаточно строго определенную семантику, которая описывается с использованием метамодели языка – набора классов, выражающих его основные понятия. Тем самым, метамодель языка UML определяет стандарт на репозиторий (способ хранения модели программы) CASE-систем. В стандарте OMG описана также и графическая нотация [1,2] языка UML, с помощью которой отображаются элементы метамодели [3]. Как метамодель, так и предложенная графическая нотация позволяют представлять конструкции и понятия таких наиболее распространенных объектно-ориентированных языков программирования, как C++ и Java. Вместе с тем, в метамодель языка UML и в его графическую нотацию встроены стандартные механизмы расширения, позволяющие моделировать и визуализировать свойства других, не столь распространенных языков.

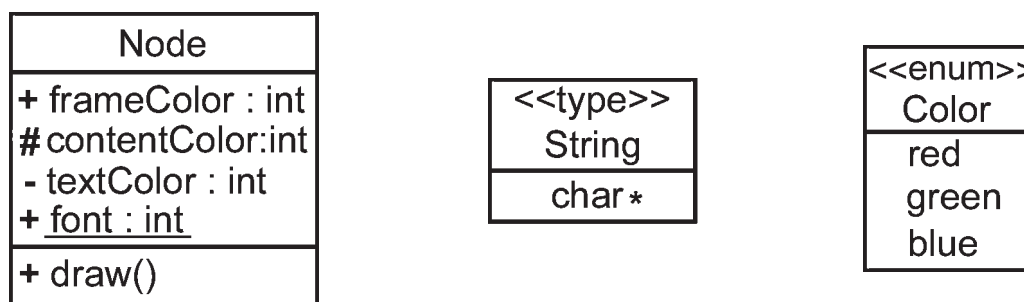
В состав системы, построенной на основе компилятора C++ фирмы «Интерстрон», среди прочих входит инструмент для визуализации программ на C++. В этом инструменте используются описанные в стандарте OMG метамодель языка UML и нотация языка UML. Далее в этой статье описывается, как основные конструкции и понятия языка C++ могут быть представлены с помощью графической нотации языка UML.

Основные элементы языка C++

Пространства имен и подсистемы. Относительно недавно в язык C++ были введены конструкции, позволяющие определять пространства имен (namespace). Эти конструкции позволяют структурировать сложную систему на множество независимых компонент, элементы которых имеют непересекающиеся имена. В метамодели языка UML пространству имен языка C++ соответствует понятие **пакета** (package). Для пакета определено также и соответствующее обозначение в UML-нотации. Другой способ структурирования систем, написанных на языке C++, – размещение файлов различных компонент в различные подкаталоги. В этом случае подкаталогу в метамодели языка UML соответствует понятие **подсистемы** (subsystem). В UML-нотации – это пакет со стереотипом <<subsystem>>. Ниже приводится пример с изображениями пакета и подсистемы:



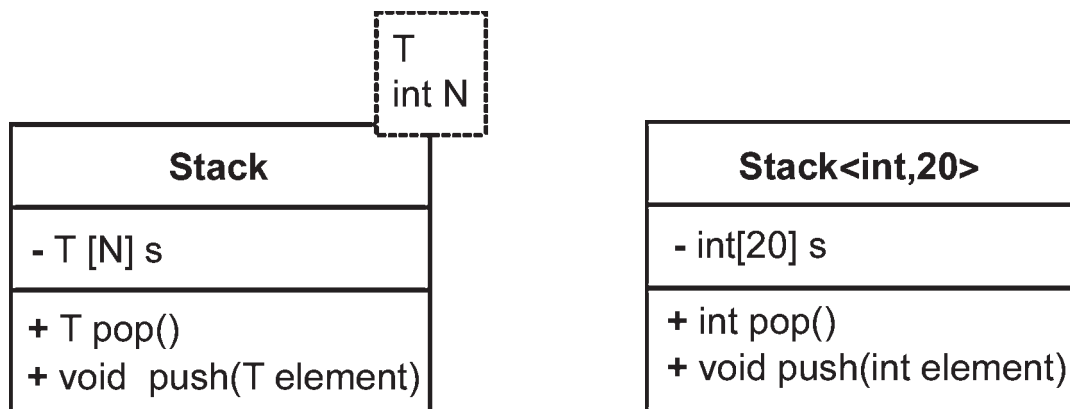
Классификаторы. В языке C++ существует ряд понятий, которые служат для классификации объектов программы. Для этих целей в язык введены классы, перечисления, типы, структуры, шаблоны. У многих из этих понятий есть ряд общих свойств. Так, например, классы, типы и перечисления могут быть использованы при описании типов переменных и параметров функций. И классы, и шаблоны могут иметь атрибуты и операции. В языке UML для таких понятий языка C++ имеется обобщающее понятие **классификатора**. В метамодели языка UML базовый класс Классификатор является предком классов, соответствующих перечисленным понятиям языка C++, а в нотации языка UML изображения различных классификаторов дополняются соответствующими стереотипами. При отсутствии стереотипа такое изображение классификатора считается изображением класса. Общим у изображений классификаторов является наличие секций, содержащих при необходимости атрибуты, операции и другую информацию, используемую в объявлении классификаторов. Ниже приводятся примеры некоторых классификаторов, которые изображают соответствующие понятия языка C++



Изображение класса Node, объявления типа String и перечисления Color на UML-диаграмме.

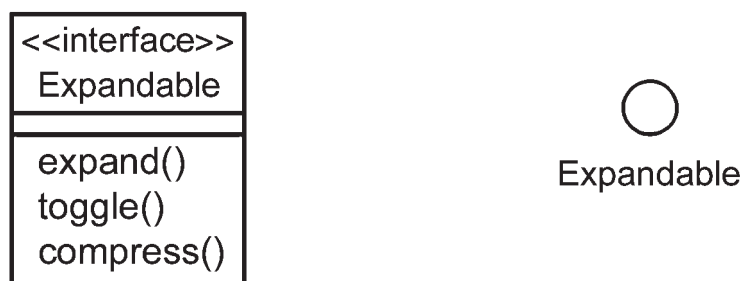
Для отображения свойств атрибутов и операций классификаторов, например, их видимости или статичности, язык UML предоставляет необходимые графические обозначения. В приведенном выше примере статические члены класса подчеркнуты, публичные помечены знаком «плюс», защищенные – знаком #, а скрытые – знаком «минус». Чистые виртуальные функции, а также классы, содержащие такие функции (абстрактные классы), имеют имена, написанные курсивом.

Шаблоны и генерируемые классы. Шаблоны языка C++ являются мощным инструментом так называемого «обобщенного» программирования. Важность шаблонов языка C++ особенно возросла после включения в стандарт языка C++ библиотеки шаблонов STL. Эта библиотека содержит реализацию универсальных структур данных и алгоритмов, существенно упрощающих и ускоряющих разработку систем на языке C++. Поскольку в метамодели языка UML параметризуемыми являются все элементы метамодели, язык UML естественным образом обеспечивает поддержку шаблонов классов и функций языка C++. Нотация UML имеет также соответствующие обозначения для шаблонов и классов, сгенерированных в результате подстановки фактических параметров в шаблоны.



Изображение шаблона Stack и класса, сгенерированного по этому шаблону, на UML-диаграмме.

Интерфейсы. Под интерфейсом в языке UML понимается класс, имеющий только чистые виртуальные функции и не имеющий атрибутов. Особое внимание к интерфейсам в языке UML обусловлено использованием интерфейсов для структурирования сложной системы на множество независимых компонент, взаимодействие с которыми осуществляется только через эти интерфейсы. Данная методология разработки систем применяется также в модели взаимодействия компонент COM фирмы Microsoft. Для изображения интерфейсов в нотации языка UML используются два типа обозначений:



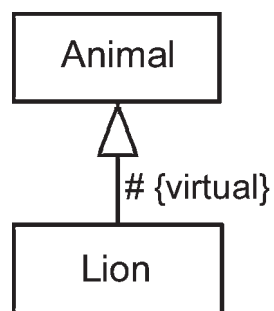
Изображение интерфейса Expandable в нормальном и в сжатом состоянии.

Отношения между элементами языка C++

Отношения, существующие между элементами языка C++, при анализе (в частности, при визуализации) программы имеют не меньшее значение, чем сами элементы языка C++. Например, иерархия наследования классов во многом определяет семантику каждого из классов иерархии. Вместе с тем, получение информации об отношениях между элементами программы – более сложная задача, нежели получение информации о самих элементах. Например, поиск потомков некоторого класса требует просмо-

тра всей системы в целом. По этой причине автоматизация такого поиска и визуализация его результатов в виде компактного графического представления дает еще более существенную помощь в понимании сложных систем, написанных на языке C++. Возможные отношения между элементами языка C++ имеют на языке UML адекватное графическое представление. Рассмотрим эти отношения.

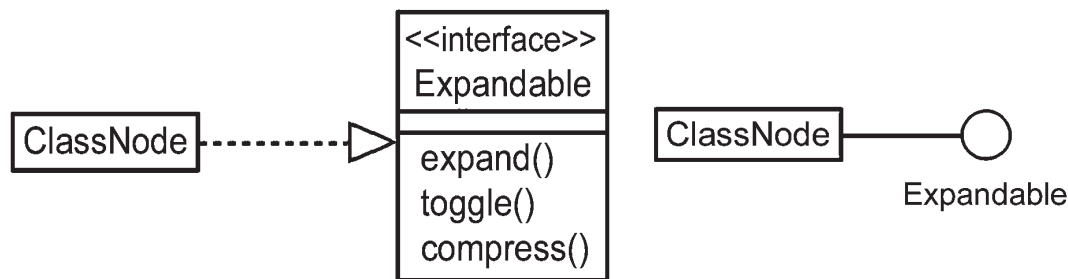
Отношение наследования. Отношение наследования в языке C++ имеет атрибуты видимости наследуемых атрибутов и операций, а также виртуальность наследования. На языке UML это отношение и его атрибуты показываются следующим образом:



Отношение наследования и его атрибуты на диаграмме UML.

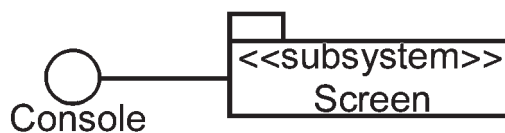
На изображенной диаграмме защищенность наследуемых атрибутов и операций показывается символом #, а виртуальность наследования – с помощью условия-ограничения (constraint) в фигурных скобках. Отношение наследования допустимо в языке C++ и между классами, генерируемыми по шаблону и, при условии, что в таком отношении шаблон является потомком, между шаблоном и классом.

Отношение реализации. Как уже упоминалось, в языке UML придается особое значение интерфейсам из-за их особой роли при структурировании программных систем. По этой же причине в язык UML введено особое изображение, показывающее отношение реализации интерфейса. Это отношение может быть показано двумя способами, в зависимости от состояния изображения интерфейса на UML-диаграмме:



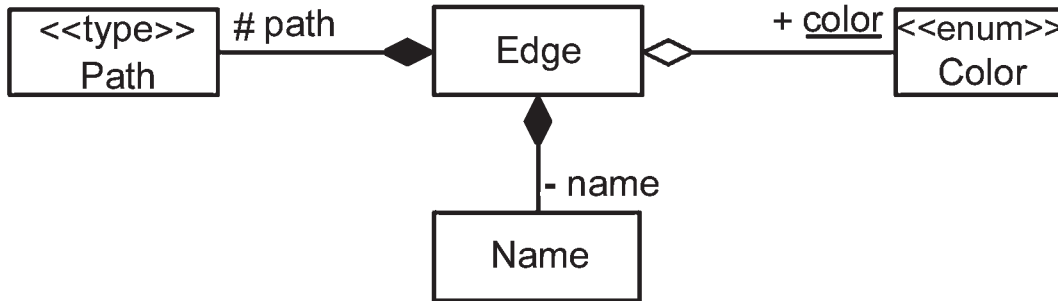
Класс ClassNode реализует интерфейс Expandable.

Если один из классов пространства имен (подсистемы) реализует какой либо внешний по отношению к нему интерфейс, то считается, что этот интерфейс реализует также и пространство имен (подсистема). Нотация языка UML позволяет показать это следующим образом:



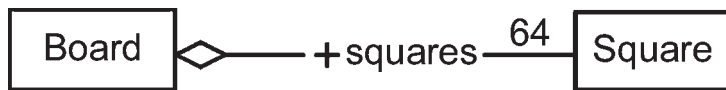
Реализация интерфейса Console подсистемой Screen.

Отношение ассоциации. Это отношение показывает включение экземпляром классификатора значения экземпляра или указателя на экземпляр другого классификатора. В изображении этого отношения на языке UML показываются количество и атрибуты включаемых экземпляров.



Включение классом Edge экземпляров: типа Path, перечисления Color и класса Name.

В приведенном выше примере черным ромбом показано включение экземпляра по значению, белым ромбом – включение указателей на экземпляр. Статичность показывается подчеркиванием, а изображение видимости совпадает с изображением видимости для атрибутов классификатора. На следующей диаграмме показано, что экземпляр класса Board включает в себя 64 экземпляра класса Square:



Изображение количества включаемых экземпляров на диаграмме UML.

Если число экземпляров может изменяться динамически без каких либо ограничений на их количество, то такая множественность экземпляров показывается с помощью символа *. С помощью ограничителя в фигурных скобках может быть указана структура из стандартной библиотеки шаблонов, которая была использована для реализации такого множественного отношения ассоциации. Например, могут быть показаны ограничители {vector}, {set} или {stack}, соответствующие одноименным структурам стандартной библиотеки шаблонов STL.

Отношения зависимости. Под этим обобщенным типом отношения понимаются отношения между элементами языка C++, имеющие самые различные причины. Этот тип отношений представляется в метамодели языка UML общим базовым классом Зависимость, а в UML-нотации типы зависимости различаются стереотипом.

Отношение доступа. Этот тип зависимости возникает между элементами языка C++, если один элемент предоставляет доступ к своему содержимому другим элементам. Например, некоторый класс может предоставить доступ к своим защищенным и скрытым членам классам-друзьям. UML-нотация позволяет это сделать с помощью отношения зависимости, имеющего стереотип <<friend>>:



Класс NodeDialog является другом класса Node и поэтому имеет доступ к переменным x и y.

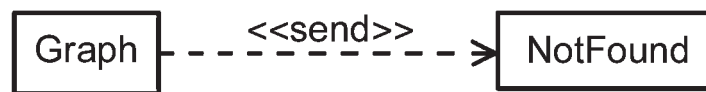
Другое отношение предоставления доступа может возникнуть, если некоторый класс получает доступ ко всем или некоторым элементам пространства имен (импортирует их). Примеры такого импорта на языке UML показаны ниже:



Импортирование шаблона `vector` из пространства имен `std`, а также импортирование всех элементов пространства имен `std`.

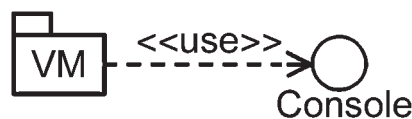
Отношение использования. Этот тип отношения зависимости возникает, если один элемент языка C++ каким-либо образом использует другой элемент. Для отношения использования существует множество стереотипов. Стереотип `<<call>>` применяется, если операция некоторого класса вызывает операцию другого класса. Если в операции класса создается экземпляр другого класса, то это может быть показано с помощью отношения зависимости со стереотипом `<<create>>`. Операция класса может использовать другой класс как параметр, как результат вызова, или как локальную переменную класса. В этих случаях применимы стереотипы `<<parameter>>`, `<<returns>>`, `<<local>>`.

Конструкция возбуждения ситуации (`throw`) языка C++ представляется с помощью отношения зависимости со стереотипом `<<send>>`.



Возбуждение ситуации `NotFound` в одной из операций класса `Graph`.

В том случае, если для отношения зависимости по смыслу возможно несколько стереотипов, может быть выбран обобщенный стереотип `<<use>>`:



Использование интерфейса `Console` в пространстве имен `VM`.

К сожалению, объем данной статьи не позволяет подробно и полностью описать способы визуализации программ на C++ с помощью представляемого инструмента. Однако, надеемся, что общее впечатление об этом читатель получил.

Ссылки.

1. Грэйди Буч, Джеймс Рамбо, Айвар Джекобсон. Язык UML. Руководство пользователя. – М.: ДМК, 2000.
2. М. Фаулер, К. Скотт. UML в кратком изложении. – М.: Мир, 1999.
3. UML 1.3. www.omg.org/docs/ad/99-06-09.zip

Автоматизированный анализ структуры сложных систем на языке C++ с помощью языков UML и SQL

Владимир Юрьевич РОМАНОВ

Ведущий специалист

В статье описан подход, согласно которому программная система, написанная на языке C++, трактуется как некоторая база данных. Для анализа структуры такой системы и выявления существующих в ней взаимосвязей предлагается использовать язык запросов SQL. Результаты поиска отображаются в виде диаграмм на языке UML. Предусматривается хранение и параметризация таких запросов как шаблонов UML-диаграмм. Задание положения шаблонов в определенном контексте системы приводит к автоматическому построению UML-диаграмм для элементов данного контекста.

Введение

При разработке сложных систем на языке C++ по мере роста объема системы возникает проблема с пониманием структуры системы и взаимосвязей между ее элементами. Такое понимание особенно затруднено, если анализ программы проводится человеком, который не является ее автором. Если разработчик программы недоступен, а объем программы достаточно велик, то дальнейшее использование, сопровождение и модификация такой программы неизбежно требует значительных усилий и времени. Такая ситуация является типичной, если разработка не сопровождалась с самого начала тщательным проектированием и документированием, а автор программы не работает в организации, использующей программу.

В такой ситуации автоматизированный анализ программной системы может существенно облегчить дальнейшую работу с ней (сопровождение, модификацию и т.п.), а зачастую такой анализ, в силу больших размеров программы, может быть произведен только автоматически. Для того, чтобы элементы системы и связи между ними были представлены компактно и наглядно, они должны отображаться в графическом виде. Для представления программ, написанных на объектно-ориентированных языках программирования, существует графическая нотация – язык UML (Unified Modeling Language) [1,2]. Этот язык предлагается в качестве стандартного языка при проектировании программ консорциумом Object Management Group (OMG)[3]. Однако этот язык может быть использован также и для анализа системы уже после того, как она разработана. Как показывается в другой статье данного сборника, конструкции языка C++ имеют адекватное представление на графическом «объектно-ориентированном» языке UML.

Язык C++ содержит значительное число «строительных элементов», из которых могут конструироваться программы: классы, перечисления, типы, шаблоны, пространства имен – это далеко не полный перечень. Еще больше количество отношений, в которые могут вступать сущности C++ друг с другом. Например, класс может использовать другой класс как своего предка, как своего друга, использовать экземпляр этого класса по ссылке или по значению, использовать другой класс как тип параметра одной из своих функций. Отметим еще раз, что для всех элементов языка C++ и для всех отношений между ними имеется адекватное представление на языке UML.

Однако, возникает проблема: как построить UML-диаграммы для такого большого объема разнотипной информации, полученной из программ на языке C++? Программа может быть представлена в виде дерева, как это делается в окружениях многих известных компиляторов. В таком дереве обычно показываются пространства имен, классы, атрибуты и их функции. Представление в виде дерева бывает полезно само по себе при анализе программы в том случае, если в этом дереве представлены все элементы программы. Однако, у многих компиляторов языка C++ это не так. Например, в системе Visual C++ не отображаются типы, перечисления, шаблоны и пространства имен. В принципе, в дереве программы можно показать также и отношения между ее элементами. Однако отображение отношений сильно теряет в наглядности, поскольку элементы программы, находящиеся в некотором отношении, в таком случае находятся в неравноправном положении. Для отображения отношений между элементами программы более подходит двумерное пространство.

Промышленные инструменты для анализа и проектирования программ, например, широко известная система Rational Rose, представляют в дереве большинство элементов, которые могут присутствовать в программах на C++ и, соответственно, на UML-диаграммах. В таком случае элементы дерева могут быть использованы для копирования их на диаграммы с помощью операции drag-and-drop. Отношения для уже скопированных на диаграмму элементов в таком случае строятся автоматически.

Такой способ построения UML-диаграмм удобен, если размеры программы на C++ не очень велики. Представим себе случай, если нам надо скопировать на диаграмму классы, которые являются потомками некоторого другого класса. В этом случае необходимо вручную проверить все классы, а затем вручную скопировать на диаграмму те из них, которые являются потомками заданного класса. Ситуация осложнится, если необходимо найти и скопировать на диаграмму, например, все классы, которые используют данный класс как тип параметра одной из своих функций. В таком случае объем информации, которую человеку необходимо просмотреть и найти в дереве программы визуально, значительно возрастает. Понятно, что такой подход к анализу утомителен и чреват ошибками. Принимать какие-либо важные решения по результатам такого «анализа» сложной системы весьма рискованно.

Приведенные примеры показывают, что для анализа больших программ на C++ необходимы инструменты, позволяющие автоматизировать как поиск необходимых элементов и отношений, так и их отображение в виде UML-диаграмм. В этом случае программу естественно рассматривать как некоторую «базу данных», к которой делаются запросы. Результатом таких запросов служит UML-диаграмма. Рассмотрим особенности взгляда на программу как на базу данных.

Первый вопрос – каким должен быть **язык запросов** к такой базе данных. Ясно, что для этого вполне подходит некоторый универсальный язык запросов к базам данных, например, SQL. Одно из достоинств применения SQL – широкая распространенность этого языка. По этой причине реализация некоторого подмножества SQL как языка запросов к такой специализированной базе данных возможна и оправдана. Вместе с тем, структура нашей «базы данных» весьма специфична, поэтому возможен также более и простой язык, отражающий особенности C++-программы. В частности, запрос к базе данных может быть специфицирован, как заполнение некоторой формы, содержащей понятия языка C++. Использование форм, как правило, требует меньшей квалификации и знания особенностей структуры базы данных. После того, как форма заполнена, по ней строится запрос на текстовом языке SQL.

Второй вопрос – **отображение результатов поиска**, выполненного в базе данных по запросу. Естественно, возможно простое решение – «выбросить» на UML-диаграмму найденные в результате поиска элементы программы. После этого UML-диаграммы автоматически будут дополнены отношениями для тех элементов программы, которые присутствуют на диаграмме. Затем пользователь вручную выполняет планировку диаграммы так, как считает это нужным. Однако такой подход имеет и ряд недостатков. Отображение на диаграмме всех связей между элементами UML-диаграммы может быть излишним. Кроме того, для некоторых типов отношений может возникнуть ситуация, когда на диаграмме «все связаны со всеми». Поэтому необходима **автоматическая фильтрация отношений** для элементов, найденных в результате поиска. Например, нас могут интересовать отношения наследования с уже добавленными к диаграмме отношениями и не интересовать отношения включения присутствующих на диаграмме классов как экземпляров добавляемых классов. Найденные элементы при добавлении к диаграмме могут быть отображены с различной степенью детализации. Например, для найденных и добавленных к диаграмме классов могут быть показаны только атрибуты, либо атрибуты и функции. Могут быть показаны только публичные атрибуты и функции добавленного к диаграмме класса. По аналогии с фильтрацией отношений добавляемых элементов, необходимо выполнить также и **автоматическую фильтрацию атрибутов элементов**.

Другой недостаток такого простого решения – утомительность последующей перепланировки диаграммы. Если диаграмма строится поэтапно, то результаты предыдущей ручной планировки могут пропасть или стать в новой ситуации бесполезными. Поэтому желательно дать возможность указать также и возможные способы **автоматической планировки** для добавляемых на диаграмму элементов. Например, для вновь добавляемых потомков класса на UML-диаграмме должно быть построено дерево, корнем которого будет базовый класс в иерархии наследования.

Как можно заметить, запрос информации о программе, фильтрация отношений и атрибутов, планировка тесно связаны между собой. Вполне естественно и удобно, если все эти операции будут описываться на едином языке. Запросы на таком языке могут представлять собой типы UML-диаграмм. Для каждого из таких запросов может быть дано уникальное имя для повторного использова-

ния. Набор именованных типов диаграмм может играть ту же роль, какую играют именованные шаблоны и стили при создании документов в текстовых редакторах.

До сих пор мы рассматривали запросы на поиск информации, которые делались вручную. Однако запрос на поиск информации в системе на C++ и построение по ней UML-диаграммы может быть выполнен автоматически, если для такого запроса может быть заранее задан **контекст**. Предположим, что некоторый тип UML-диаграмм необходимо построить для всех классов, находящихся в некотором пространстве имен. Поместим описание данного типа диаграммы в дерево программы в требуемое для этого пространство имен, как один из элементов дерева. В этом случае у анализатора будет достаточно информации для построения необходимых диаграмм для **каждого** из классов в указанном пространстве имен. Построенные в соответствии с данным шаблоном UML-диаграммы будут также помещены как элементы программы в дерево программы в тот же контекст, что и шаблон, по которому они были построены. Для того, чтобы такое было возможно, в указанном выше языке запросов должны быть средства указания контекста для запроса. Включение экземпляра такого шаблона в дерево программы автоматически определяет его контекст.

В последующих разделах описывается, как описанный выше подход к анализу сложных систем на языке C++ реализуется в окружении компилятора C++, разработанного фирмой «Интерстрон».

Промежуточное представление компилятора с языка C++ и метамодель языка UML

Важно заметить, что анализ сложной системы на языке C++ невозможен без необходимой поддержки со стороны компилятора. За свою более чем пятнадцатилетнюю историю язык включил в себя множество возможностей, которые необходимы для разработки сложных объектно-ориентированных систем. В то же время язык C++ обеспечивает преемственность для огромного объема программного кода, уже написанного на языке C. Достижение этих двух, часто противоречащих друг другу целей, обусловило сложность как самого языка, так и его реализации. Из-за этой сложности текст программы на C++ может быть разложен на свои составляющие элементы только лишь после его полной трансляции. Например, без трансляции шаблонов проводить анализ текста программы на C++ не имеет смысла. По этой причине использование препроцессоров для языка C++, извлекающих из программы лишь «избранные» конструкции, не может дать полного представления о программе.

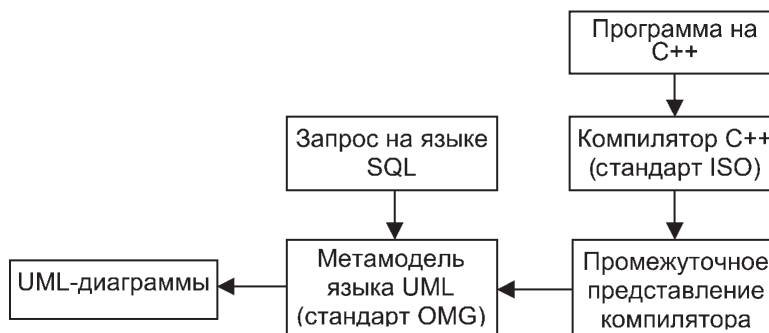
Компилятор языка C++, разработанный нами, обеспечивает полную поддержку, необходимую для анализа программ, поскольку этот компилятор соответствует международному стандарту ISO/IEC 14882 на язык C++.

Другим фактором, определяющим возможность анализа сложных систем на языке C++, является результат работы компилятора. Насколько высок уровень промежуточного представления программы, полученного в результате трансляции, чтобы по такому представлению стал возможен анализ? Принципиально важно, что промежуточное представление программы, формируемое указанным компилятором, вполне удовлетворяет и этому требованию. Инструменты, созданные фирмой на базе этого компилятора, позволяют делать адекватное восстановление (reverse engineering) программы на языке C++ по ее промежуточному представлению. При этом в промежуточном представлении сохраняется вся информация о программе, необходимая для проведения такого анализа.

Наличие промежуточного представления, адекватно представляющего исходную программу, позволяет использовать его как источник информации для построения UML-диаграмм. Вместе с тем, отметим, что стандарт на язык UML определяет не только графическую нотацию для моделирования объектно-ориентированных систем, но также перечень объектов и отношений, используемых для построения таких моделей. Совокупность таких объектов и отношений получила название **метамодели** языка UML [3]. При разработке этой метамодели предусматривалось, в частности, унифицированное представление в метамодели систем, написанных на таких широко используемых языках как Pascal, C++, Java. Поэтому информация из промежуточного представления компилятора C++ адекватно трансформируется в стандартную метамодель языка UML.

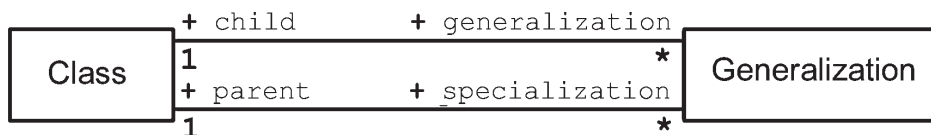
Метамодель языка UML содержит информацию, которая подготовлена для ее отображения в стандартизованном графическом виде – UML-нотации. Необходимая для отображения информация в метамодели собирается в результате однократного обхода структур промежуточного представле-

ния. В метамодели уже содержится вся информация как об элементах программы, так и их отношениях. Классы (таблицы) метамодели возможно использовать не только для визуализации программы, но также и для поиска информации по запросам на языке SQL.



Метамодель языка UML как база данных программ на C++

Традиционно язык SQL используется для поиска информации в реляционных базах данных. Однако подмножество языка SQL может быть использовано также и для работы с экземплярами классов, расположенных в оперативной памяти. На приводимом ниже рисунке показывается фрагмент метамодели языка UML, представленный с помощью нотации UML:



На приведенной выше UML-диаграмме показаны два класса с именами Class и Generalization. Класс Class представляет в метамодели классы языка C++. Класс Generalization представляет отношения наследования между классами C++. Между этими двумя классами метамодели существуют два отношения ассоциации: у каждого экземпляра класса Generalization хранятся ссылки на экземпляры класса Class с именами child и parent. Каждый экземпляр класса Class хранит ссылки на экземпляры класса Generalization, для которых он является предком (specialization) и потомком (generalization) других классов. Данные два класса можно считать таблицами, для выборки информации из которых можно использовать запросы на языке SQL. Предположим, мы хотим найти всех потомков класса с именем Piece. Для этого может быть использован следующий запрос на языке SQL:

```

select * from Class, Generalization
where Generalization.parent.name = 'Piece'
  
```

В результате такого запроса на UML-диаграмме показываются все классы – предки класса с именем Piece, сам класс Piece, а также в виде ребер графа отображаются отношения, существующие между этими классами. При этом для каждого из классов будут показаны все его атрибуты и операции.

Поскольку атрибутов и операций у класса может быть достаточно много, необходимы средства для фильтрации лишь необходимых для отображения на UML-диаграмме атрибутов и

операций. Пусть необходимо просмотреть все публичные атрибуты найденных классов. Соответствующий запрос будет выглядеть следующим образом:

```
select Class.attribute from Class, Generalization
where Generalization.parent.name = 'Piece'
filter Class.attribute.visibility = 'public'
```

Как можно заметить, в запросе появилась новая конструкция **filter**, которая в традиционном языке SQL отсутствует. Мы поместили условие *Class.attribute.visibility = 'public'* именно в эту конструкцию, и поэтому на диаграмме будут показаны даже те классы, которые не имеют публичных атрибутов. Если бы условие *Class.attribute.visibility = 'public'* было включено в конструкцию **where**, то классы, не имеющие публичных атрибутов, на диаграмме показаны не были бы. Мы не указали в конструкции **select** также и *Class.name*, поскольку секция имени у класса на UML-диаграмме показывается в любом случае.

До сих пор мы не интересовались, как будут расположены на UML-диаграмме найденные классы. По умолчанию эти классы располагаются в фиксированном месте, и пользователь сам выполняет планаризацию графа, показанного на диаграмме. Однако можно автоматизировать и этот рутинный этап анализа. Добавим к запросу конструкцию планаризации графа, построенного в результате запроса:

```
select Class.attribute from Class, Generalization
where Generalization.parent.name = 'Piece'
filter Class.attribute.visibility = 'public'
layout tree(Class.name = 'Piece', Generalization, 'down')
```

В результате такого запроса на диаграмме будет построено дерево, направленное вниз с корнем – классом Piece. Построению дерева будут использоваться те классы, которые вступают в отношении наследования (Generalization). Заметим, что планаризация графа в виде дерева – лишь один из возможных способов планаризации.

Далее. Было бы заманчиво сохранить часто используемые запросы к базе данных-метамодели UML в виде некоторой параметризованной процедуры:

```
procedure childTree(Class class, string direction = 'down')
begin
  select Class.attribute from Class, Generalization
  where Generalization.parent.name = class.name
  filter Class.attribute.visibility = 'public'
  layout tree(class, Generalization, direction)
end
```

Использование набора таких процедур существенно упрощает построение UML-диаграмм при анализе программы. Однако важно также и то, что эта процедура определяет некоторый шаблон UML-диаграммы с именем childTree. Если поместить экземпляр этого шаблона в некоторый контекст, например, пространство имен, то возможно построение диаграмм по шаблону childTree для каждого класса в этом пространстве имен. Такой же шаблон может быть помещен в контекст конкретного класса. Для удобства определение контекста может задаваться перемещением имени шаблона с помощью операции drag-and-drop в дерево проекта программы на C++.

Ссылки

1. Грэйди Буч, Джеймс Рамбо, Айвар Джекобсон. Язык UML. Руководство пользователя. – М.: ДМК, 2000.
2. М. Фаулер, К. Скотт. UML в кратком изложении. – М.: Мир, 1999.
3. UML 1.3. www.omg.org/docs/ad/99-06-09.zip

«Локализация» – от страны до рабочего места (когда все в Ваших руках)

Антон Георгиевич СЕРГЕЕВ

Генеральный директор

В статье показано, почему можно полагать, что компилятор переднего плана «Интерстрон» является средством создания рабочих мест со свойствами, нужными именно Вам.

Введение

Локализация [от латинского *localis* – местный]
– ... связанность с определенным местом
(Словарь иностранных слов)

Процесс, обозначаемый термином «локализация», обычно сводится к тому, что пользователю предоставляют для ввода / вывода дополнительно один язык, отличающийся от «English (United States)» и характерный для данной местности. Попробуем (для целей этой статьи) такое толкование расширить, сузив рамки процесса от местности вообще к частным пределам рабочего места.

Будем говорить в нашем случае, что локализация – это процесс, в результате которого не только пользовательский *интерфейс* (язык, термины, содержание, детальность, тексты сообщений), но также и выполняемые *функции* локализуемого программного обеспечения (ПО) станут в условиях конкретного рабочего места наилучшим образом соответствовать его назначению. С некоторой натяжкой, в частности, и разработка ПО с требуемыми функциями, выполненная на базе существующего программного ядра, тоже может быть названа локализацией «по функциям».

Цель этого сообщения:

- показать на примерах, какого рода задачи могут быть решены, когда в роли программного ядра выступает компилятор переднего плана;
- показать, что компанией «Интерстрон» созданы объективные предпосылки для решения множества подобных задач в соответствии с локальными требованиями заказчиков.

Особенности компилятора переднего плана и принципы построения программных систем на его основе детально обсуждались в других статьях этого сборника, поэтому здесь они не повторяются.

Примеры применений

Примеры применений разделим на две группы, в первой из которых суть локализации составляет использование русского языка, а во второй – развитие функций, соответствующих решаемой задаче.

Приложение в конце статьи содержит развернутый перечень возможных применений.

Использование русского языка

Некоторые предпосылки к применению русского языка заложены в действующем стандарте ISO. Стандарт языка Си++ допускает использование в пользовательских идентификаторах символов из достаточно широкого набора национальных алфавитов, который включает, в частности, все символы кириллицы. Кроме того, стандарт допускает использование произвольных символов в комментариях.

Введение национальных эквивалентов для служебных слов Си++ представляет собой более спорную задачу; поскольку формально это является отступлением от стандарта языка. Однако, веским доводом в пользу введения таких эквивалентов служит то обстоятельство, что двуязычная смесь английских служебных слов и разрешенных стандартом русских идентификаторов оказывается неестественной для зрительного восприятия (подробнее см. статью «Русские «плюсы»» в журнале «Мир ПК» № 4 за 1999 год или по адресу, например, <http://www.cio.ru/pcworld/1999/04/109.htm>).

Проект С11

Одна из основных проблем с применением языка Си++ – это неумение его применять. Подготовка квалифицированных программистов Си++ является поэтому важнейшей самостоятельной задачей.

Язык Си++ и так представляет собой чрезвычайно трудный объект для изучения, однако положение усугубляется тем, что известные системы программирования на Си++ рассчитаны на англоязычных пользователей. Перед начинающими программистами возникают сразу две проблемы: высокая сложность систем объектно-ориентированного программирования и необходимость знания английского языка.

Для начинающих программировать на Си++ предназначена **система С11** компании «Интерстрон», позволяющая изучать Си++, используя исключительно русский язык. Этому способствуют:

- полностью русскоязычная среда разработки,
- система помощи с интеллектуальными средствами доступа;
- учебник по языку Си++, написанный опытными преподавателями МГУ.

С11 позволяет не только читать по-русски пояснения и учебник, но и программировать на русском «диалекте» языка Си++, который полностью совместим со стандартом Си++. Система С11 не навязывает пользователю русский язык в качестве единственного или основного. При желании можно программировать с использованием только английских обозначений (впрочем, допускается свободное чередование английских и русских имен и служебных слов). Кроме того, система С11 имеет средства автоматического приведения исходных текстов программ к международной системе обозначений.

Виртуальная машина – интерпретатор русскоязычной программы

Существует немалое количество приложений, в которых надежность исполнения играет гораздо более важную роль, нежели эффективность.

Для такого рода приложений можно предложить схему исполнения в некоторой специальной программной среде, которую обеспечивает **виртуальная машина**, интерпретирующая программу на языке Си++. Виртуальная машина является стандартным компонентом окружения для компилятора переднего плана нашей компании. Вклад в повышение конечной надежности исполнения здесь вносят, с одной стороны, свойственные интерпретации расширенные возможности диагностики, позволяющие еще заранее лучше отладить программу, а с другой стороны, то обстоятельство, что программная среда, обеспечиваемая виртуальной машиной, позволяет полностью контролировать все аспекты поведения программы.

Еще многие годы назад в трудах, посвященных надежности программного обеспечения, указывалось что в процессе разработки ПО каждый переход от одного способа описания задачи к другому таит в себе потенциальную опасность внесения ошибки. В этом смысле не составляет исключения и вынужденная для типичного отечественного программиста практика описания своей задачи в среде, основанной на чужом языке. Напротив, степень понятности, читабельности, а следовательно, и надежности программ, для написания которых использован **только родной язык**, существенно повышается.

Естественным образом, виртуальная машина, настроенная на работу с программами на русском «диалекте» языка Си++, и собственно русскоязычная программа образуют комбинацию, которую можно рекомендовать для исполнения приложений с повышенными требованиями к надежности.

Выполнение специальных функций

Системы, построенные на основе компилятора переднего плана, обладают практически неограниченными возможностями по надделению их специфическими функциями в соответствии с локальными требованиями. Это обстоятельство мы проиллюстрируем на примере задач инструментальной поддержки качества при разработке программного обеспечения.

Инструментальная поддержка качества при разработке программного обеспечения

Компилятор, разработанный и оттестированный согласно методике, принятой в нашей компании, в свою очередь, может использоваться для тестирования программ на Си++. С помощью компилятора могут быть выполнены общие **статические проверки семантики**:

- проверка на соответствие стандарту Си++;
- проверка на наличие в программе формально допустимых фрагментов, которые представляют собой потенциально опасные конструкции (например, произвольные преобразования типов), либо ограничивают переносимость программы на другие платформы.

После минимальных доработок могут быть выполнены проверки на соответствие локальным требованиям:

- проверка на соответствие отраслевым или внутрифирменным стандартам программирования.

Виртуальная машина дает возможность проводить **динамический анализ** программ, обеспечивая тем самым решение гораздо более широкого круга задач, связанных с тестированием ПО. Например, традиционная отладка программ (пошаговое исполнение, визуализация текущих значений переменных и т.п.) является частным случаем динамического анализа.

Виртуальная машина поддерживает модельное исполнение программ в специальной программной среде, которая позволяет полностью контролировать все аспекты поведения программы.

Примерами вариантов поведения программы, которые детектируются соответствующими схемами модельного исполнения, могут быть:

- непосредственное обращение к средствам операционной системы,
- попытки доступа к сетевому сервису и т.д.

В принципе в соответствии с локальными требованиями можно задать любую схему «подозрительно-го» поведения программы, которую будет детектировать виртуальная машина.

Когда все в Ваших руках

Во введении было заявлено, что компанией «Интерстрон» созданы объективные предпосылки для решения множества задач в соответствии с локальными требованиями заказчиков.

Вот тезисы в подтверждение этого:

1. **Продукция компании** – компилятор полного стандарта Си++ – **качественный программный продукт.**

Компилятор – это полностью оригинальная разработка с доступными исходными текстами (более 200 тыс. строк языка Си++). Исходные тексты выполнены в едином стиле с подробными комментариями. Имеет обширный комплект документации, в том числе, описание внутреннего устройства.

Компилятор полностью переносим на различные программно-аппаратные платформы, допускает естественную интеграцию практически в любое программное окружение.

Компилятор прошел полное тестирование на соответствие стандарту с использованием специального тестового набора (около 7 тысяч программ), разработанного специалистами Московского государственного университета.

2. **Стиль компании** – организация производства – **системный подход** к технологии разработок.

Среди принципов компании – документирование деятельности и формализация рабочих процессов.

Текущее повседневное взаимодействие сотрудников и текущий рабочий процесс собственно программирования организованы с применением автоматизированных средств поддержки.

Любое изменение исходного текста компилятора в процессе его разработки и отладки сопровождается регулярным тестированием. Вопросами тестирования постоянно занимается отдельный, организационно независимый коллектив.

Компания осуществляет техническую поддержку, в том числе, непосредственно у заказчика.

3. **Сотрудники компании** – разработчики компилятора – **квалифицированная команда** программистов.

Прежде всего, они смогли «сделать это». Накоплен немалый реальный опыт. Понятие «технологическая дисциплина» стало привычно сотрудникам компании и наполнено для них практическим смыслом.

Иными словами, если Вам понадобится создать инструментальную систему, ориентированную на язык Си++, в соответствии с Вашими специфическими требованиями, то все в Ваших руках – у нас есть предложение:

- что взять за основу;
- как это делать;
- с чьим содействием.

Приложение

Варианты использования компилятора переднего плана

1. Верификация исходных текстов на C++

1.1. Все вопросы сертификации.

Компилятор переднего плана (КПП) C++ можно использовать как основу для построения систем сертификации программных систем на соответствие тем или иным требованиям обеспечения секретности.

1.2. Соответствие отраслевым стандартам.

Каждая организация, как правило, имеет набор стандартов на написание программного обеспечения (в том числе, например, и на разметку исходного текста, именованье сущностей и т.д.). Проверять выполнение требований этих стандартов обычно довольно затруднительно. На основе компилятора переднего плана можно построить автоматическую систему проверки, которая бы на входе получала формализованное описание стандартов и исходный текст проверяемой программы, а на выходе формировала отчет по результатам проверки.

1.3. Тестирование интерфейсов.

К сожалению, язык C++ сам по себе не содержит синтаксиса и семантики для описания граничных условий применения тех или иных функций, и поэтому компилятор не может самостоятельно проверить правильность написания кода в этом смысле. Однако на основе КПП C++ можно построить автоматическую систему тестирования правильности написания кода с точки зрения технического задания.

Например, если есть техническое задание на написание некоторого класса Driver, то такая система могла бы написать программу, которая использует этот класс и вызывает его методы в «случайной» последовательности с подачей на вход параметров в заданном диапазоне допустимых, критических и запредельных значений.

Обычно такие тесты пишутся людьми на основе задания на разработку тестируемого программного компонента. Но вполне можно построить программный комплекс, который будет успешно автоматизировать данную работу. Это позволит существенно увеличить качество тестирования и снизить затраты на его проведения, а сегодня это почти треть стоимости разработки ПО.

2. Анализ программ на C++

2.1. Статический.

Легко строится система для проведения любого требуемого статического анализа, выявления узких мест или недокументированных связей между модулями программной системы.

2.2. Динамический.

С использованием «Виртуальной машины C++» (VM) можно проводить также и динамический анализ. Это позволит проверять корректность поведения программы при реальном исполнении. К VM можно подключить модель внешней среды и проводить «стендовые» испытания. В условиях, когда прогон программной системы на реальном объекте управления невозможен (например, это очень дорогая операция), такая модельная среда может имитировать поведение объекта управления.

3. Компиляторы для спецпроцессоров

Современное развитие микропроцессорной техники позволяет строить специализированные микропроцессорные устройства уже с достаточно большими вычислительными ресурсами. На таких изделиях использование компилятора с языка C++ становится уже вполне оправданным. Это позволит, с одной стороны, строить довольно сложные системы, а с другой – благодаря применению высокоуровневого языка C++ не слишком сильно увеличивать при этом их стоимость.

Таким образом, можно делать полные компиляторы C++ для применения в качестве средств разработки программного обеспечения для спецпроцессоров. При этом возможна как прямая продажа КПП C++ производителю, так и кооперация с ним в целях создания такого ПО «на заказ».

4. Компилятор для сертифицированной ОС

Есть несколько сертифицированных в России операционных системы (ОС), однако для них не

существует сертифицированных компиляторов. Представляется разумным и возможным выполнить работы по написанию кодогенератора для такой ОС и получить таким образом сертифицированный компилятор. С помощью такой связки можно строить системы, которые смогут проходить сертификацию практически любого уровня.

5. Обучение

Важной особенностью КПП является его глубокая локализация. Это, например, дает существенные преимущества в использовании его в качестве компилятора, на котором происходит обучение программированию на языке C++. И если для гражданских ВУЗов это представляется желательным, но не обязательным, то для системы МО это может оказаться достаточно важным. Особенно если учесть, что количество военнослужащих офицерского состава сокращается, и остро стоит проблема их переквалификации в гражданские специальности.

6. Документирование

6.1. Построение документации.

Несложно построить систему, которая будет строить на основе исходного текста полный набор документации. При этом формат получаемой информации может быть практически любым и определяется исключительно требованиями заказчика. Например, это может быть Microsoft Word или HTML/XML. На основе последнего можно строить уже не только документацию, но и получать гипертекстовые представления, удобные для оперативной работы.

6.2. Использование «Визуализатора»

Используя компоненту «Визуализатор», такую документацию можно снабдить не только текстом, но и развитым графическим представлением данных документируемой программной системы.

Разработка компилятора Си++: примеры проектных решений

Евгений Александрович ЗУЕВ

Ведущий специалист отдела разработки компиляторов

Наверное, не надо специально доказывать, что язык Си++ – наиболее объемный и сложный из всех известных к настоящему времени языков программирования. Единственное, что необходимо отметить, – что эта сложность, по нашему мнению, носит объективный характер, являясь отражением сложности задач, стоящих перед современным программированием, на решение которых и ориентирован Си++.

Создание компилятора даже для сравнительно простого языка программирования – большая и многоаспектная проблема. Достаточно перелистать книги, посвященные вопросам компиляции, чтобы оценить сложность задачи. Спектр проблем, решаемых разработчиками компилятора, охватывает практически всю проблематику современного программирования: проектирование и реализация структур и баз данных (включая алгоритмы над графами), формальные грамматики и принципы синтаксического разбора, объектно-ориентированный подход, теория типов, архитектура современных процессоров, принципы генерации и оптимизации кода, вопросы реализации параллелизма, и многое-многое другое. Не говоря уже о том, что для успешного создания компилятора необходимо хорошо представлять себе принципы и подходы к проектированию и разработке больших программных систем...

Сочетание сложности компилятора как такового и сложности конкретного входного языка делает задачу создания компилятора Си++ особенно нетривиальной. Нельзя сказать, чтобы мы отчетливо понимали эту сложность в начале работы (если бы мы в свое время это осознавали – ни за что бы не взялись :-). Но и это еще не все.

Были, по крайней мере, два дополнительных «аспекта сложности». Во-первых, компилятор реального промышленного языка всегда существенно отличается от «академических» компиляторов, структура которых описывается в многочисленных книгах. Иными словами, приходится учиться в процессе разработки («...Забудьте все, чему вас учили в школе...»). Второе «измерение сложности», с которым пришлось столкнуться, заключалось в том, что в процессе разработки компилятора *изменялся сам входной язык*, так как разработка шла параллельно с процессом стандартизации Си++. Этот процесс закончился в конце 1998 года, и именно к этому моменту компилятор успешно прошел основной массив тестов на соответствие (только что принятому!) Международному Стандарту.

В статье приводится краткое описание двух относительно небольших, но концептуально весьма важных проектных решений. Эти решения, помимо прочего, иллюстрируют и определенную неполноту книг и учебных курсов по компиляции: описываемые проблемы, несмотря на их насущность и актуальность, даже не обозначены в большинстве информационных источников. В конце статьи высказываются некоторые дополнительные соображения по поводу дальнейших усовершенствований компилятора.

Организация лексического разбора: отдельно или вместе?

Проектирование и разработка лексических и синтаксических компонент компилятора – хорошо изученная тема. Любая книга по компиляции начинается с обсуждения этой проблематики и зачастую посвящает ей большую часть своего объема. Однако при создании компилятора промышленного ЯП (например, Си++) возникает ряд проблем, недостаточно отраженных в литературе. С несколькими такими проблемами мы и столкнулись.

Первая проблема касается взаимоотношения лексического анализатора и препроцессора. Как известно, компиляция программ на Си/Си++ предусматривает логически отдельный этап – препроцессорную обработку исходного текста. Эта фаза является откровенной архаикой, сохранившейся еще с конца 60-х годов, приносит при программировании больше проблем, нежели решает, и, строго говоря, не имеет прямого отношения к языку Си++ как таковому. Тем не ме-

нее, слишком многие реальные программы активно используют средства препроцессорирования (можно даже сказать, что это общепринятая парадигма программирования на Си/Си++), и поэтому компилятор, безусловно, обязан их поддерживать.

Большинство систем реализуют этот этап в виде отдельного препроцессора, выход которого в виде текстового файла поступает на вход компилятора. Однако на практике взаимодействие двух компонент – препроцессора и компилятора – посредством дискового файла является слишком неэффективным, так как в этом случае мы имеем две дополнительных и весьма затратных операции – запись результата препроцессорирования на дисковую память и повторное считывание ее компилятором. Стандарт языка Си++ не фиксирует конкретный способ реализации этих фаз; единственное, что требуется, – обеспечить логическое соответствие очередности этапов. Иными словами, если входной поток символов перед декомпозицией его на лексемы будет «на лету» подвергаться препроцессорной обработке, то требование стандарта будет соблюдено, и при этом мы избежим записи промежуточного результата на внешний носитель с последующим чтением его.

Кроме того, характер обработки текста на этапах препроцессорирования и лексического разбора очень схож; в обоих случаях основным понятием является «лексема»: на этапе препроцессорирования это «лексема препроцессора» («pp-token»), на этапе лексического разбора – просто «лексема» («token»), при том, что содержательно эти понятия отличаются друг от друга весьма незначительно. На практике это приводит к тому, что препроцессор и лексический анализатор, реализованные как отдельные компоненты, в многом повторяют друг друга и попросту содержат значительные фрагменты одинакового кода.

Что касается таких возможностей «раздельной» организации, как, скажем, препроцессорирование без компиляции или, наоборот, компиляция без предварительного препроцессорирования, которые естественно реализуются в рамках двух отдельных компонент (препроцессора и компилятора), то обеспечить такую возможность достаточно легко и в «объединенной» схеме, введя соответствующие режимы ее работы.

По этим причинам после некоторого этапа пробных разработок было принято решение, характерное далеко не для всех известных нам компиляторов, – реализовать «объединенную» схему, при которой лексический анализатор выполняет также и все функции препроцессора. Мы не проводили специальных сравнений эффективности, однако можно определенно утверждать, что в результате общий код существенно сократился, а производительность заметно повысилась.

Описанная проблема, а также выбранный способ ее решения показывает сложность и неоднозначность проектирования любой сколько-нибудь нетривиальной программной системы. В самом деле, на первый взгляд, наше решение нарушает один из основополагающих принципов проектирования – принцип модульности, согласно которому компоненты системы должны представлять собой относительно небольшие единицы с простыми и строго определенными интерфейсами. Мы же, наоборот, объединили две «немаленькие» компоненты в одну. В пользу такого «неправильного» решения говорит сильная схожесть этих двух компонент, большее удобство последующего сопровождения, а также явный выигрыш в быстродействии.

Синтаксический анализ: вместе или отдельно?

Вторая проблема, с которой мы столкнулись, проектируя начальные этапы компиляции, – это способ организации синтаксического разбора. Если коротко, вопрос стоял так: делать «ручной» разбор, то есть непосредственно программировать распознавание синтаксических конструкций исходной программы, либо сконструировать такой анализатор автоматически, используя какой-либо доступный генератор распознавателей (например, YACC или Bison)?

У автора было сильное искушение сделать все вручную, методом рекурсивного спуска, благо, опыта было достаточно. В пользу такого варианта говорил и опыт создателей известного компилятора фирмы Edison Design Group, в котором использовался такой же подход. Тем не менее, в результате обсуждений было принято решение использовать одну из свободно-доступных версий YACC. Схема разработки, диктуемая семейством YACC, заключается в следующем: грамматика входного языка описывается на специальном метаязыке. Из этого описания генери-

руется распознаватель, представляющий собой Си-программу, которая объединяется с другими компонентами компилятора.

Кратко коснемся преимуществ описанной схемы, а также ее недостатков и проблем, вызываемых ее применением.

Основное и в конечном счете решающее преимущество этой схемы с точки зрения общей архитектуры компилятора заключается в явном разделении различных компонент компилятора, то есть, в повышении его модульности. Синтаксис языка, представленный в очень наглядном, а потому в удобном для сопровождения виде, оказывается сосредоточенным в относительно компактном текстовом файле. Это описание (которое, между прочим, имеет и самостоятельную ценность, представляя собой однозначную спецификацию входного языка) легко отделяется от других компонент компилятора (как от лексической части, так и от алгоритмов семантической обработки); сгенерированный анализатор автоматически обеспечивает взаимодействие анализатора с этими компонентами.

Другое преимущество связано со сравнительно небольшими временными затратами на разработку грамматического описания входного языка. Даже учитывая сложность Си++ и необходимость существенной переработки его синтаксиса (об этом см. ниже), сделать работоспособное описание грамматики оказалось существенно быстрее и легче, чем вручную программировать разбор всех синтаксических конструкций входного языка. Имея некоторый навык «мета-программирования», можно создать работающую грамматику реального ЯП с нуля за несколько дней (другое дело, что для получения этого навыка придется потратить много времени на освоение метасредств :-)

Однако, проектное решение такой, в некотором смысле фундаментальной важности, не может не иметь и своих теневых аспектов. Преодолевая недостатки, присущие описанной схеме, пришлось решать множество сопутствующих проблем.

Во-первых, YACC-подобный генератор оказался не в состоянии корректно воспринять синтаксис Си++ в том виде, в котором он представлен в Стандарте. Правда, на это мы и не слишком рассчитывали: во-первых, он и не предназначен для генерации распознавателей, а служит только целям презентации и, во-вторых, он явно слишком общий и содержит ряд принципиальных неоднозначностей (их подробное обсуждение займет слишком много места). Стало быть, синтаксис языка должен быть специально подготовлен и существенно переработан относительно оригинального (по существу, переписан заново).

Во-вторых, хотя класс грамматик, воспринимаемых YACC'ом, достаточно широк и соответствует многим реальным языкам программирования; тем не менее, оригинальный синтаксис Си++, даже после переработки, «не вмещается» в этот класс. Следовательно, приходится специально адаптировать его (упрощать), чтобы генератор смог сформировать по нему корректный синтаксический распознаватель. Однако, модификация синтаксиса неизбежно влияет на «соседние» компоненты, прежде всего, на лексический анализатор! Дело в том, что адаптация неизбежно подразумевает введение дополнительных грамматических правил, а также новых лексем. Новые лексемы появляются либо в результате агрегации нескольких исходных лексем в одну (например, все знаки операций класса присваивания обрабатываются совершенно одинаково, и потому можно заменить всю группу соответствующих лексем на единственную), либо, наоборот, из-за необходимости детализировать некоторые лексемные излишне общего характера. Так, лексема «идентификатор» оказалась слишком «абстрактной», и вместо нее пришлось вводить целый класс более «конкретных» лексем («имя типа», «имя-нетипа», «новое-имя» и т.д.). Естественно, это потребовало усложнить алгоритм лексического анализа, который мог бы идентифицировать и поставлять парсеру такие детализированные лексемные.

Следующая проблема коренится в способе организации интерфейса сгенерированного анализатора с остальными компонентами компилятора. Кратко об этом способе можно сказать следующее: во-первых, сложный, во-вторых, устаревший. Так например, на вход анализатору должен подаваться целочисленный код очередной лексем; синтаксический анализатор оперирует именно кодами лексем. Однако целочисленного кода лексем явно недостаточно для целей компиляции в целом: нам хотелось бы знать о лексеме гораздо больше (и мы реально получаем эту информацию от лексана), поэтому дополнительную информацию приходится хранить где-то отдельно. Естественно, это усложняет общую организацию компилятора.

Еще больше неудобств доставляет взаимодействие анализатора с семантическими процедурами компилятора. Чтобы передать в некоторую синтаксическую конструкцию информацию, полученную в результате разбора другой конструкции (например, при переходе от обработки низкоуровневой конструкции к более общей), приходится использовать так называемый семантический стек – базовую структуру разбора. Состав и структура семантической информации, возникающей в ходе разбора той или иной конструкции, определяется ее смыслом и для различных конструкций сильно различается. Поэтому, чтобы воспользоваться общим «хранилищем» такой информации – семантическим стеком, приходится искусственно подгонять эти структуры к формату стека. Каждый элемент стека представляет собой структуру, описанную посредством Си-объединения (union); чтобы обеспечить передачу информации, приходится специально определять структуры-варианты этого объединения (которые не имеют самостоятельной ценности и нужны только для целей передачи) и заполнять их семантической информацией. В результате сейчас мы имеем более трех десятков таких «подсобных» структур, и все семантические процедуры оказываются логически ориентированными на семантический стек анализатора.

Наконец, последнее, о чем хотелось бы сказать в связи с обсуждаемым решением: сравнительная «жесткость» схемы с явным отделением синтаксической части. Так, отталкиваясь от первоначального описания языка, мы не стали явно специфицировать в грамматике конструкцию квалифицированных имен Си++, решив делать разбор таких имен вручную, на этапе лексического анализа, а в синтаксический анализатор подавать «окончательную» лексему, полученную в результате идентификации квалифицированного имени. Это сильно упрощало синтаксический анализ и в целом приводило к более эффективному разбору.

Много позже того, как такое решение было полностью реализовано и отлажено, получил существенное развитие механизм шаблонов Си++, и оказалось, что синтаксис квалифицированных имен теперь может включать настройки шаблонов! Для нас это означало, что квалифицированное имя, вообще говоря, не может быть разобрано лексическим анализатором – в нем теперь могут встречаться выражения и другие синтаксические конструкции, разбор которых – прерогатива синтаксического анализа. Преодоление возникшей проблемы потребовало значительных усилий и времени.

Оценивая решение, связанное с отдельной спецификацией синтаксиса Си++, следует отметить, что в целом оно оправдало надежды, возлагавшиеся на него, хотя и не оказалось панацеей от всех проблем. С другой стороны, опыт, полученный в результате разработки YACC-ориентированной синтаксической части компилятора показал настоятельную необходимость в более продвинутом генераторе распознавателей, который, во-первых, воспринимал бы более широкий класс грамматик и, во-вторых, обеспечивал бы более современный и развитый интерфейс с компонентами-«потребителями» конструируемого парсера.

Ошибки и их диагностика: несколько соображений на будущее

Сложность языка Си++, о которой говорилось в начале статьи, приводит, в частности, и к такому выводу: для ее преодоления, то есть для успешного освоения языка, традиционные подходы, вроде курсов, учебных пособий и т.п. – кажутся недостаточными. Если мы хотим, чтобы компилятор был привлекателен для пользователей, он должен обладать специальными, особыми свойствами, которые бы сделали его удобным не только для программирования как такового, но и для освоения языка в большем, нежели обычно, объеме. Я утверждаю, что даже опытные программисты – признают они это или нет – как правило, знают язык в далеко не достаточной мере и реально используют сравнительно небольшое его подмножество.

Два важнейших свойства компилятора, которые, на наш взгляд, помогут программистам как в их реальной работе, так и в освоении языка, – это локализованность компилятора и продвинутое средства диагностики ошибок.

Если говорить о локализованности, то основное здесь – допущение идентификаторов в национальном (в нашем случае – в русском) алфавите и выдача диагностических сообщений на русском языке. Оставим на время дискуссии о необходимости и правомерности введения русскоязычных эквивалентов служебных слов – оправданность русских идентификаторов (допускаемых, кстати, Стандартом) и сообщений вряд ли нуждается в серьезном обосновании. Кроме

того, понятно, что обеспечение подобного свойства не требует существенных дополнительных усилий при проектировании и разработке и потому уже содержится в компиляторе.

Что же касается качества диагностики компилятора, то здесь ситуация иная. Синтаксис и семантика Си++ как будто специально придумывались, чтобы допускать ошибки при программировании было очень легко, выявлять и локализовывать их компилятору очень трудно, а корректно продолжить обработку программы после обнаружения ошибки – почти невозможно. Даже самые лучшие компиляторы не могут похвастаться адекватной диагностикой. Как правило, только несколько первых сообщений действительно соответствуют некоторой ошибке – последующие являются «наведенными». В некоторых компиляторах (например, в Visual C++) адекватным является зачастую не первое сообщение, а почему-то второе!

Попробуйте, скажем, пропустить завершающую фигурную скобку у тела функции или класса, особенно если вслед за ней (за ним), опять идет функция или класс. Попробуйте объявить переменную, тип которой задан с помощью неопisanного ранее идентификатора. Вы получите все, что угодно, но только не сообщение, действительно соответствующее допущенной ошибке.

Чтобы хоть в какой-то мере преодолеть отмеченные проблемы, необходимо дальнейшее развитие компилятора. Кроме того, нужно учитывать статус компилятора переднего плана как центральной компоненты многих подсистем, о чем говорится в других статьях сборника. Поясню свою мысль.

Проблемы, связанные с диагностикой, как правило, рассматриваются в контексте необходимости выявления ошибок в программе, что имеет прямое отношение к традиционной задаче компиляции – порождению объектного кода. Базисная логика, которая в этом случае определяет подход к диагностике, примерно такова: если в программе обнаружена ошибка, то компилятор не в состоянии сгенерировать объектный код. Стало быть, необходимо обнаружить все ошибки, причем как можно больше за один сеанс компиляции (для экономии времени программиста), даже за счет ложных сообщений. По существу, самое важное действие, выполняемое компилятором при обнаружении ошибки, – это параметризация и вывод в выходной поток текстового сообщения, которое заранее подготовлено в компиляторе для определенного класса сходных ошибок.

Решение проблемы в традиционной парадигме связано с разработкой изоощренных контекстно-чувствительных алгоритмов диагностики и восстановления разбора. В силу крайней нерегулярности синтаксиса Си++ и сложной семантики его конструкций такие алгоритмы неизбежно должны будут рассматривать очень много «частных случаев» и в целом будут носить существенно эвристический характер. Попросту говоря, очень сложно придумать единое стройное решение, которое бы охватило сколько-нибудь существенный класс ошибок.

Системы, которые создаются на основе нашего компилятора, не обязательно связаны с традиционной компиляцией, понимаемой как формирование кода, подлежащего выполнению. Можно даже сказать, что задача генерации кода является лишь одной из многих задач, возникающих в процессе создания и сопровождения больших программных систем. Результат работы нашего компилятора – высокоуровневое промежуточное представление (ПП) – служит удобной основой для выполнения самых разных операций, связанных с программными текстами, включая их анализ с различных точек зрения, верификацию, тестовое (модельное) исполнение и прочее, и, в том числе, – генерацию кода.

Поэтому не только подход к диагностике ошибок должен отличаться от традиционного – само понятие «ошибки» нуждается в переосмыслении. В самом деле, если, например, в традиционной модели использование в объявлении неопisanного типа (класса) безусловно трактуется как ошибка (действительно, не зная типа переменной, невозможно сгенерировать код для случаев ее использования!), то для формирования корректного ПП совершенно необязательно знать тип переменной немедленно! Достаточно того, что компилятор распознает конструкцию «объявление», построит для нее эквивалентное ПП, отметив в нем неопisanный тип, как еще не известный, а конструкцию в целом – как неполную. Далее, в процессе комплексации, обнаружив описание недостающего типа, редактор связей может скорректировать данное объявление, превратив ПП для него в «полное».

Таким образом, «ошибка» из сущности, возникающей при анализе программы, неким сторонним инструментом (компилятором), превращается в определенное семантическое свойство

программы, изначально ей присущее. Наличие этого свойства (если оно сохранено в окончательном ПП) не даст генератору кода возможность сгенерировать объектный код, но оно никоим образом не мешает, скажем, визуализатору сформировать графическое представление программы!

Между прочим, это расширяет сферу использования того же визуализатора: тогда он может корректно обрабатывать и неполные программы (наброски), которые, как правило, только и имеются в процессе проектирования программной структуры.

Резюмируя сказанное, можно выдвинуть следующую базисную идею: «ошибку» (по крайней мере, семантическую ошибку) в программе на Си++ следует трактовать как одно из **свойств** программы (или ее части), которое тем или иным образом отражается в ее промежуточном представлении. Способ *интерпретации* этого свойства определяется задачами конкретной клиентской программы и может приводить к выводу соответствующего сообщения генератором кода, отображению визуализатором конструкции, обладающей этим свойством, в виде определенной пиктограммы, и т.д.

При такой трактовке задача формирования адекватных и подробных диагностик становится частью более общей задачи *анализа* программы. В этом случае, располагая полностью построенным ПП, такой анализатор, в принципе, может с любой степенью подробности и с учетом сколь угодно широкого контекста выдать информацию, относящуюся к данному «свойству» программы. Эта информация, будучи аккуратно структурированной и отображенной в понятном и привязанном к исходному тексту виде, может служить дополнительным и очень мощным средством как анализа, так и обучения разработчиков тонкостям языка.

