Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Введение в дипломную работу
«восстановление объектных структур данных при декомпиляции»

Выполнил:
студент 527 группы
Фокин А. П.

Научный руководитель:
к.ф.-м.н., доцент Чернов А. В.

Москва, 2008

Contents

1 Введение

Сегодня компьютеры используются повсеместно, и огромные средства вкладываются в создание программного обеспечения. Программное обеспечение требует постоянных изменений — для исправления ошибок, добавления новой функциональности, адаптации к новым требованиям, или новым возможностям аппаратуры. При этом исходный код программы является ключом к ее пониманию, и если он недоступен, то понимание принципов ее работы и внесение изменений оказывается чрезвычайно затруднено. Так как не редки случаи отсутствия исходного кода используемого приложения или компонента, то актуальной становится задача *обратного проектирования*.

Обратное проектирование (Reverse engineering) — это процесс извлечения информации о строении и принципах работы системы путем анализа ее функций, структуры, и поведения. В применении к программному обеспечению целью обратного проектирования является повышение уровня абстракции представления программ. В процессе повышения уровня абстракции принято выделять несколько этапов:

1. Получение ассемблерного представления программы из бинарного модуля, называемое *дизассемблированием*;

2. Восстановление программы на языке высокого уровня из ассемблерного представления, называемое *декомпиляцией*;

3. Последующее применение средств повышения уровня абстракции программ на языке высокого уровня в зависимости от целей анализа.

Сегодня для разработки большого числа программных систем применяется объектно-ориентированный подход с использованием языка Си++, и потому довольно часто возникают соответствующие задачи обратного проектирования. Дизассемблирование написанных на Си++ программ не имеет каких-либо специфичных особенностей, и потому может быть произведено стандартными методами. Декомпиляция же для языка Си++ является задачей, представляющей определенные трудности, так как Си++ оперирует множеством абстракций, некоторые из которых оказываются никак не отражены в машинном коде, получаемом в результате компиляции. Однако следует помнить, что основными концепциями объектно-ориентированного подхода является классы и объекты, и потому восстановление объектных структур существенно упрощает понимание принципов работы подобных программ.

Целью данной работы является разработка и реализация методов восстановления объектно-ориентированных структур данных в программах на Си++ из низкоуровнего представления.

2 Особенности языка Си++

Си++ является мультипарадигмальным языком программирования, и потому на Си++ возможно написание программ, которые вообще не будут использовать принципов объектно-ориентированного программирования. К примеру, даже если в программе на языке Си++ встречается ключевое слово class, это еще не значит, что в ней используются такие концепции объектно-ориентированного программирования, как полиморфизм или наследование. Классы в Си++ можно использовать просто как замену структурам в Си.

Также с развитием Си++ стало возможно использование техник метапрограммирования[1], таких как *expression templates*[2], которые производят нетривиальные преобразования и вычисления на этапе компиляции, и в результате применения которых в исполняемый файл попадают только результаты этих преобразований. В общем случае автоматическое восстановление конструкций метапрограммирования не представляется возможным — к примеру, константа 720, встреченная при декомпиляции, в оригинальном исходном коде могла быть записана как вызов метафункции, вычисляющей факториал.

С учетом сказанного выше, мы будет рассматривать программы на Си++, использующие такие средства объектно-ориентированного программирования, как:

- Динамический полиморфизм — механизм, позволяющий единообразно работать с объектами различных классов с отличающийся реализацией через общий интерфейс, определенный в суперклассе. При этом фактическая реализация класса определяется во время выполнения. В Си++ динамический полиморфизм обеспечивается механизмом виртуальных функций. В дальнейшем класс, имеющий хотя бы одну виртуальную функцию, будем называть *полиморфным*.

- Динамическая идентификация типов данных (в дальнейшем RTTI, Run-Time Type Identification) — механизм, который позволяет определить тип данных переменной или объекта во время выполнения программы.

В Си++ RTTI используется для реализации следующих механизмов:

- Оператор dynamic_cast<>. Этот оператор используется для приведения типов, однако в отличие от приведения типов в стиле C, проверка на возможность корректного приведения производится во время выполнения программы, а не во время компиляции. Обычно этот оператор используется для "нисходящего" приведения типов в иерархии классов - от базового класса к потомку. При этом производится проверка на то, что переданный объект действительно является экземпляром нужного класса-потомка. Очевидно, что для реализации такой проверки необходимо на этапе времени выполнения обладать информацией о структуре иерархии классов.

Еще один способ использования оператора dynamic_cast<> - это преобразование к типу void*. При этом, в отличие от обычного приведения типов, возвращается указатель на объемлющий объект-потомок, который в случае множественного наследования может не совпадать с указателем на текущий объект.

- Оператор typeid. Этот оператор прежде всего используется для определения класса переданного объекта во время выполнения. Он возвращает объект типа type_info, с использованием которого можно в частности получить строковое представление имени рассматриваемого класса. Вообще говоря, оператор typeid можно применять к выражениям любого типа, и эта особенность может быть использована в реализации механизма обработки исключений.

- Исключения (exceptions). При обработке исключений необходимо сравнивать тип выброшенного исключения с типом, указанным в catch-блоке. Для таких проверок на этапе времени выполнения также может потребоваться информация о структуре иерархии классов.

Очевидно, что использование описанных выше средств Си++ требует сохранения в исполняемом файле дополнительной информации, которая будет использоваться реализацией их функциональности во время выполнения программы. Но если динамический полиморфизм при разработке приложений используется очень часто, то механизмы RTTI применяются значительно реже. В частности, использование оператора dynamic_cast<> считается признаком наличия просчетов в архитектуре приложения, и поэтому его стараются избегать. Оператор же typeid используется для решения очень узкого круга задач. Только механизм исключений применяется достаточно часто, но современные компиляторы позволяют использовать его при отключенном RTTI. В этом случае дополнительная информация будет сгенерирована не для всех полиморфных классов, а только для тех, которые используются в throw- и catch-блоках [3].

Казалось бы, это означает, что в большинстве программ на Си++ дополнительная RTTI информация должна отсутствовать. Однако по следующим причинам это зачастую оказывается не так:

- RTTI является стандартизированным механизмом Си++. Отключение этого механизма фактически делает компилятор несовместимым со стандартом Си++. Поэтому в компиляторах поддержка RTTI обычно включена по умолчанию.

- Согласно идеологии Си++, неиспользуемые механизмы языка не должны добавлять издержек времени выполнения. Поддержка RTTI не замедляет тот код, который не использует этот механизм. По сути дела, в таком случае RTTI лишь увеличивает размер бинарного модуля, добавляя в него дополнительную информацию о полиморфных классах.

- В случае, если конечным продуктом разработки является библиотека, а не исполнимый модуль, то компиляция без поддержки RTTI может вызвать проблемы с совместимостью. В частности, использование библиотеки, скомпилированной без поддержки RTTI в проекте, использующем RTTI, может привести к ошибкам.

3 Детали реализации механизмов языка Си++

Как было отмечено выше, использование таких механизмов Си++, как динамический полиморфизм и RTTI, требует сохранения в исполняемом файле дополнительной информации. Знание формата этой информации может существенно упростить восстановление объектных структур данных при декомпиляции. Используемый формат, вообще говоря, зависит от компилятора и платформы. В этой работы мы рассмотрим способы восстановления объектно-ориентированных структур данных в программах на языке Си++, скомпилированных с помощью компиляторов GCC[1] и MSVC[2] под платформу x86. Формат дополнительной информации, используемой во время выполнения программы, определяется *бинарным интерфейсом приложений*.

Бинарный интерфейс приложений (в дальнейшем ABI, Application Binary Interface) — набор конфигураций среды разработки и компилятора, гарантирующих успешную бинарную совместимость разрабатываемых приложений. Бинарный интерфейс приложений определяет взаимодействие на низком уровне между приложениями, между компонентами приложения, между приложением и библиотеками и между приложением и операционной системой на используемой платформе.

Рассмотрим реализацию описанных выше средств Си++ в ABI компиляторов GCC и MSVC для платформы x86.

3.1 Реализация динамического полиморфизма в компиляторах GCC и MSVC

Как было сказано выше, динамический полиморфизм в Си++ реализуется через механизм виртуальных функций. При этом для каждого полиморфного класса генерируется *таблица виртуальных функций*. Эта таблица содержит в себе указатели на все виртуальные функции класса. Каждый объект такого класса первым полем хранит указатель на соответствующую таблицу, и все вызовы виртуальных функций производятся косвенно, с использованием этого указателя.

Такая реализация механизма виртуальных функций используется как в GCC, так и в MSVC.

[1]GNU Compiler Collection — набор компиляторов для различных языков программирования и платформ, разработанный в рамках проекта GNU и распространяемый на условиях GNU General Public License. Нами рассматривается компилятор языка C++ для платформы x86.

[2]Microsoft Visual C++ — интегрированная среда разработки приложений на языке C++, разработанная фирмой Microsoft и поставляемая как часть комплекта Microsoft Visual Studio. Нами рассматривается компилятор для платформы x86, используемый в этой среде разработки.

3.2 Реализация RTTI в компиляторе MSVC

Компилятор MSVC при использовании RTTI для каждого полиморфного класса генерирует несколько структур, которые впоследствии используется реализацией механизма обработки исключений и оператором dynamic_cast<>. Эти структуры не документированы Microsoft, и их формат был восстановлен с использованием методов обратного проектирования. Приведенное ниже описание этих структур было взято из исходных кодов проекта Wine [8].

Для каждого полиморфного класса перед таблицей виртуальных функций помещается указатель на RTTICompleteObjectLocator, имеющий следующую структуру:

```
struct RTTICompleteObjectLocator {
  /** Всегда ноль. */
  unsigned long signature;

  /** Смещение соответствующей таблицы виртуальных функций
   * в объемлющем классе. */
  unsigned long offset;

  /** Смещение для вызова конструктора. */
  unsigned long cdOffset;

  /** Указатель на соответствующий объект типа type_info. */
  type_info* pTypeInfo;

  /** Описание иерархии наследования. */
  RTTIClassHierarchyDescriptor* pClassDescriptor;
};
```

Иерархия наследования описывается несколькими структурами типа RTTIClassHierarchyDescriptor:

```
struct RTTIClassHierarchyDescriptor {
  /** Всегда ноль. */
  unsigned long signature;

  /** Нулевой бит отвечает за множественное наследование,
   * первый бит — за виртуальное. */
  unsigned long attributes;

  /** Количество классов в pBaseClassArray. */
  unsigned long numBaseClasses;

  /** Описывает базовые классы. */
  RTTIBaseClassDescriptor** pBaseClassArray;
};
```

Базовые классы описываются структурами типа RTTIBaseClassDescriptor:

```
struct RTTIBaseClassDescriptor {
  /** Указатель на соответствующий объект типа type_info. */
  type_info* pTypeInfo;

  /** Количество последующих записей в массиве. */
  unsigned long numContainedBases;

  /** Информация для создания указателей на член класса. */
  struct PMD where;

  /** Атрибуты, обычно ноль. */
  unsigned long attributes;
};
```

Описанные выше структуры RTTICompleteObjectLocator, RTTIClassHierarchyDescriptor и RTTIBaseClassDescriptor помещаются в сегмент данных бинарного модуля. Очевидно, что зная точное расположение этих структур, можно восстановить полную иерархию полиморфных классов приложения. Кроме того, для каждого полиморфного класса генерируется соответствующий объект типа type_info, имеющий следующую структуру:

```
class type_info {
public:
  /* ... */
private:
  void *_m_data;
  char _m_d_name[1];
  /* ... */
};
```

В поле _m_d_name записано закодированное (mangled) имя класса, соответствующего данному объекту type_info. Для декодирования этого имени можно использовать недокументированную функцию _unDName из библиотеки времени выполнения Microsoft Visual C. Таким образом, используя RTTI информацию, можно восстановить полную иерархию полиморфных классов вместе с их именами. Стоит отметить, что так как при проектировании объектных систем классам стараются давать осмысленные имена, соответствующие их функциональности, то при условии, что перед компиляцией рассматриваемая программа не была подвергнута обфускации, знание оригинальных имен классов может существенно упростить понимание принципов ее работы.

3.3 Реализация RTTI в компиляторе GCC

В отличие от MSVC, компилятор GCC хранит всю дополнительную информацию о полиморфных классах в соответствующих им объектах type_info. Описание иерархии классов type_info приводится в заголовочном файле <cxxabi.h>, и доступно для использования из любой программы,

написанной на Си++. Организация доступа к RTTI информации в GCC схожа с подходом, используемым в MSVC — для каждого полиморфного класса перед таблицей виртуальных функций помещается указатель на соответствующий ему объект type_info.

В GCC ABI используется иерархия классов с корнем в std::type_info. Для классов, не имеющих базовых классов, используются тип __class_type_info:

```
class __class_type_info: public std::type_info {}
```

Этот тип также является базовым для других представлений информации о классах. Для классов, имеющих один невиртуальный публичный (public) базовый класс, используется тип __si_class_type_info:

```
class __si_class_type_info: public __class_type_info {
public:
  /** Указатель на __class_type_info базового класса. */
  const __class_type_info *__base_type;
};
```

Для всех прочих классов используется тип __vmi_class_type_info

```
class __vmi_class_type_info: public __class_type_info {
public:
  /** Комбинация флагов из __flags_masks. */
  unsigned int __flags;

  /** Количество базовых классов. */
  unsigned int __base_count;

  /** Массив структур, описывающих базовые классы. */
  __base_class_type_info __base_info[1];

  enum __flags_masks {
    /** В иерархии имеется несколько экземпляров
     * одного и того же базового класса. */
    __non_diamond_repeat_mask = 0x1,

    /** Иерархия имеет ромбовидную форму. */
    __diamond_shaped_mask = 0x2
  };
};
```

Где структура __base_class_type_info описана следующим образом:

```
struct __base_class_type_info {
public:
  /** Указатель на __class_type_info базового класса. */
  const __class_type_info *__base_type;

  /** Флаги и смещение данного класса в объемлющем. */
  long __offset_flags;

  enum __offset_flags_masks {
    /** Базовый класс - виртуальный. */
    __virtual_mask = 0x1,

    /** Базовый класс - публичный. */
    __public_mask = 0x2,

    /** Сдвиг в поле __offset_flags до смещения. */
    __offset_shift = 8
  };
};
```

Ясно, что используя информацию, предоставляемую объектами type_info, можно восстановить полную иерархию полиморфных классов приложения. Также, аналогично с MSVC, возможно восстановление имен классов. Рассмотрим класс type_info:

```
class type_info {
public:
  /* ... */
private:
  const char *__type_name;
  /* ... */
};
```

В поле __type_name хранится указатель на закодированное (mangled) имя класса, соответствующего данному объекту type_info. Декодирование осуществляется с помощью функции __cxa_demangle, объявленной в <cxxabi.h>. Таким образом, GCC предоставляет возможности по восстановлению иерархий полиморфных классов как минимум такие же, как MSVC.

4 Постановка задачи

Цель данной работы — разработка и реализация методов восстановления объектно-ориентированных структур данных в программах на Си++ из ассемблерного представления для компиляторов GCC и MSVC на платформе x86.

С учетом описанных выше особенностей языка Си++, задачу разработки таких методов можно разделить на следующие подзадачи:

1. Определить, какую информацию можно извлечь из структур, поддерживающих механизмы RTTI и обработки исключений, и предложить способы извлечения этой информации;

2. Определить, какую информацию об объектно-ориентированных структурах данных можно извлечь из ассемблерного представления программы при отключенном RTTI;

3. Разработать алгоритмы восстановления объектно-ориентированных структур данных с использованием извлеченной информации.

Данная работа посвящена решению этих подзадач и реализации полученных методов в рамках существующей системы восстановления типов в программах, написанных на языках Си и Си++, разрабатываемой в ИСП РАН.

5 Существующие исследования в области восстановления объектно-ориентированных структур данных

Несмотря на то, что проблема декомпиляции известна давно, серьезных исследований в области восстановления объектно-ориентированных структур данных для языка Си++ до сих пор не было проведено. Для таких языков, как Java, или C#, восстановление объектно-ориентированных структур зачастую не представляет никаких сложностей, и существует множество успешных декомпиляторов для этих языков, способных восстановить код в практически изначальном виде. Однако в общей задаче декомпиляции из ассемблерного представления до сих пор существует много нерешенных проблем.

Наиболее близки к проблеме восстановления объектно-ориентированных структур данных исследования в области восстановления типов данных при декомпиляции, например [5]. Однако такие исследования фокусируются на восстановлении внутренней структуры типов данных, а не отношений между ними.

Наиболее интересны с точки зрения рассматриваемой проблемы работы [6] и [7]. Первая из этих работ посвящена обзору основных проблем, возникших при использовании декомпилятора Boomerang для восстановления алгоритмов, использованных в системе анализа речи. Система представляла собой большое Windows-приложение, написанное на Си и Си++, и авторам были доступны исходные коды прототипа системы. В этой работе авторы описали возникшие при декомпиляции проблемы и использованные для их решения методы, а также некоторые "неожиданности", обнаруженные в ходе работы. Одной из таких "неожиданностей" стало наличие в исполнимом модуле RTTI информации, с использованием которой авторам удалось восстановить полную иерархию классов системы со всеми именами. Однако технические детали восстановления иерархии классов авторами предоставлены не были. Во второй работе также упоминается возможность использования RTTI информации при декомпиляции, но детали не приводятся.

Как мы видим, задаче восстановления объектно-ориентированных структур данных на данный момент было уделено мало внимания, и было проведено лишь небольшое количество исследований в этой области.

6 Текущие результаты

На данный момент была сформулирована задача дипломной работы и предложены подходы к ее решению. Для того, чтобы достичь цель дипломной работы, необходимо исследовать возможности восстановления объектно-ориентированных структур данных при отсутствии RTTI информации, реализовать предложенные методы в рамках существующей системы, провести эксперименты на реальных программах, и при необходимости внести изменения в используемые методы.

References

[1] D. Abrahams, A. Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, 2004.

[2] T. Veldhuizen. Expression Templates. C++ Report, Vol. 7, No. 5, June 1995, pp. 26-31.

[3] V. Kochhar. How a C++ compiler implements exception handling. Article at CodeProject.org, 2002. Retrieved Mar 2009 from http://www.codeproject.com/KB/cpp/exceptionhandler.aspx.

[4] M. Ball, et al. Determining the actual class of an object at run time. United States Patent 6138269, 2000.

[5] A. Mycroft. Type-Based Decompilation. Proceedings of the 8th European Symposium on Programming Languages and Systems, pp. 208-223, 1999.

[6] M. V. Emmerik, T. Waddington. Using a Decompiler for Real-World Source Recovery. Proceedings of the 11th Working Conference on Reverse Engineering, 2004.

[7] M. V. Emmerik. Static Single Assignment for Decompilation. A thesis submitted for the degree of Doctor of Philosophy at The University of Queensland, 2007.

[8] Wine web page. LGPL licensed software. Retrieved Mar 2009 from http://www.winehq.org.