Московский Государственный Университет

им. М.В.Ломоносова

Научно-исследовательский вычислительный центр

На правах рукописи

УДК 004.4'422

КРОТОВ АЛЕКСАНДР НИКОЛАЕВИЧ

Принципы реализации семантики языка Си++ в системе 3С++

05.13.11 - математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация

на соискание ученой степени

кандидата технических наук

Научный руководитель

Доктор технических наук, профессор

Сухомлин Владимир Александрович

Москва – 2002 г.

Содержание:

ВВЕДЕНИЕ 5

Актуальность работы 5

Стандартизация языка Си++ 6
Проект тройного стандарта 7
Компилятор Си++ 9
Компилятор переднего плана 9
Генератор кода 10

Цель работы 10

Личный вклад автора 11

Научная новизна 12

Практическая ценность и реализация 13

Структура работы 13

Апробация работы 14

ГЛАВА 1. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ 15

1.1 Введение 15

1.2 Отношение наследования 17

1.3 Граф подобъектов 23

1.4 Правило доминирования 32

1.5 Доступность базовых классов и членов класса 41

1.6 Использование отношений между классами в компиляторе 43

1.7 Структуры данных для хранения информации о классах 45

1.8 Заключение. Выводы главы 1 48

ГЛАВА 2. РЕАЛИЗАЦИЯ КЛАССОВ 49

2.1 Введение 49

2.2 Объекты классов 50

2.3 Члены класса 53

2.4 Механизмы времени компиляции 54

2.4.1 Обозначения 55
2.4.2 Построение списка всех виртуальных функций 55
2.4.3 Выявление чистых виртуальных функций и абстрактных классов 59
2.4.4 Подбор основного базового класса 60
2.4.5 «Раскладка» класса 60
2.4.6 Создание таблицы смещений виртуальных базовых классов 63
2.4.7 Создание таблицы смещений виртуальных базовых классов для подобъектов базовых классов 65
2.4.8 Создание таблицы виртуальных функций 68
2.4.9 Создание таблиц виртуальных функций для подобъектов базовых классов 69
2.4.10 Пример генерации таблиц виртуальных функций для базовых классов 73
2.4.11 Создание объектов класса type_info 74

2.4.12 Оптимизация таблиц виртуальных функций 75
2.4.13 Структуры данных, используемые при обработке класса и генерации таблиц 77
2.5 Механизмы времени выполнения 78
2.5.1 Создание и уничтожение объектов классов 79
2.5.2 Преобразования указателей на классы 80
2.5.3 Статические члены класса 82
2.5.4 Доступ к нестатическим членам классов 82
2.5.5 Вызов функций-членов класса 83
2.5.6 Указатели на члены и функции-члены классов 84
2.5.7 Преобразования указателей на члены и функции-члены классов 85
2.5.8 Реализация операции typeid 86
2.5.9 Реализация оператора dynamic_cast 87
2.5.10 Реализация определения типов исключительных ситуаций 89
2.6 Заключение. Выводы главы 2 90

ГЛАВА 3. ПОИСК ИМЕН 92

3.1 Введение 92
3.2 Особенности поиска имен в языке Си++ 94
3.3 Реализация механизма поиска имен в одной области действия 103
3.3.1 Традиционный способ поиска имен и его недостатки 104
3.3.2 Использование общей хеш-таблицы 106
3.3.3 Использование «перевернутой» таблицы 107
3.3.4 Алгоритмы добавления и поиска 109
3.4 Свойства областей действия 111
3.5 Поиск неквалифицированных и квалифицированных имен 115
3.5.1 Поиск неквалифицированного имени 115
3.5.2 Поиск квалифицированного имени 116
3.6 Заключение. Выводы главы 3 118

ГЛАВА 4. РЕАЛИЗАЦИЯ СИСТЕМЫ ТИПОВ И ПРОВЕРКИ ТИПОВ 119

4.1 Введение 119
4.2 Общие требования к реализации системы типов 120
4.3 Методы реализации типов 122
4.3.1 Реализация типов с помощью типовых цепочек 122
4.3.2 Реализация системы типов при помощи таблицы 124
4.3.3 Реализация типов с помощью структур 128
4.4.4 Квалификаторы `const` и `volatile` 130
4.4 Дополнительные возможности по работе с типами 131
4.5 Реализация системы типов 132
4.6 Преобразования типов 134
4.7 Реализация проверки типов для операторов и выбора наилучшей перегруженной функции 139
4.8 Заключение. Выводы главы 4 146

ЗАКЛЮЧЕНИЕ 147

ЛИТЕРАТУРА 149

Рисунки и диаграммы:

Глава 1.

Рис. 1-1 Пример графа наследования.. 19
Рис. 1-2. Примеры путей.. 20
Рис. 1-3. Пример множества V(s,k).. 22
Рис. 1-4. Пример построения графа Sub*(D)... 24
Рис. 1-5. Пример полного графа *Sub(D)*... 25
Рис. 1-6. Рост размера графа подобъектов 31
Рис. 1-7. Пример доминирования.. 34
Рис. 1-8. Отсутствие доминирования для базовых классов. 40
Рис. 1-9. Структуры данных... 47

Глава 2.

Рис. 2-1. Пример смещений подобъектов... 52
Рис. 2-2. Пример неоднозначности виртуальной функции. 57
Рис. 2-3. Пример разрешения неоднозначности в производном классе.............. 57
Рис. 2-4. Различные виды неоднозначностей... 58
Рис. 2-5. Пример точности оценки сложности построения списка виртуальных функций. 59
Рис. 2-6. Пример изоморфизма подграфов сохраняющего смещения ребер. 63
Рис. 2-7. Пример неэффективности оптимизаций таблиц смещения виртуальных базовых классов.. 67
Рис. 2-8. Пример частичного доминирования виртуальных функций.................. 69
Рис. 2-9. Пример построения таблиц. 73
Рис. 2-10. Пример неэффективности оптимизаций таблиц виртуальных функций.76
Рис. 2-11. Различные варианты возможных преобразований `dynamic_cast`......... 88
Рис. 2-12. Отношение доступности и оператор dynamic_cast............................. 88

Глава 3.

Рис. 3-1. Классическая схема поиска идентификаторов 105
Рис. 3-2. Применение общей хеш-таблицы 106
Рис. 3-3. Использование "перевернутой" таблицы 108
Рис. 3-4. Допустимые вложения областей действия 111
Рис. 3-5. Взаимодействие различных частей механизма поиска имен. 117

Глава 4.

Рис. 4-1. Типы как лес ориентированный к корню. 127
Рис. 4-2. Функциональные типы как DAG.. 127
Рис. 4-3. Таблица типов как хеш-таблица... 129
Рис. 4-4. Таблица типов как таблица с атласом. 130
Рис. 4-5. Облегченный вариант реализации квалификаторов 131
Рис. 4-6. Диаграмма классов реализации системы типов 132
Рис. 4-7. Взаимодействие функций проверки типов. 146

Введение

Актуальность работы

В настоящее время одним из перспективных и экономически оправданных подходов к развитию информационной индустрии является создание информационных технологий (ИТ) и реализующих их систем (ИТ-систем) на принципах открытости. Основными свойствам открытых систем являются переносимость (программ, данных, пользовательских окружений), интероперабельность (сетевая взаимосвязь и совместное использование ресурсов и данных компонентами распределенных систем), масштабируемость (эффективность функционирования в широких диапазонах характеристик производительности и ресурсов). Достижимость этих качеств возможна лишь на основе высокого уровня стандартизованности интерфейсов ИТ-систем и поддерживающих их платформ.

При всем кажущемся благополучии в области построения открытых ИТ-систем узким местом остается, казалось бы, давно решенная проблема - стандартизация языков программирования и их библиотечных окружений, составляющих основную часть прикладного интерфейса (в терминологии международных стандартов - интерфейса на границе прикладной платформы и прикладной программы или API).

Особый интерес к стандартизации языков программирования и их окружений обусловлен прежде всего быстрым развитием аппаратных средств вычислительной техники и высокой стоимостью создания нового программного обеспечения в сравнении с переносом на новые архитектуры существующих программных комплексов, созданных с использованием ЯВУ. Следует отметить, что поколения языков программирования меняются значительно медленнее, чем поколения аппаратных средств.

Другим важным преимуществом всеохватывающей стандартизации языков программирования является потребность в квалифицированных разработчиках программного обеспечения, для которых частое изменение языков программирования является болезненным процессом, требующим серьезной переподготовки.

Стандартизация языка Си++

В эру объектно-ориентированных технологий одним из основных инструментальных языков построения открытых ИТ-систем становится язык Си++ ([1], [25], [26]). С целью обеспечения высокого уровня переносимости программ, поддержки объектно-ориентированной технологии проектирования ИТ-систем ([28], [29]), бесшовной интеграции языков программирования с языками баз данных следующего поколения ([35], [36]), в настоящее время в рамках ISO (подкомитет 22) осуществляется интенсивная работа по развитию и стандартизации базового языка объектно-ориентированного программирования Си++, а также набора объектно-ориентированных и шаблонных библиотек [38], [39]. Одним из основных результатов этой работы является выход в 1998 году международного стандарта на язык Си++ [1]. После публикации стандарта рабочая группа по стандартизации продолжает активно работать над подготовкой следующей версии стандарта, в который планируется расширить как сам язык, так и стандартную библиотеку языка. В язык планируется включить также новые возможности, появившиеся в языке Си после принятия второй версии стандарта этого языка [2], [3]. В стандартную библиотеку языка планируется включить существенную часть библиотеки Adaptive Communication Environment (ACE) ([32], [33]), которая пользуется большой популярностью у разработчиков и отличается продуманным дизайном. Еще до принятия первой версии стандарта Си++ в него была включена спецификация библиотеки STL ([37], [38]) также развивавшейся достаточно долго самостоятельно, вне процесса стандартизации языка. Включение этой библиотеки потребовало изменений в спецификации самого языка, особенно механизма шаблонов.

Слабым местом процесса стандартизации Си++ является отсутствие каких-либо спецификаций проверки реализаций на конформность стандарту [8], [9]. Это особенно хорошо заметно, если сравнивать стандартизацию языков программирования Ада ([4], [6]) и Си++.

Стандарт также не затрагивает таких вопросов, как создание инструментального окружения для языка Си++, аналогичных, например стандарту ASIS [5], частично стандартизующему инструментальное окружение для языка Ада.

Проект тройного стандарта

В последнее время наблюдается тенденция значительного повышения цен на компиляторы, особенно заметное для пользователей, работающих на более мощных, чем обычные персональные компьютеры, системах - рабочих станциях. Помимо материальных потерь, которые несет Россия ежегодно на массовых покупках этого вида продукта, большая опасность монополизации данного рынка малым числом зарубежных фирм представляется, в частности, для организаций, ответственных за решение задач, имеющих стратегическое значение. Зависимость от зарубежных поставщиков, невозможность переноса их компиляторов на оригинальные отечественные образцы вычислительной техники, делает неконкурентными отечественные разработки, отрицательно сказывается на уровне технологий, в частности, кросс-технологий, применяемых при создании многих технических систем [50]. Таким образом, отсутствие качественных отечественных компиляторов, соответствующих стандартному определению языка Си++, оснащенных стандартным набором шаблонных библиотек и предусматривающих переносимость и/или адаптацию их на широкий спектр машинных архитектур в значительной степени сдерживает развитие отечественной вычислительной техники и прикладных технологий, делает их неконкурентоспособными на международном рынке.

Проект тройного стандарта 3С++ ([72]) нацелен на разработку перспективной методологии систем программирования тройного стандарта, оригинальных методов и алгоритмов построения переносимой и адаптируемой к машинным архитектурам системы программирования для объектно-ориентированного языка Си++, реализующей стандарт языка, полный набор стандартных шаблонных библиотек, технологический аттестационный комплекс для проверки на соответствие стандарту компилятора и его версий [47-49]. Таким образом, система 3С++ обеспечит высокий уровень переносимости (на уровне исходного текста) создаваемого программного обеспечения открытых систем.

Основные задачи проекта можно сформулировать следующим образом:

- Разработать концепцию создания и сопровождения систем программирования тройного стандарта, обеспечивающих реализацию стандарта входного языка и стандарта его библиотечного окружения, а также содержащих аппарат аттестации систем программирования на соответствие их языковому стандарту. Данный аппарат обеспечивает устойчивость систем тройного стандарта в условиях развития стандарта языка.

- Исследовать и разработать высокоэффективные алгоритмы и структуры данных для реализации на их основе переносимого компилятора объектно-ориентированного языка Си++ (с адаптируемой генерацией кода), соответствующего полному стандарту языка. Создать базовый компилятор объектно-ориентированного языка Си++, соответствующий стандарту языка, удовлетворяющий требованию параметризации кодогенерации для настройки его на широкий спектр машинных архитектур, а также требованию переносимости компилятора (на уровне исходного текста) на различные платформы.
- Исследовать и разработать высокоэффективные алгоритмы и структуры данных для реализации на их основе объектно-ориентированных и шаблонных библиотек, входящих в стандартное описание Си++. Создать полный набор объектно-ориентированных и шаблонных библиотек, входящих в стандартное описание языка.
- Исследовать и разработать методы, высокоэффективные алгоритмы и структуры данных, вспомогательные инструментальные средства для реализации на их основе пакета программ для аттестации компиляторов Си++ на их соответствие стандарту. Создать пакет программ для аттестации компиляторов Си++ на их соответствие стандарту.

В 1996-97 гг. проект поддерживается Грантом РФФИ 96-01-01449.

Диссертационная работа выполнялась в НИВЦ МГУ в контексте проекта тройного стандарта и завершилась в 1999 году реализацией компилятора переднего плана для языка Си++ для операционных сред SunOS, Solaris и Win32. Входной язык компилятора соответствует международному стандарту [1]. Благодаря оригинальным структурным решениям и алгоритмам, созданный компилятор обладает производственными характеристиками, сравнимыми с имеющимися для этих платформ зарубежными компиляторами ([54], [55]).

Также в контексте проекта тройного стандарта выполнялись работы по реализации важнейших компонент Стандартной Библиотеки Си++ ([1], [39]) и разработке пакета программ для аттестации компилятора на соответствие исходному описанию. Реализация тестового пакета основывалась на оригинальном подходе и методах, разработанных сотрудниками НИВЦ МГУ [47-49], и подтвердила практичность предложенных методов аттестации компиляторов. Созданная тестовая система обеспечила достаточно полное покрытие языкового многообразия и позволила как протестировать

разрабатываемый компилятор, так и выявить многочисленные ошибки в существующих компиляторах.

Ниже рассматриваются отдельные компоненты компилятора, реализованного в рамках работы над проектом тройного стандарта.

Компилятор Си++

Важнейшей задачей при разработке компилятора с языка Си++, предпринятой в рамках проекта системы программирования тройного стандарта, являлось обеспечение максимально полного соответствия его входного языка стандартному определению Си++ [1]. Разработка компилятора проводилась на фоне независимо идущего процесса стандартизации, который протекал весьма интенсивно (новые версии проекта стандарта появлялись примерно один раз в квартал) и сопровождалась постоянными модификациями как синтаксиса, так и семантики языка.

Компилятор в целом можно рассматривать как композицию следующих процессоров, каждый из которых реализован в виде независимой программы:

1. Препроцессор Си++;
2. Компилятор переднего плана Си++ (front-end compiler);
3. Генератор кода.

Построение компилятора в виде отдельных подсистем достаточно традиционно (в частности, для среды UNIX [19]). Помимо очевидного выигрыша по времени при одновременной разработке трех компонент, а также упрощения тестирования и отладки, указанная структура обеспечивает более легкую перенастраиваемость системы в целом, позволяя, например, путем добавления новых генераторов кода переносить компилятор на различные платформы [85].

Компилятор переднего плана

Эта компонента является центральной частью всего компилятора Си++. Она воспринимает на входе текст программы на Си++ (в терминах стандарта - translation unit), формируя на выходе файл с семантически эквивалентным представлением этой программы на промежуточном языке. Первая версия компилятора генерировала промежуточный код, который можно трактовать как "обобщенный ассемблер". За основу промежуточного представления принят формат Джонсона, предложенный для компилятора Portable C [63]. Более новая версия генерирует промежуточный код в оригинальном формате TTT, содержащем средства для представления деревьев функций, семантических таблиц и типов [85]. Промежуточное представление, аналогичное TTT, было реализовано и в

первоначальной версии компилятора, но не было оформлено в виде отдельного модуля, а также отсутствовала возможность сохранения промежуточного представления ТТТ в виде файла. В настоящее время этот формат расширяется в сторону поддержки большего числа языков программирования [87-88].

Компилятор переднего плана построен по традиционной однопроходной схеме. На первой фазе трансляции осуществляется лексический и синтаксический анализ исходного текста, на второй фазе выполняется большой массив семантических проверок и преобразований.

Фазы трансляции и некоторые детали реализации лексического и синтаксического разбора подробно рассмотрены в [85].

Генератор кода

В первой версии компилятора использовался генератор объектного кода для платформ Sun3 и Sun Sparc, разработанный для компилятора Potable C, а также разработан оригинальный генератор кода для микропроцессора NM6403 ([67], [68]). Генераторы кода, как и другие компоненты компилятора, представляют собой независимые программы, которые легко могут быть заменены на эквивалентный по интерфейсу генератор для некоторой другой платформы.

Для второй версии компилятора разрабатывается виртуальная машина, воспринимающая промежуточное представление ТТТ как входной язык [83], [84].

Цель работы

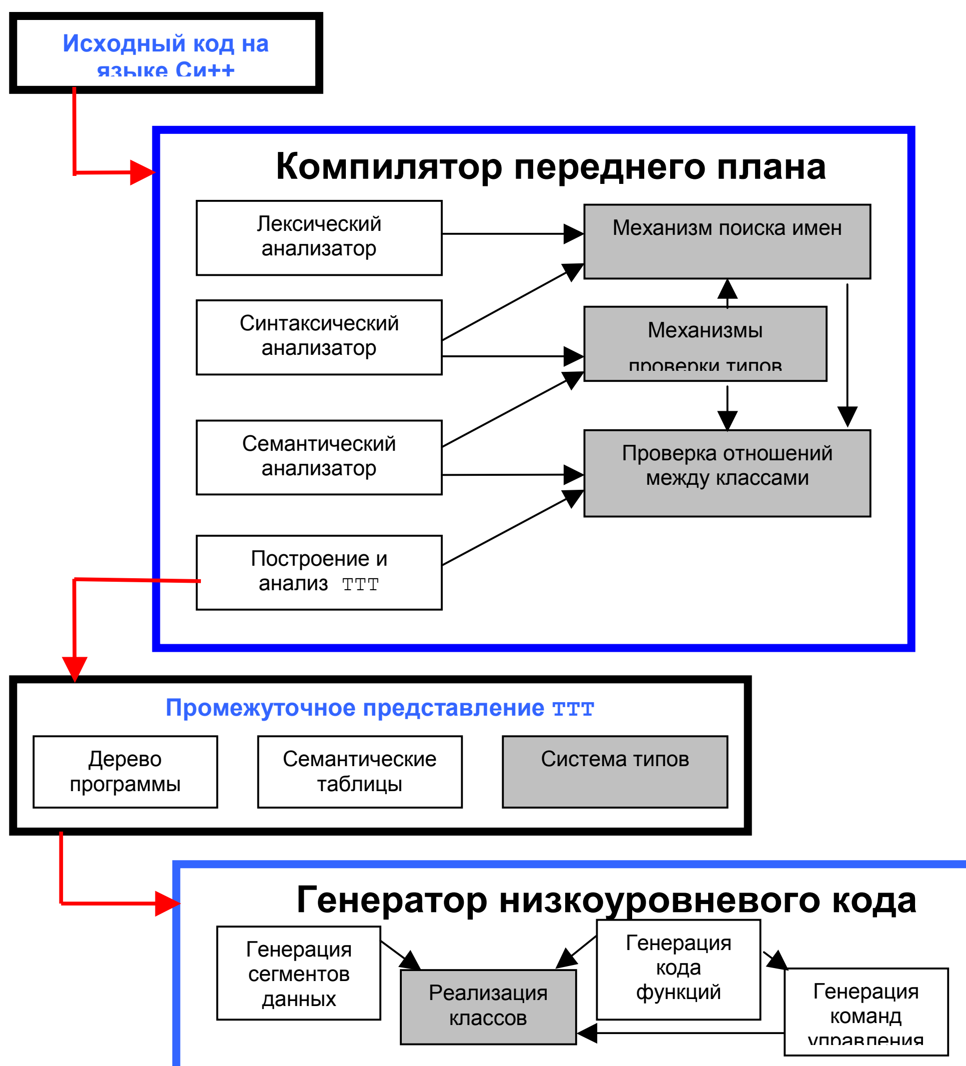
Целью данной работы является:
исследование, разработка и программное воплощении в производственном компиляторе методов и алгоритмов реализации важнейших механизмов языка С++ таких, как контроль типов, механизмы классов, наследования, виртуальных функций, поиска имен.

В соответствии с этой целью были определены следующие задачи:

1. Исследование семантических свойств механизма классов языка Си++, построение формальной модели, семантически адекватной этому механизму, и разработка на ее основе эффективных методов и алгоритмов поддержки в компиляторе семантики механизма классов, включая механизмы наследования и виртуальных функций языка;
2. Исследование семантических свойств механизма поиска имен в языке Си++, разработка формализованной модели механизма поиска имен, а также разработка основанных на ее свойствах эффективных методов и алгоритмов поддержки в компиляторе семантики механизма поиска имен.

3. Исследование семантических свойств системы типов языка Си++, разработка методов и алгоритмов поддержки в компиляторе семантики системы типов языка Си++.
4. Разработка алгоритмов генерации кода, реализующих семантику периода выполнения механизмов классов, наследования и виртуальных функций языка Си++.
5. Программная реализация разработанных в диссертационной работе методов и алгоритмов в производственном компиляторе языка Си++.

Следующий рисунок демонстрирует место использования проведенных исследований в общей структуре созданного компилятора. Детально структура рассматриваемого компилятора рассмотрена в работе [85]. В этом разделе лишь рассматривается местоположение рассматриваемых механизмов в общей структуре компилятора. На приведенной ниже схеме цветом фона выделены механизмы, рассматриваемые в этой работе.



Личный вклад автора

Работы по созданию компилятора переднего плана языка Си++ были начаты в НИВЦ МГУ в 1993г. под руководством проф. Сухомлина. Автор работы, будучи студентом, и позже аспирантом механико-математического факультета МГУ принимал участие в разработке с самого начала. Основным направлением работы являлась разработка алгоритмов семантического анализа и генерации машинно-независимого промежуточного кода. Также автор частично принимал участие в разработке препроцессора Си++, алгоритмов лексического анализа и поиска идентификаторов. Автор принимал непосредственное участие в создании генератора машинного кода для процессора NM6403 ([79], [81], [82]), частей стандартной библиотеки и тестового набора.

Научная новизна

1. Проведен детальный анализ системы типов одного из самых сложных современных языков программирования и предложен эффективный метод реализации типов, применимый также и к другим языкам программирования.
2. Построена и исследована формализованная модель системы классов языка Си++. В результате исследований построены алгоритмы превосходящие опубликованные ранее другими авторами.
3. Разработаны алгоритмы реализации механизма классов и механизма виртуальных функций, применимые при реализации языков поддерживающих объектно-ориентированное программирование.
4. Построена формализованная модель, на основе которой реализованы алгоритмы поиска имен.
5. Реализован единственный в России препроцессор и компилятор полного стандарта Си++.

Практическая ценность и реализация

На момент завершения работы над диссертацией все описанные в ней механизмы реально проверены в созданном компиляторе. На основании проведенных исследований создан полный компилятор для архитектур Sun Sparc и российского микропроцессора NM6403 (НТЦ "Модуль ", [67], [68]). Система программирования, созданная для процессора NM6403, прошла этап тестирования и практически используется в НТЦ "Модуль".

Рассматривался вопрос о применении компилятора переднего плана и тестового набора, созданного в процессе работы над проектом 3С++, в составе программного обеспечения системы "Эльбрус-2000".

На основании разработанного компилятора переднего плана фирмой “Интерстрон” (Москва) создается система программирования на Си++, ориентированная на цели обучения. В рамках этой разработки создаются виртуальная машина и редактор связей, использующие промежуточное представление ТТТ.

Структура работы

В первой главе диссертации описывается механизм классов Си++ и отношения между классами. Построена формализованная модель классов, приводятся основные алгоритмы выявления и проверок отношений между классами. Также обсуждаются вопросы эффективной реализации этих механизмов.

Во второй главе описывается реализация механизмов классов, виртуальных функций, а также некоторых операций Си++ на промежуточном языке низкого уровня. В качестве формальной модели используются модели, построенные в первой главе.

В третьей главе описывается механизм поиска имен. Выделяются ключевые особенности поиска имен в языке Си++. Строится формализованная модель для поиска идентификаторов с использованием графа наследования, после чего описываются алгоритмы и структуры данных, реализующие поиск имен.

В четвертой главе описывается система типов языка Си++, ее ключевые особенности и сложность по сравнению с другими языками программирования. Описывается метод реализации типов в компиляторе и промежуточном представлении ТТТ, а также рассматриваются алгоритмы проверки типов.

Апробация работы

Основные результаты, полученные в процессе разработки и представленные в настоящей работе, изложены в печатных публикациях с участием автора [69-78, 90], а также публикациях [79-85] и докладывались на научных конференциях и семинарах:

1. Ломоносовские чтения в МГУ в 1994, 1995, 1998гг.
2. Конференция "Индустрия программирования". Москва, 1996.
3. Конференция "Развитие и применение открытых систем". Нижний Новгород, 1997.
4. Конференция “Теоретические и прикладные проблемы информационных технологий”, ВМиК МГУ, Москва 2001.

Глава 1. Отношения между классами

1.1 Введение

Язык программирования Си++ поддерживает понятия объектно-ориентированного программирования (ООП). Механизм классов Си++ является основой для поддержки таких понятий ООП, как классы, инкапсуляция, скрытие данных и наследование ([28], раздел 2.2).

В настоящее время язык Си++ является одним из самых популярных языков программирования, которые условно называют языками объектно-ориентированного программирования. Например, второе издание известной книги "Объектно-ориентированный анализ и проектирования с примерами приложений" Гради Буча [27], [28], в отличие от первого издания, целиком ориентировано на Си++.

Идеи, лежащие в основе механизмов поддержки объектно-ориентированного программирования в расширении языка Си, автор языка Си++ Бьярн Страуструп изложил в статье [23]. В ней рассматриваются некоторые концепции, которые принято считать составными частями ООП и которые, с точки зрения автора языка, являются наиболее важными с точки зрения явной поддержки на уровне языка программирования.

Сам термин "класс" является термином объектно-ориентированного программирования ([28], гл. 3). Отношения наследования интерфейса и наследования реализации в языке Си++ отображаются как различные варианты единого механизма наследования. Скрытие данных адекватно отображается с использованием средств ограничения прав доступа в Си++ и дополняется отношениями доступности для классов. Понятие инкапсуляции отображается в языке Си++ использованием механизма виртуальных функций.

В то же время не все понятия ООП явно поддержаны в языке Си++, в частности из 9 типов отношений между классами, описанных в [28, раздел 3.4], в Си++ явно поддерживаются только отношения наследования. Другие типы отношений между классами, такие как агрегирование, также могут быть выражены на языке Си++, но для их реализации используются языковые средства, предназначенные не только для реализации отношений между классами. Например, отношение агрегирования между классами выражается в Си++ с

помощью члена класса типа "указатель на класс", а кратные отношения - с помощью массивов указателей на классы.

Си++ является языком со строгой проверкой типов на этапе компиляции. Это свойство языка несколько сужает свободу программиста при реализации объектных программ по сравнению с такими языками программирования, как Smalltalk [10], где отсутствует проверка типов и интерфейсов классов на этапе компиляции, и сообщения диспетчеризуются и проверяются на этапе выполнения программы. Как правило, платой за большую свободу программиста и отсутствие строгой типизации на этапе компиляции являются высокие накладные расходы на передачу и проверку сообщений во время выполнения.

Ограничения, накладываемые строгой проверкой типов, частично компенсируются поддержкой обобщенного программирования ([37]). Такие механизмы основаны на понятии параметрического полиморфизма ([40], [41]) и представлены поддержкой шаблонов функций и классов ([85] глава 4, [1] глава 14). Шаблоны классов сами не являются классами, хотя их синтаксис во многом повторяет синтаксис описания классов. Большую часть правил семантической проверки и даже часть правил синтаксической проверки сложно применять к шаблонам как таковым, и стандарт языка не требует таких проверок ([85] раздел 4.10, [1] глава 14). Но все семантические правила работы с классами также справедливы по отношению к классам, полученным по шаблонам. Описанные в этой главе механизмы не используются при обработке непосредственно шаблонов, но используются при обработке их настроек. Построенная ниже формальная модель для классов и алгоритмы работы с классами применимы как для классов, непосредственно описанных в исходном тексте программы, так и для классов, полученных в результате настройки шаблонов, но используются фактически на разных этапах работы компилятора ([85], раздел 1.1).

Механизм позднего связывания (later binding), необходимый для поддержки понятия инкапсуляции, представлен в Си++ механизмом виртуальных функций ([1], раздел 10.3). Семантика языка ограничивает свободу программирования произвольных объектно-ориентированных моделей, так как требует строгой спецификации интерфейсов виртуальных функций на уровне базовых классов. В языке Си++ также отсутствует явная поддержка понятия "интерфейс класса". Для описания интерфейса класса, как правило, используется класс, имеющий виртуальные функции, который затем используется как базовый класс для других классов, реализующих этот интерфейс. Для более явной поддержки понятия

интерфейсов классов в Си++ существуют механизмы "чистых виртуальных функций" и "абстрактных классов” ([1], раздел 10.4).

В языке Си++ поддерживается множественное наследование [25], которое хотя и усиливает выразительные средства и гибкость механизма классов, но значительно усложняет семантику по сравнению с языками программирования, поддерживающими только единичное наследование, такими как Java [89]. В этой главе приводится модель для формализации свойств множественного и тесно связанного с ним виртуального наследования. С помощью той же формальной модели даются определения однозначности базовых классов, двух типов неоднозначностей, порождаемых множественным наследованием (неоднозначностей имен и неоднозначностей членов), доступности для базовых классов и членов классов.

Далее в этой и следующей главе приводится описание реализации механизма классов в компиляторе, которая во-первых, адекватно отражает семантику Си++ и, с другой стороны, максимально использует особенности языка в целях генерации эффективного кода.

В этой главе будут рассматриваться:

- отношения наследования, доступности и однозначности между классами, для этих отношений будет построена формализованная модель.
- метод реализации этих отношений в компиляторе.

1.2 Отношение наследования

В этом разделе рассматривается отношение наследования и строится формальная модель для отношения наследования и граф наследования, рассматриваются свойства этого графа и приводятся примеры.

В языке Си++ классы объявляются последовательно и могут быть вложенными. Для определенности будем полагать, что имя класса объявлено (declared), как только в процессе разбора компилятор встретил конструкцию вида `class A` и класс описан (defined), как только встретилась закрывающая описание класса скобка.

Множество объявленных в единице трансляции классов линейно упорядоченно отношением лексического следования окончаний их описаний. Для этого отношения между классами введем обозначение $A \rightarrow B$, означающее, что класс A был описан до класса B. По семантике отношения очевидно, что

отношение $\rightarrow$ является антисимметричным. Для определенности будем считать отношение $\rightarrow$ антирефлексивным.

Основным отношением между классами является отношение наследования. Следуя сложившейся традиции, будем изображать это отношение между классами при помощи направленного графа - *графа наследования Inh(C, >)*, где *C* - *множество классов* в единице трансляции, отношение > - отношение непосредственного наследования, которое определено на всех парах элементов *C*. Отношение непосредственного наследования устанавливается в процессе синтаксического разбора объявления класса и устанавливается только в том случае, когда наследование объявлено явно. Будем обозначать *A>B* в том случае, когда *B* является непосредственным базовым классом для класса *A*. На графе *Inh* такое отношение будем изображать дугой идущей от производного класса *A* к базовому *B*. По семантике языка отношение наследования *A>B* может быть установлено только если $B \rightarrow A$. Также по семантике языка для каждого класса его базовые классы не должны повторяться. Таким образом, нет необходимости приписывать ребрам графа *Inh* кратность. Очевидно, что граф *Inh* является направленным ациклическим графом, а отношение > является антирефлексивным и антисимметричным. Отношение > является ограничением отношения $\rightarrow$.

Отношение > может быть нетранзитивным. Введем общее отношение наследования $\Rightarrow$ как транзитивное и рефлексивное замыкание отношения >. По определению графа *Inh* $A \Rightarrow B$ тогда и только тогда, когда вершина *B* достижима из вершины *A*. Так как отношение $\Rightarrow$ является рефлексивным, оно не является ограничением $\rightarrow$.

Для каждой пары вершин *A, B* такой, что *A>B* определено отношение *virt(A,B)*, которое также устанавливается по исходному тексту разбираемой единицы трансляции и означает дополнительный атрибут отношения наследования - виртуальность. На графе *Inh* ребра, такие, что *virt(A,B),* будем обозначать пунктирными стрелками, в противном случае - сплошными стрелками. Такие ребра будем называть *ребрами виртуального наследования*.

Например, следующему фрагменту исходного текста

```
class A {
};
class B: A {
};
class C: virtual A {
};
```

```
class D: B, C {
};
```

соответствует граф наследования

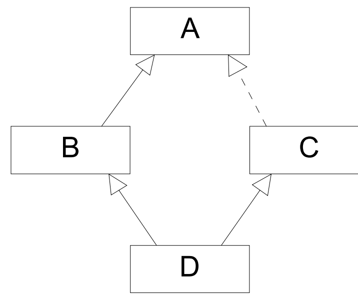


Рис. 1-1 Пример графа наследования

Ниже в основном будет использоваться понятие *пути по вершинам*. Для определенности будем считать, что каждая вершина достижима по вершинам из самой себя. Также будет использоваться понятие достижимости ребра из вершины: будем говорить что *ребро (B,C) достижимо из вершины A,* если его начальная вершина *B* достижима по вершинам из вершины *A*. Кроме этого будет использоваться понятие пути из множества вершин в вершину: *путем из множества вершин M в вершину A* будем называть произвольный путь из вершины $B \in M$ в вершину *A*. *Циклами* в графе будем называть только нетривиальные циклы, то есть циклы, состоящие более чем из одной вершины.

Пусть *Inh(C,>)* - граф наследования. Пусть *A, B* - вершины графа *Inh,* и $A \Rightarrow B$. Пусть $p = \{A=X_1, X_2,..X_k=B\}$ - некоторый путь по вершинам от вершины *A* к вершине *B*. В графе *Inh* может быть несколько путей от *A* к *B*. Назовем участок пути $\{X_i..,X_k=B\}$ *фиксированным*, если для любого ребра (X_j, X_{j+1}) $I<=j<k$ верно $virt(X_j, X_{j+1})$ и либо $j=1$, либо $virt(X_{i-1}, X_i)$. Фиксированный участок пути *p* будем обозначать *fix(p)*.

Пусть *P* - множество всех путей из вершины *A* в вершину *B*. Рассмотрим множества путей, имеющих равные фиксированные части. Введем отношение *S(a,b)* - если *a* и *b* принадлежат *P*, и *fix(a)=fix(b)*.

Утверждение 1. Отношение *S* является отношением эквивалентности.

Рефлексивность отношения *S* следует из однозначности определения фиксированной части пути.

Симметричность и транзитивность отношения *S* следует из однозначности определения фиксированной части пути и симметричности и транзитивности отношения равенства для путей по вершинам.

Таким образом, отношение *S* является отношением эквивалентности на множестве путей в графе *Inh*. Класс эквивалентности путей, содержащий данный путь *p,* будем обозначать как *[p]*, а вершину X_i, в которой начинается *fix(p),* будем обозначать как *[p]**.

На следующем рисунке приведен пример путей и их фиксированных частей.

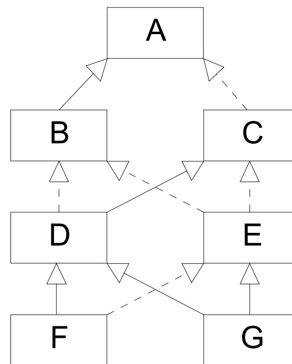


Рис. 1-2. Примеры путей.

В приведенном выше примере фиксированная часть пути *p=(G,E,B,A)* *fix(p)=(B,A),* и для этого пути *[p]* = B*. Для пути *q=(G,D,B,A)* справедливо *fix(q)=(B,A)*. Таким образом, пути *p* и *q* принадлежат к одному классу эквивалентности путей. Путь *r=(G,E,C,A)* имеет фиксированную часть *fix(r)=A* и принадлежит к другому классу эквивалентности.

Содержательное значение *[p]** в случае *[p],* заканчивающегося ребром виртуального наследования, – класс, соответствующий вершине *[p]*,* используется как виртуальный базовый класс для некоторых классов. Следует отметить, что если из вершины класса *F* доступна вершина *[p]** для некоторого пути *p,* то из этого не следует, что класс *[p]** является виртуальным базовым классом *F*, так как путь из *F* в *[p]** может и не содержать ребер, для которых верен предикат *virt*. Если быть более точным, то класс *[p]** является виртуальным базовым классом класса *F* в том и только том случае, если для последнего члена *(x, [p]*)* пути по ребрам из вершины *F* в вершину *[p]** верно *virt(x,[p]*)*. Таким образом, виртуальность наследования не может быть выражена как атрибут базового класса. В приведенном выше примере для пути *p=(G,E,C) [p]*=C*. В то же время путь *(F,D,C)*

не содержит ребер виртуального наследования, и содержательно *C* не является виртуальным базовым классом класса *F*.

Утверждение 2. Два различных пути *p* и *q* в графе *Inh,* не содержащих ребер виртуального наследования, не эквивалентны.

Очевидно, что если *p* и *q* не содержат ребер виртуального наследования, то *fix(p)=p* и *fix(q)=q*.

Пусть *V(s,k)* - множество вершин *v* графа *Inh* таких, что *v* достижима из *s* и для некоторого пути *p* из *s* в *k* верно *[p]*=v*.

Содержательный смысл множества *V* - это множество всех, включая опосредованные, виртуальных базовых классов класса *s,* производных от класса *k*, имеющие класс *k* в качестве невиртуального базового класса.

Утверждение 3. О числе неэквивалентных путей.
Число классов эквивалентности по отношению *S* путей графа *Inh* из вершины *s* в вершину *k* равно сумме числа классов эквивалентности путей из вершины *s* в вершину *k,* не содержащих ребер виртуального наследования, и суммы по всем вершинам *v* из *V(s,k)* числа классов эквивалентности путей из *v* в *k,* не содержащих ребер виртуального наследования.

Предыдущее утверждение говорит о том, что все пути из вершины *s* в вершину *k,* не содержащие ребер виртуального наследования, неэквивалентны. Остальные пути из *s* в *k* содержат ребра виртуального наследования. Следовательно, для любого такого пути *fix(p)* не совпадает с *p*. Для любого такого пути существует вершина *v=[p]*,* достижимая из *s,* но не совпадающая с *s*, следовательно, принадлежащая множеству *V(s,k)*.

Докажем, что любые два неэквивалентных пути *p* и *q* из вершины *s* в вершину *k,* содержащие ребра виртуального наследования, порождают неэквивалентные пути из множества *V* в вершину *k,* не содержащие ребер виртуального наследования. Рассмотрим *fix(p)* и *fix(q)*. По определению эти два пути не содержат ребер виртуального наследования. Они начинаются в вершинах *[p]** и *[q]** соответственно, и заканчиваются в вершине *k*,

следовательно, они являются путями, ведущими из вершины, принадлежащей множеству *V,* в вершину *k*. Так как пути *p* и *q* неэквивалентны, *fix(p)* не совпадает с *fix(q)*. То есть они порождают различные пути из множества *V* в *k,* не содержащие ребра виртуального наследования.

В обратную сторону. Пусть *p'* и *q'* - различные пути из множества *V* в вершину *k,* не содержащие ребер виртуального наследования. По определению множества *V* существуют пути *p* и *q* из *s* в *k* такие, что *p'* и *q'* являются их частями, и ребра этого пути, входящие в вершины *[p]** и *[q]*,* являются ребрами виртуального наследования. Необходимо доказать, что такие пути *p* и *q* являются неэквивалентными. Что с очевидностью следует из того, что *fix(p)=p'*, а *fix(q)=q'*.

Таким образом, любой класс эквивалентности путей из *s* в *k,* содержащий ребра виртуального наследования, порождает путь из *V(s,k)* в *k,* не содержащий ребер виртуально наследования, и наоборот.

Для завершения доказательства необходимо применить предыдущее утверждение, которое в данном случае говорит о том, что любые два различных пути, не содержащие ребер виртуального наследования из множества *V* в *k*, неэквивалентны.

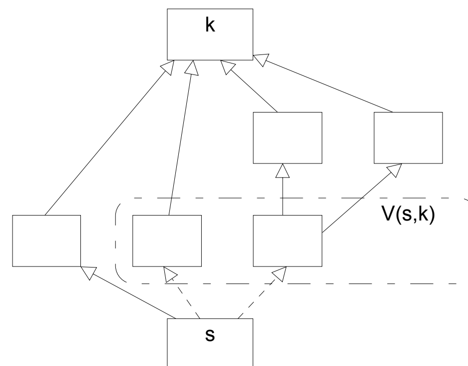


Рис. 1-3. Пример множества V(s,k).

Как видно из приведенного выше примера, из одной вершины множества *V(s,k)* в вершину *k* может вести несколько путей.

Утверждение 4. Число классов эквивалентности по отношению *S* путей из вершины *s* в *k* равно сумме по всем вершинам объединения множеств *V(s,k)* и *{s}* числа путей из множества в вершину *k*, не содержащих ребер виртуального наследования.

Утверждение, очевидно, вытекает из двух предыдущих утверждений.

Пусть *W(s)* - объединение по всем *k* таким, что $s \Rightarrow k$ множеств *V(s,k)*. Содержательный смысл множества *W* - это множество всех, включая опосредованные, виртуальных базовых классов класса *s*.

Следствие 1 о числе неэквивалентных путей.
Число классов эквивалентности по отношению *S* путей из вершины *s* в *k* равно сумме по всем вершинам объединения множества *W(s)* и *{s}* числа путей из вершин множества в вершину *k*, не содержащих ребер виртуального наследования.

Это следствие, очевидно, вытекает из предыдущего утверждения, если принять во внимание, что если для некоторого *w,* принадлежащего *W(s),* вершина *k* недостижима по ребрам невиртуального наследования, то число таких путей равно 0.

1.3 Граф подобъектов

Граф подобъектов используется как модель для изучения структуры объектов класса. Он строится на основании графа наследования. В этом разделе рассматриваются свойства этого графа и доказывается несколько утверждений о его структуре. Как будет показано ниже, размер графа подобъектов (число его вершин и ребер) может расти экспоненциально по сравнению с размером графа наследования. По этой причине граф подобъектов в явном виде в процессе трансляции не строится, и будет использоваться ниже только как удобная формализованная модель для описания структуры подобъектов и исследования их свойств.

Теперь мы готовы дать определение графа подобъектов. Граф подобъектов будет строиться не для всех классов из множества *C*, а для каждого конкретного его элемента *s* отдельно. Такой граф будем обозначать как *Sub(s)*.

Пусть множество *L* - счетное множество, из которого будут строиться вершины графа. Каждому элементу *l* из *L,* задействованному в построении графа подобъектов, будет ставиться в соответствие некоторый класс (элемент множества *C*) с помощью функции m_s. Индекс *s* при функции *m* будет опускаться в тех случаях, когда речь идет об одном конкретном графе *Sub(s)*. Вершины *k* такие, что $m_s(k)=K$, будем называть *вершинами класса K*.

Пусть *s* - некоторая вершина из *C*. Пусть *fp* - все пути в графе *Inh,* начинающиеся в вершине *s,* и не проходящие ни через одно ребро *(a,b)* такое, что *virt(a,b)*, то есть для любого пути *x* из *fp* верно *x=fix(x)*. Значит все пути из *fp* принадлежат различным классам эквивалентности из *P*.

Шаг 1. Возьмем первый элемент l_0 из *L* и определим для него $m(l_0)=s$. Элемент l_0 множества *L* станет *корнем* графа *Sub(s)*, то есть вершиной, из которой будут достижимы все остальные вершины *Sub(s)*.

Сначала построим часть графа *Sub(s),* которую будем обозначать как *Sub*(s)*. Эта часть графа является деревом и строится рекурсивно.

Если $D=\{d_1..d_n\}$ - множество вершин *C* таких, что $l_0>d_i$ и не $virt(l_0,d_i)$, то рекурсивно построим графы $Sub^*(d_i)$ из незанятых вершин множества *L* и достроим ребра из вершины l_0 к корням деревьев $Sub^*(d_i)$. Функцию m_s для этих подграфов определим как объединение функций m_d.

Пример подграфа *Sub*(s)*:

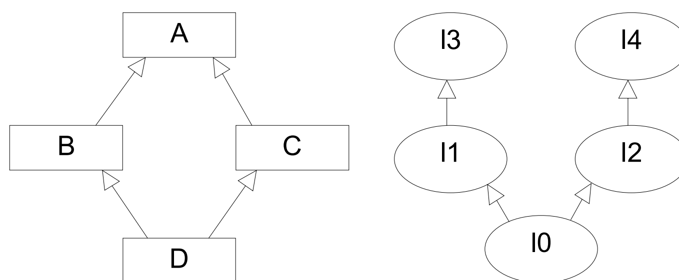


Рис. 1-4. Пример построения графа Sub*(D).

В приведенном примере $m(l_0)=D$, $m(l_1)=B$, $m(l_2)=C$, $m(l_3)=m(l_4)=A$.

Полный граф *Sub(s)* строится добавлением к графу новых вершин и ребер.

Шаг 2. Пусть *Q(s)* - множество путей в графе *Inh,* начинающихся в вершине *s* и содержащих хотя бы одно ребро, помеченное как ребро виртуального наследования. Пусть *Q*(s)* - множество классов эквивалентности путей из *Q(s)*, таких что $A\Rightarrow[p]^*$. Для каждого элемента *[p]* из *Q*(s)* построим граф *Sub*([p]*).*

Добавим построенные подграфы в граф *Sub(s).* Функцию m_s для этих подграфов определим как объединение функций *m[p]**.

Шаг 3. Для всех вершин *x*, таких что *m(x)>[p]** и *virt(m(x),[p]*)* добавим в граф *Sub(s)* ребро, соединяющее *x* с корнем подграфа *Sub*([p]*)*. Очевидно, что после завершения этого шага и, следовательно, всего процесса построения граф *Sub(s)* является связным, и все вершины этого графа доступны из его корня l_0.

Процесс построения графа *Sub(s)* является рекурсивным, но в силу того, что граф *Inh* является ациклическим и на каждом шаге рекурсивного построения мы переходим вдоль ребер графа *Inh,* это построение не может зациклиться.

Очевидно, что этот граф является направленным, но не обязательно является деревом.

Пример построения графа *Sub(s)*.

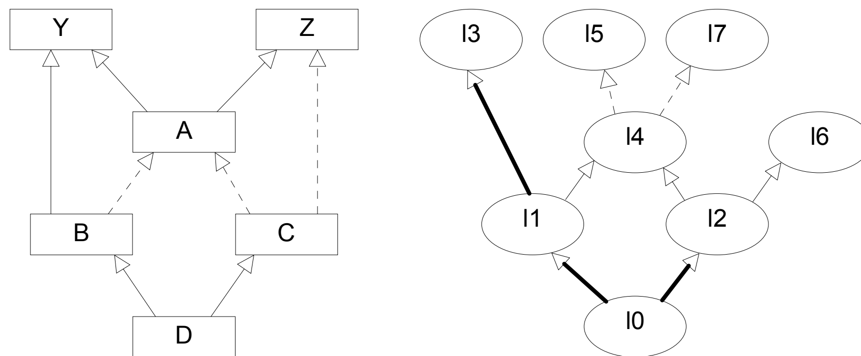


Рис. 1-5. Пример полного графа *Sub(D)*.

В приведенном примере жирными стрелками выделены ребра, которые являются ребрами графа *Sub*(D),* а пунктирными стрелками - ребра добавленного графа *Sub*(A)*.

В приведенном примере $m(l_0)=D$, $m(l_1)=B$, $m(l_2)=C$, $m(l_3)=m(l_5)=Y$, $m(l_4)=A$, $m(l_6)=m(l_7)=Z$.

Построение графа *Sub(s)* закончено. Рассмотрим некоторые свойства графа *Sub(s)*.

Утверждение 5. Если *(l,k)* - ребро графа *Sub(s)*, то в графе *Inh* существует ребро *(m(l), m(k))*.

Рекурсивное построение графа *Sub(s)* определяет способ доказательства этого утверждения. Ребра графа *Sub(s)* строятся либо в результате

добавления ребра в результате рекурсивного применения построения, либо в результате соединения корня рекурсивно полученного графа к уже построенной его части. Во втором случае по алгоритму построения ребро *(l,k)* добавляется только в том случае, если *(m(l), m(k))* является ребром графа *Inh*. В первом случае применяем данное утверждение рекурсивно к подграфу, в результате присоединения которого это ребро получено. Очевидно, применяя рекурсивно это утверждение к очередной части конечного подграфа, мы рано или поздно придем ко второму случаю.

Следствие 2. Граф *Sub(s)* является направленным ациклическим графом.

Если в графе *Sub(s)* существует цикл по вершинам $(p_1, p_2, .. p_n)$, то в графе *Inh* существует цикл по вершинам $(m(p_1), m(p_2), .. m(p_n))$.

Утверждение 6. Каждому ребру *(a,b)* графа наследования *Inh,* такому что вершина *a* достижима из вершины *s*, соответствует по крайней мере одно ребро *(k,l)* графа *Sub(s)* такое, что *m(k)=a* и *m(l)=b*.

В процессе рекурсивного построения графа *Sub(s)* в граф добавляются ребра, соответствующие всем непосредственным базовым классам класса *s*. Более строго: для любого *c,* такого что *s>c,* в граф *Sub(s)* по построению добавляется ребро *(k,l)* такое, что *m(k)=s*, *m(k)=c*. Для завершения доказательства остается рассмотреть случай, когда *s* не совпадает с *a* и *virt(a,b)*: в этом случае утверждение следует из второго шага построения графа *Sub(s)*, на котором добавляются ребра, соответствующие этому случаю.

Утверждение 7 о существовании изоморфного подграфа.
Если вершина *r* достижима из вершины *s*, то в графе *Sub(s)* (с отображением m_s вершин графа *Sub(s)* в вершины графа *Inh*) найдется подграф *R* и отображение *I* подграфа *R* на граф *Sub(r)* (с отображением m_r), такие, что *I* является изоморфизмом графов, и для всех вершин *k* подграфа *R* верно $m_s(k)=m_r(I(k))$. Ограничение *I** изоморфизма *I* на *Sub*(r)* является изоморфизмом деревьев.

В процессе рекурсивного построения графа *Sub(s)* процесс рано или поздно доходит до любой вершины, достижимой из вершины *s*. На этом этапе строится граф *Sub*(r)*. Возможно, такой подграф будет строиться несколько раз. Достаточно зафиксировать хотя бы одно такое построение. Допустим, на некотором шаге построен один подграф *Sub*(r)*, который мы будем обозначать *R**. Его изоморфизм *I** на подграф графа *Sub(r),* удовлетворяющий условиям утверждения, строится естественным образом. Очевидно, *I** является изоморфизмом деревьев, удовлетворяющим условиям утверждения.

Затем, на втором шаге построения для каждой вершины, достижимой из вершины *s*, достраиваются подграфы, соответствующие виртуальному наследованию. При этом, так как любое ребро виртуального наследования, достижимое в графе *Inh* из вершины *r,* также достижимо из вершины *s,* соответствующие виртуальному наследованию ребра и подграфы при построении графа *Sub(r)* будут также строиться для графа *Sub(s)*.

Если на втором шаге построения *Sub(s)* некоторый класс эквивалентности *[p]* из *Q*(s)* эквивалентен относительно отношения *S* классу *[p']* из *Q*(r)*, то продолжим отображение *I** для подграфа *Sub*([p]*)*, а при добавлении ребер, входящих в его корень из вершин графа *R*,* на шаге 3 будем также достраивать отображение *I**.

Полученное в результате дополненное отображение *I** и будет отображением *I*.

По построению отношения *I* оно удовлетворяет условиям утверждения.

Определим функцию *sub(s,k)* на парах вершин графа *Inh* как число вершин *x* графа *Sub(s)* таких, что $m_s(x)=k$. А функцию *sub*(s,k)* - как число вершин *x* графа *Sub*(s)* таких, что $m_s(x)=k$. Из этого определения, очевидно, следует, что если $A \Rightarrow B$, то $sub(A,B)>0$, иначе $sub(A,B)=0$. Очевидно также, что $sub^*(s,k) \leq sub(s,k)$.

Следствие 3. Пусть вершина *k* достижима из вершины *l* графа *Inh*. Тогда для произвольной вершины *x* этого графа $sub(k,x) \leq sub(l,x)$.

Заметим, что так как в процессе построения графа *Sub(s)* построение подграфа *Sub*(r)* может происходить несколько раз, в графе *Sub(s)* может существовать несколько подграфов, изоморфных *Sub(r)*.

Утверждение 8. *sub*(s,k)* равно числу путей графа *Inh*, ведущих из вершины *s* в вершину *k* и не содержащих ребер виртуального наследования.

Утверждение следует из того, что для любого пути $p=(a_1=s, a_2, \dots a_n=k)$ при рекурсивном построении графа *Sub*(s)* строятся подграфы $Sub^*(a_i)$ для каждой вершины этого пути. Таким образом, в графе *Sub*(s)* будет существовать путь $q=(b_1, b_2, \dots b_n)$ такой, что $m(b_i)=a_i$. При этом для произвольного $0<i<n$ существует одна и только одна вершина *c* такая, что (b_i,c) является ребром графа *Sub*(s)* и $m(c)=a_{i+1}$. Граф *Sub*(s)* является деревом, поэтому два различных пути в этом графе не могут вести в одну вершину. Следовательно, путь *p* по вершинам графа *Inh* однозначно определяет один путь из корня графа в вершину *a* такую, что *m(a)=k*.

Обратно, если для некоторой вершины *a* графа *Sub*(s)* верно *m(a)=k*, то рассмотрим путь $q=(b_1, b_2, \dots b_n=a)$ из корня дерева в вершину *a*. Путь $p=(m(b_1)=s, m(b_2), \dots, m(b_n)=m(a)=k)$ будет путем в графе *Inh* из вершины *s* в вершину *k*, при этом этот путь не содержит ребер виртуального наследования.

Таким образом, утверждение доказано.

Заметим, что в условиях предыдущего утверждения все пути, ведущие из вершины *s* в вершину *k*, являются неэквивалентными относительно отношения *S*.

Следствие 4. *sub*(s,k)* равно числу классов эквивалентности *[p]* относительно отношения *S* путей графа *Inh* из вершины *s* в вершину *k* таких, что *[p]* = s*.

Если путь *p* в графе *Inh* не содержит ребер виртуального наследования, то *fix(p)=p* и, следовательно, *[p]*=s*. После этого следствие вытекает из предыдущего утверждения.

Утверждение 9. Если в вершину *b* графа *Sub(s)* входит несколько ребер $((a_1,b), (a_2,b), \dots (a_n,b))$, то для всех ребер $((m(a_1),m(b)), (m(a_2),m(b)), \dots (m(a_n),m(b)))$ верно $virt(m(a_i), m(b))$, то есть, все эти ребра является ребрами виртуального наследования.

По построению все части графа *Sub(s),* построенные рекурсивно, являются графами *Sub*(r)* для некоторого класса *r*. Графы *Sub*(r)* по построению являются деревьями. В результате добавления ребра, соответствующего невиртуальному наследованию, на первом шаге построения (построении графа *Sub*(s)*) всегда добавляется только одно ребро, соединяющее вершину графа *Sub*(r)* с уже существующей вершиной графа *Sub(s)*. Несколько ребер могут соединить корень графа *Sub*(r)* только на третьем шаге построения, при котором всегда добавляются ребра, соответствующие ребрам виртуального наследования графа *Inh*.

Утверждение 10. Основное утверждение о числе подобъектов.
sub(s,k) равно числу классов эквивалентности относительно отношения *S* путей графа *Inh,* ведущих из вершины *s* в вершину *k*.

Частично это утверждение уже доказано для части *Sub*(s)* графа *Sub(s)* (следствие 4). Остается лишь рассмотреть случай путей, содержащих ребра виртуального наследования. Пусть $p=(a_1=s, a_2, .. a_n=k)$ - некоторый путь в графе *Inh,* содержащий ребро виртуального наследования. Тогда *[p]** не совпадает с *s,* и в процессе построения графа *Sub(s)* на шаге 2 был построен граф *Sub*([p]*)*. По утверждению о существовании изоморфного подграфа, число его вершин *b* таких, что *m(b)=k,* равно числу неэквивалентных по отношению *S* путей в графе *Inh* из вершины *[p]** в вершину *k*. Воспользовавшись утверждением о числе неэквивалентных путей, получаем, что каждому классу эквивалентности путей соответствует одна вершина *a* такая, что *m(a)=k*.

Очевидно, что любой вершине *a*, такой что *m(a)=k,* соответствует некоторый путь в графе *Sub(s)*, а значит и путь в графе *Inh*, который, в свою

очередь, принадлежит некоторому классу эквивалентности путей, который, по доказанному выше, однозначно определяет вершину графа *Sub(s)*.

Осталось доказать, что два различных класса эквивалентности путей приводят к различным вершинам графа *Sub(s)*. Часть этого утверждения уже доказана для путей, не содержащих ребер виртуального наследования (утверждение 4). В случае наличия виртуального наследования и двух классов эквивалентности путей *[p]* и *[q]* необходимо рассмотреть два случая:

Случай $[p]^* \neq [q]^*$. В этом случае пути не могут привести к одной вершине *a,* так как *Sub([p]*)* и *Sub([q]*),* построенные на шаге 2, по построению не содержат общих вершин.

Случай $[p]^*=[q]^*$. В этом случае можно применить утверждение о существовании изоморфного подграфа *Sub([p]*)*, а так как *fix(p)* для произвольного *p* из *[p]* не содержит ребер виртуального наследования, можно повторить доказательство утверждения о числе подобъектов в графе *Sub*([p]*)*.

Таким образом, утверждение доказано.

Следствие 5. Если в графе наследования *Inh* заменить все ребра виртуального наследования ребрами невиртуального наследования, то для произвольных вершин *a* и *b* *sub(a,b)* не уменьшится.

При замене всех ребер виртуального наследования на ребра невиртуального наследования неэквивалентные пути останутся неэквивалентными. В то же время все классы эквивалентности распадутся на одноэлементные классы эквивалентности.

Утверждение 11. Для произвольных вершин *x* и *s* графа *Inh* $sub(s,x) \leq 2^{n-1}$, где *n* - число вершин графа *Inh,* достижимых из вершины *s*. Эта оценка точна.

Рассмотрим следующий пример графа наследования.

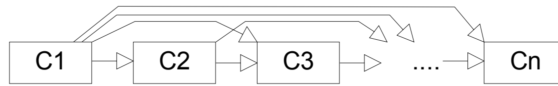


Рис. 1-6. Рост размера графа подобъектов

Приведений выше граф наследования является максимальным в том смысле, что число путей из вершины C_1 в вершину C_n в этом графе максимальное по сравнению со всеми возможными графами наследования на n вершинах. Действительно, произвольный ациклический граф на n вершинах изоморфен некоторому подграфу этого графа. Все пути этого графа неэквивалентны, так как в этом графе нет ни одного ребра виртуального наследования. Таким образом, исходя из предыдущего утверждения, число вершин x графа $Sub(C_1)$ таких, что $m(x)=C_n$, равно числу путей из вершины C_1 в вершину C_n. Рассмотрим путь $p=(y_1=C_1,y_2,..y_k=C_n)$. При фиксированном y_2 число путей такого вида равно числу путей из вершины y_2 в вершину C_n. Суммируя по всем возможным y_2, получаем, что число путей $f(n)$ из вершины C_1 в вершину C_n равно $f(n)=f(n-1)+f(n-2)+..f(1)$. При этом $f(1)=1$, так как существует единственный путь по вершинам, состоящий из одной вершины в ациклическом графе, состоящем из единственной вершины. Выпишем это равенство для n и $n-1$.

$f(n)=f(n-1)+f(n-2)+..+f(1)$

$f(n-1)=f(n-2)+..f(1)$.

Вычитая второе из первого, получим

$f(n)-f(n-1)=f(n-1); f(n)=2f(n-1)$.

Учитывая то, что $f(1)=1$, получаем $f(n)= 2^{n-1}$.

Таким образом, получена оценка сверху. Из приведенного примера видно также, что эта оценка точна.

Построенный граф подобъектов $Sub(s)$ является моделью строения объекта класса и его подобъектов, учитывающей особенности виртуального наследования и множественного наследования Си++. Основным вопросом, на который мы будем искать ответ, используя граф подобъектов, является нахождение количества подобъектов базового класса в объекте производного класса. Содержательно значение функции $sub(s,k)$ есть число подобъектов класса k в объекте класса s.

Вопрос о числе подобъектов является весьма важным в процессе проверки семантики программы на языке Си++. Сначала на основании полученных выше

результатов рассмотрим способ быстрого вычисления этой функции, а также функции *sub*(s,k)*, если уже известны значения функций *sub(l,k)* и *sub*(l,k)* для всех вершин *l,* достижимых из вершины *s.* Это содержательно означает, что значения этих функций известны для всех базовых классов класса *s*.

Допустим *A*⇒*B*. Количеством подобъектов класса *B* в объекте класса *A* является число вершин *X* графа *Sub(A*) таких, что *m(X)=B*. Как было показано выше, число вершин графа *Sub(A)* может расти экспоненциально относительно числа вершин графа *Inh,* достижимых из вершины *A*. Поэтому мы будем искать метод быстрого вычисления *sub(A,B),* не требующий построения графа *Sub(A)*.

Алгоритм построения графа *Sub(A)* подсказывает способ вычисления числа подобъектов, не использующий явное построение графа.

Из доказанных выше утверждений о числе подобъектов следует, что

$$sub(A,B) = \sum_{I \in W(A) \cup \{A\}} sub^*(I,B)$$

$$sub^*(A,B) = \sum_{I: A > I \,\&\, \neg virt(A,I)} sub^*(I,B)$$

$$\forall A \; sub^*(A,A) = sub(A,A) = 1$$

Первое равенство вытекает из утверждения 10 и следствия 1. Второе равенство вытекает из утверждений 8 и 4. Третье из приведенных равенств вытекает из того, что существует единственный путь по вершинам из вершины в саму себя, и такой путь не содержит ребер виртуального наследования.

Таким образом, можно быстро находить значения функции *sub(A,B),* если известны множество *W(A)* виртуальных базовых классов класса *A* и значения функций *sub*(I,B)* для всех непосредственных базовых классов *I* класса *A (I:A>I)* и элементов множества *W(A)*.

Теперь можно дать эквивалентное определение для понятия однозначности базового класса, данное в определении языка Си++ [1, раздел 10.2] неформально. Класс *B* является *однозначным базовым классом* класса *A,* если *sub(A,B) = 1*.

1.4 Правило доминирования

В этом разделе дается определение и рассматриваются свойства доминирования имен в языке Си++. В качестве формальной модели для исследования свойств доминирования используется граф подобъектов построенный в предыдущем разделе.

Классы Си++ имеют собственные области видимости, в которых могут быть определены именованные сущности, такие как члены классов и именованные

типы. Эти сущности также доступны по своим именам в производных классах. Так как одно и то же имя может именовать различные сущности в различных классах, в производных классах имена могут быть неоднозначны. Например:

```
class A1 {
      int x;
};
class A2 {
      char x(int);
};
class B: A1, A2 {
      void f() { x; }
};
```

В данном примере использование имени *x* в классе *B* неоднозначно. В месте использования имени *x* доступно два имени из базовых классов *A1* и *A2*, что вызывает неоднозначность.

Для определения неоднозначности использования имен, определенных в базовых классах, используется отношение доминирования имен. В стандарте языка это отношение определяется следующим образом:

При использовании имени *x* в классе *A* имя *x,* определенное в базовом классе *B,* доминирует над именем *x,* определенным в базовом классе *C,* если любой подобъект базового класса *C* объекта класса *A* является также подобъектом базового класса *B* ([1], раздел 10.2)

Использование имени, определенного в базовом классе, в контексте производного класса является однозначным (семантически корректным) только в том случае, когда это имя доминирует над всеми определениями этого имени в других базовых классах.

В работе [42] рассматривается связь доминирования с графом подобъектов. Однако авторы статьи не пытались избегать явного построения графа подобъектов, число вершин которого, как было показано выше, может расти экспоненциально по отношению к числу вершин графа наследования, что означает, что предложенный там формализм приводит к экспоненциально сложному алгоритму. В работе [51] этот недостаток устранен. В этой же статье предложен алгоритм, который работает на основании графа наследования, приписывая ребрам графа дополнительные атрибуты. Этот алгоритм имеет полиномиальную сложность $O(n^5)$ по отношению к числу вершин графа. Ниже будет приведен алгоритм определения доминирования, работающий без приписывания ребрам графа дополнительных атрибутов и лишь опосредованно,

через функции *sub* и *sub**, апеллирующий к графу наследования. Сложность этого алгоритма $O(n^2)$ по отношению к числу вершин графа.

Переформулировав данное выше определение доминирования в терминах графа подобъектов, получим:

При использовании имени *x* в классе *A* имя *x*, определенное в базовом классе *B*, *доминирует* над именем *x*, определенным в базовом классе *C*, в графе *Sub(A)*, если для любой вершины *c*, такой, что $m_A(c)=C$, существует путь из корня графа *Sub(A)* в вершину *c*, проходящий через вершину *b* такую, что $m_A(b)=B$.

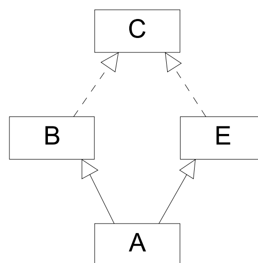


Рис. 1-7. Пример доминирования.

В данном примере имя *x* определенное в классе *B*, доминирует над именем *x*, определенным в классе *C*. Также на этом примере можно видеть отличие отношения доминирования для имен и стандартного отношения доминирования из теории графов.

Так как граф *Sub(A)* может не являться деревом, как видно из приведенного выше примера, в графе могут существовать пути в вершину *c*, не проходящие ни через одну вершину *b*, такую, что *m(b)=B* и в то же время имя, определенное в классе *B*, будет доминировать над именем, определенным в классе *C*.

Как видно из приведенного выше определения доминирования для имен, оно зависит только от набора классов, для которых оно применяется. Переформулируем это наблюдение в виде утверждения.

Утверждение 12. Для любых двух имен *x* и *y*, определенных в классах *B* и *C*, имя *x* определенное в классе *B*, доминирует над именем *x* в классе *C* тогда и только тогда, когда имя *y*, определенное в классе *B* доминирует над именем *y*, определенным в классе *C*.

Таким образом, от отношения доминирования имен можно перейти к отношению доминирования для классов.

Будем говорить, что в графе *Sub(A)* класс *B доминирует* над классом *C*, если для любой вершины *c* графа *Sub(A)* такой, что *m(c)=C*, существует путь из корня графа в вершину *c*, проходящий через вершину *b* такую, что *m(b)=B*.

Из этого определения и сформулированного выше утверждения следует

Следствие 6. Если класс *B* доминирует над классом *C* в графе *Sub(A),* то любое имя *x,* определенное в классе *B,* доминирует над именем *x* в классе *C*.

Следующие несколько утверждений устанавливают связь между определенной выше в разделе 1.3 функцией *sub* и отношением доминирования.

Утверждение 13. Пусть даны три вершины графа *Inh* такие, что $A \Rightarrow B \Rightarrow C$ и любой путь, не содержащий ребер виртуального наследования из вершины *A* в вершину *C* проходит через вершину *B*. Тогда *sub*(A,C)=sub*(A,B)sub*(B,C)*.

Граф *Sub*(A)* является деревом. Для любого пути в графе *Sub*(A)* существует его образ по отображению m_A в графе *Inh,* являющийся путем, и этот образ не содержит ребер виртуального наследования. Любой путь из корня графа в вершину *c* такую, что *m(c)=C* проходит через вершину *b* такую, что *m(b)=B*. Из этого следует, что все вершины *c* такие, что *m(c)=C*, добавлены к графу *Sub*(A)* в результате добавления подграфов, изоморфных графу *Sub*(B)*.

Число путей из корня графа *Sub*(A)* в корни таких подграфов по определению равно *sub*(A,B)*. Число путей из корня каждого подграфа, изоморфного *Sub*(B)*, в его вершины *c* такие, что $m_A(c)=C$, по определению равно *sub*(B,C)*. Таким образом исчерпываются все пути в вершины класса *C*. Остается показать, что все такие пути различны. Это следует из того, что подграфы, изоморфные *Sub*(B),* не содержат общих вершин, а все вершины класса *C* являются вершинами этих подграфов.

Утверждение 14. Пусть даны три вершины графа *Inh* такие, что $A \Rightarrow B \Rightarrow C$, и любой путь из вершины *A* в вершину *C* проходит через вершину *B,* и любой путь из вершины *B* в вершину *C* не содержит ребер виртуального наследования. Тогда *sub(A,C)=sub(A,B)sub*(B,C)*.

В условиях утверждения в процессе построения графа *Sub(A)* все вершины класса *C* добавляются в граф в результате добавления подграфов, изоморфных *Sub*(B)*. Все такие графы не имеют общих вершин, и их число

равно числу их корней *sub(A,B)*, а каждый такой подграф содержит *sub(B,C)=sub*(B,C)* вершин класса *C*.

Утверждение 15. Пусть даны три вершины графа *Inh* такие, что $A \Rightarrow B \Rightarrow C$. Если любой путь из вершины *B* в вершину *C* не содержит ребер виртуального наследования и верно равенство *sub(A,C)=sub(A,B)sub*(B,C)*, то класс B доминирует над классом *C* в графе *Sub(A)*.

sub(A,C) равно числу классов эквивалентности путей из *A* в *C* в графе *Inh*, и каждый такой класс эквивалентности однозначно определяет вершину класса *C* в графе *Sub(A)*. Так как все пути из вершины *B* в вершину *C* не содержат ребер виртуального наследования, то для всех путей *p* из вершины *A* в вершину *C*, проходящих через вершину *B*, *fix(p)* является продолжением некоторого пути из *B* в *C*.

Допустим, вершина *B* не доминирует над вершиной *C*, что по определению означает, что в графе *Sub(A)* существует вершина *c* такая, что $m_A(c)=C$ и ни один путь *p* в эту вершину из корня не проходит через вершину класса *B*. Ни один из таких путей не эквивалентен ни одному пути, проходящему через вершину *B*, так как иначе его фиксированная часть *fix(p)* проходила бы через вершину *B*. Таким образом, для этой вершины найден новый класс эквивалентности путей из вершины *A* в *C*. В то же время существует *sub(A,B)sub*(B,C)* классов эквивалентности путей из вершины A в вершину *C*, проходящих через *B*. Из этого следует, что *sub(A,C)>sub(A,B)sub*(B,C)*. Получено противоречие с условиями утверждения.

Таким образом получен эффективный способ выявления доминирования для троек классов *(A,B,C)* таких, что все пути из вершины *B* в вершину *C* не содержат ребер виртуального наследования. Если приведенное выше, в условии утверждения, равенство выполнено, то вершина *B* доминирует над вершиной *C*.

Остается рассмотреть общий случай, когда существуют пути из вершины *B* в вершину *C*, содержащие ребра виртуального наследования.

Утверждение 16. Пусть даны три вершины графа *Inh* такие, что $A \Rightarrow B \Rightarrow C$. Число вершин *c* класса *C* в графе *Sub(A)* таких, что в них существует путь из корня, проходящий через вершину класса *B,* равно *(sub(A,B)-1)sub*(B,C)+sub(B,C)*.

Построим граф *Inh',* исключив из графа *Inh* все ребра виртуального наследования, содержащиеся в путях из вершины *B* в вершину *C*. Граф *Sub'(A),* построенный на основании графа наследования *Inh'* , будет изоморфен подграфу графа *Sub(A),* построенному на основании графа *Inh,* и такой изоморфизм сохраняет классы вершин. Для него будет справедливы условия предыдущего утверждения. Следовательно, для него будет справедливо равенство *sub'(A,C)=sub'(A,B)sub*'(B,C).*

Теперь рассмотрим множество *D* всех вершин *c* класса *C* графа *Sub(A),* удовлетворяющих условиям утверждения и не являющиеся образами изоморфизма. Для каждой такой вершины существует путь в *c* из некоторой вершины *b* класса *B*, образ по отображению *m* некоторого ребра *(k,l)* этого пути является ребром виртуального наследования, и образ части пути из вершины *l* в вершину *c* не содержит ребер виртуального наследования. По построению графа *Sub(A)* все пути графа *Sub(A),* содержащие ребро, образом которого в графе *Inh* является ребро *(m(k),m(l)),* проходят через вершину *l*. Значит и все пути из множества вершин класса *B* в *c* проходят через эту вершину. По условию утверждения класс *B* доминирует над классом *C*, значит для любой вершины *c* из множества *D* существует путь через такую вершину *l* и $m(l) \Rightarrow C$. Множество таких вершин *l* обозначим *L*. Так как ребро *(k,l)* является ребром виртуального наследования, *m(l)* принадлежит множеству *W(B),* определенному выше. Значит из любой вершины *b* класса *B* в графе *Sub(A)* существует путь в любую вершину *c* из множества *D*. Следовательно, для того чтобы найти мощность множества *D* достаточно рассмотреть количество вершин, достижимых из одной конкретной вершины *b* класса *B*.

Рассмотрим подграф *Sub(B)*, он будет изоморфен некоторому подграфу графа *Sub(A)*, содержащему все вершины из множества *D*. При этом прообразами изоморфизма для множества *D* будут те и только те вершины

класса C, которые не являются вершинами подграфа $Sub^*(B)$. Их число равно $sub(B,C)-sub^*(B,C)$.

Суммируя, получаем, что общее число вершин класса C равно $sub'(A,B)sub^{*\prime}(B,C)+sub(B,C)-sub^*(B,C)$. При этом, так как в графе Inh' не исключен ни один путь из вершины A в вершину B, $sub'(A,B)=sub(A,B)$, а так как не было исключено ни одной вершины подграфов $Sub^*(B)$, $sub^{*\prime}(B,C)=sub^*(B,C)$. В результате получаем, что общее число вершин класса C, удовлетворяющих условиям утверждения, равно $sub(A,B)sub^*(B,C)+sub(B,C)-sub^*(B,C)=(sub(A,B)-1)sub^*(B,C)+sub(B,C)$.

Таким образом, получен эффективный способ проверять доминирование для классов на основании значений функций *sub* и *sub**, который можно сформулировать в виде утверждения:

Утверждение 17. Основное утверждение о связи отношения доминирования для классов и функций *sub* и *sub**.
Пусть даны три вершины графа *Inh* такие, что $A \Rightarrow B \Rightarrow C$. Тогда класс B доминирует над классом C в графе $Sub(A)$ тогда и только тогда, когда выполнено равенство $sub(A,C)=(sub(A,B)-1)sub^*(B,C)+sub(B,C)$.

В одну сторону утверждение повторяет предыдущее.

В другую сторону. Допустим, что выполнено приведенное равенство. Покажем, что в графе $Sub(A)$ существует $(sub(A,B)-1)sub^*(B,C)+sub(B,C)$ вершин класса C, достижимых из корня графа по путям, содержащим вершину класса B. Таким образом, исходя из равенства, множество всех вершин класса C будет исчерпано, что будет означать, что класс B в этом графе доминирует над C.
Каждая вершина c класса C графа $Sub(A)$ однозначно определяется некоторым классом эквивалентности путей из вершины A в вершину C. Рассмотрим множество F классов эквивалентности таких путей, содержащих, по крайней мере, один путь p из A в C, содержащий вершину B. Каждый такой класс эквивалентности $[p]^* \in F$ однозначно определяет

вершину *c* класса *C,* доступную из корня через вершину класса *B*. Рассмотрим фиксированные части таких путей $fix(p) = (a_1,a_2,..a_k=C)$.

Допустим, что $a_1=A$. В этом случае *fix(p)=p*, а значит *fix(p)* содержит вершину *B*. Число путей такого вида равно *sub*(A,B)sub*(B,C).*

Допустим, что $a_1 \neq A$, и *fix(p)* содержит вершину *B*. Число путей такого вида равно $\sum_{a_1 \in W(A)} sub^*(a_1, B) sub^*(B,C) = (\sum_{a_1 \in W(A)} sub^*(a_1, B))sub^*(B,C)$. Таким образом известно число вершин класса *B* графа *Sub(A),* доступных по пути, m_A-образ которого проходит по ребру виртуального наследования.

Получаем, что $\sum_{a_1 \in W(A)} sub^*(a_1, B) = sub(A,B) - sub^*(A,B)$ и,

следовательно, число путей в этом случае равно *(sub(A,B)-sub*(A,B))sub*(B,C)*.

Последний случай $a_1 \neq A$ и *fix(p)* не содержит вершину *B*. Число путей в этом случае равно $\sum_{a_1 \in W(B)} sub^*(a_1, C) = sub(B,C) - sub^*(B,C)$.

Таким образом, суммируя число путей во всех трех случаях, получим равенство *sub*(A,B)sub*(B,C)+(sub(A,B)-sub*(A,B))sub*(B,C)+sub(B,C)-sub*(B,C) = sub(A,B)sub*(B,C)+sub(B,C)-sub*(B,C)*.
По условию утверждения оно также равно *sub(A,C)*. Таким образом, для каждой вершины класса *C* существует путь в эту вершину из корня, проходящий через вершину класса *B*, что по определению означает, что класс *B* доминирует над классом *C* в графе *Sub(A)*.

Утверждение 18. Для произвольных вершин *A* и *B* графа *Inh* таких, что $A \neq B$ и $A \Rightarrow B$, класс *A* доминирует над классом *B* в графе *Sub(A)*.

Утверждение, очевидно, вытекает из того, что для вершины *r* - корня графа *Sub(A)* верно *m(r)=A*.

Таким образом, для определения доминирующего определения имени в базовых классах, если это имя не определено в самом классе *A*, достаточно иметь список *bases(A)* всех базовых классов (классов вершин графа *Sub(A)*). Из этого списка можно выделить список *Y* классов, в которых искомое имя определено. Из этого списка можно исключить все доминируемые другими элементами списка классы, пользуясь критерием из утверждения 17, за $O(n^2)$ сравнений, использующих три вычисления функций *sub* и *sub**, где *n* - число элементов списка. Максимальное число элементов списка базовых классов равно числу вершин в графе *Inh*. Если в результате сокращения списка получен одноэлементный список, то единственный его элемент будет доминирующим, иначе, если список оказался пуст, имя не определено в базовых классах (что не всегда означает семантическую ошибку), иначе, если в списке более одного элемента, имя является неоднозначным в рамках рассмотренной модели, что всегда означает семантическую ошибку.

То, что список членов может быть не сокращаем, то есть ни один из базовых классов не доминирует над другим, можно видеть на следующем графе наследования:

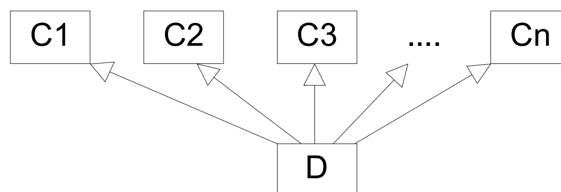


Рис. 1-8. Отсутствие доминирования для базовых классов.

Если некоторое имя определено во всех базовых классах *C1*,*C2*..*Cn*, то все эти классы будут элементами несокращенного списка, и ни один из них не доминирует над другим, что видно из того, что граф *Sub(D)* в данном случае изоморфен графу *Inh*.

Найти максимально доминирующее вхождение в список можно простым поиском максимального элемента за *O(n)* сравнений, а пустой список в результате сокращения может получиться, только если список базовых классов, в которых определено искомое имя, изначально пуст. Таким образом, для каждого успешного поиска имени требуется не более *O(n)* операций сравнения.

Модель, рассмотренная выше, не учитывает существования using-объявлений, которые подробно описываются ниже, в третьей главе диссертации. Если такие объявления учитывать, то наличие в сокращенном списке более чем

одного элемента не всегда означает семантическую ошибку, а именно, если все оставшиеся в списке имена являются именами using-объявлений и все они ссылаются на одну и ту же сущность, семантически использование имени однозначно. Проверить это можно за $O(mn)$ сравнений, где m-число сущностей в сокращенном списке, и, следовательно, $O(mn) \leq O(n^2)$.

Составить список базовых классов для класса *A* можно, если известны такие списки для всех его базовых классов и эти списки упорядочены произвольным полным отношением порядка (например отношением $\rightarrow$) за $O(n^2)$ операций, так как у класса *A* не может быть более *n* базовых классов, каждый из которых в свою очередь имеет не более *n* различных базовых классов [13, раздел 5.3.2]. Точность этой оценки можно проверить на графе наследования, приведенном выше на рисунке 1-6.

Список непосредственных базовых классов для конкретного класса также не может содержать более *n* элементов, и для его упорядочивания требуется $O(n \ln n)$ сравнений.

1.5 Доступность базовых классов и членов класса

Последним важным с точки зрения проверки семантики программы на языке Си++ отношением между классами является отношение доступности. Использование членов и других именованных сущностей, определенных в базовых классах, в объекте производного класса должно проверяться на доступность. При объявлении каждой именованной сущности в классе ей приписывается атрибут *доступность*, который может принимать одно из трех значений *public*, *protected* или *private* [85, раздел 3.5].

Различаются два контекста, из которых происходит обращение к именованной сущности класса: из самого класса, его члена или дружественной сущности и общий случай. В общем случае доступны только именованные сущности, имеющие public доступность. Из самого класса, его членов и дружественных функций доступны все именованные сущности класса. Текущий контекст, в котором происходит доступ к именованной сущности, можно определить на основании контекста компиляции [85, раздел 1.4]

Именованные сущности базовых классов также доступны как именованные сущности производных классов. В стандарте языка [1, раздел 11.2] описывается, как изменяется доступность членов базовых классов при непосредственном наследовании. Доступность наследования при непосредственном наследовании

известна из таблиц, построенных синтаксическим анализатором [85, раздел 3.5]. Эти правила сведены в следующей таблице.

Доступность наследования / Доступность сущности	Public	protected	private
public	Public	protected	private
protected	Protected	protected	private
private	недоступен	недоступен	недоступен

Табл. 1 Доступность членов классов при наследовании

Эти преобразования применяются последовательно для всего пути от производного класса к базовому, в котором определена сущность. В графе наследования может существовать несколько путей от производного класса к базовому, и по различным путям могут получаться различные значения для доступности члена базового класса. Среди различных вариантов доступности, полученных по различным путям, всегда выбирается наименее строгий, то есть, если по одному из путей именованная сущность доступна как public, в производном классе она считается доступной как public, иначе, если по одному из путей именованная сущность доступна как protected, в производном классе она считается доступной как protected, иначе, если по одному из путей именованная сущность доступна как private, в производном классе она считается доступной как private, иначе сущность считается недоступной.

Доступность самого базового класса определяется в стандарте [1, раздел 11.2] как доступность искусственно введенной в базовый класс именованной сущности с доступностью public.

Первый вопрос, который будет рассматриваться - это нахождение доступности базового класса *C* в классе *A* на основании того, что уже известна доступность этого базового класса для всех непосредственных базовых классов *B* класса *A*. Функция доступности ниже обозначаться как *Acc(X,Y)* - доступность базового класса *Y* в классе *X*. Будем считать эту функцию определенной, только если $X \Rightarrow Y$. Функцию, возвращающую доступность для непосредственного базового класса, будем обозначать как *BAcc(X,Y)*. Множество различных значений этой функции упорядочим следующим образом: *public<protected<private<not_accessible*. Пользуясь приведенной выше таблицей можно предложить следующий алгоритм.

```
if (A=C) {
    Acc(A,A)=public;
    stop;
```

```
    }
    acc=not_accessible;
    foreach (B: A>B) {
        if (sub(B,C)>0) {
            if (BAcc(A,B)>Acc(B,C))
                bacc = BAcc(A,B);
            else if (Acc(B,C)==private)
                continue foreach;
            else
                bacc = Acc(B,C);
            if (bacc<acc) {
                acc=bacc;
                if (acc=public)
                    break foreach;

            }
        }
    }
    Acc(A,C)=acc;
```

Этот алгоритм выбирает наименее строгое ограничение по всем путям через непосредственные базовые классы, а, следовательно, по всем путям от класса *A* к *C*.

Доступность именованной сущности *x* базового класса *C* в производном классе *C* теперь можно определить как максимум доступности самой сущности *x* и *Acc(A,C)*. Этот алгоритм опирается на то, что значение функции *Acc(A,B)* уже известно.

В разделе 11.5 стандарта Си++ [1] специфицированы дополнительные правила для проверки доступности protected именованных сущностей класса, которые сводятся к сравнению на совпадение классов, в контексте которых используется имя из базового класса, и класса, через который происходит обращение, который может быть специфицирован явно, как часть квалифицированного имени, так и неявно, как тип подвыражения в левой части операции доступа к члену класса.

1.6 Использование отношений между классами в компиляторе

Именованные сущности в классах доступны по их именам, и каждое использование имени сущности, определенной в классе, должно быть однозначным в смысле отношения доминирования. Иначе такое использование имени является неоднозначным и влечет за собой семантическую ошибку. Для

некоторых сущностей, таких как имена типов, имена статических членов и статических функций-членов классов однозначность имени означает отсутствие неоднозначности в их использовании, потому что эти сущности не являются частью подобъектов, а относятся к классам целиком или ко всем объектам и подобъектам этих классов. Для нестатических членов класса и нестатических функций-членов и только для них также требуется однозначность подобъекта, которую можно установить на основании однозначности конкретного базового класса с помощью функции *sub*, сравнив ее значение с 1. Каждое использование имени сущности, определенной в классе, должно именовать доступную в текущем контексте сущность.

Если в процессе лексического разбора встретилось имя, далеко не всегда возможно определить какую именно сущность оно именует. Простейший пример такого вхождения - имя, стоящее в правой части операции `->`. В выражении `p->m` определить член `m` какого класса используется можно, только если известен тип подвыражения `p`, стоящего в левой части операции, где может находиться произвольно сложное выражение. Определение типа выражения, стоящего в левой части производится на второй фазе компиляции.

В других контекстах на этапе лексического разбора на основании текущего контекста компиляции ([85], раздел 1.4) можно определить, что имя именует функцию-член класса, но невозможно определить, какую именно из множества совместно используемых функций это имя именует. Это определяется только на этапе проверки типов при выборе наиболее подходящей по типам аргументов функции, что будет рассмотрено ниже, в четвертой главе.

В некоторых случаях использование операторов преобразования и в большинстве случаев использования деструкторов их использование является неявным, то есть никак не представлено в исходном тексте программы. Такие случаи использования членов класса также подлежат проверкам на однозначность и доступность.

Семантическая проверка преобразований указателей и ссылок на классы заключается в проверке однозначности и доступности базового класса в производном и также осуществляется на второй фазе трансляции, когда известны типы преобразуемого подвыражения и тип, к которому его значение преобразуется. Тип, к которому преобразуется выражение, не всегда явно специфицируется в исходном тексте программы.

В то же время использование имен типов, определенных в классе, и имен перечислителей часто легко определяется в процессе лексического разбора. В таких случаях семантические проверки на однозначность имени и доступность выполняются на первой фазе трансляции.

Таким образом, механизмы проверки отношений между классами нельзя отнести к механизмам только первой или только второй фазы компиляции.

1.7 Структуры данных для хранения информации о классах

Определенные выше функции *sub*, *sub** и *Acc* вычисляются рекурсивно на основании значений этих функций для базовых классов. Проверка доминирования также основывается на вычислении этих функций. После того как некоторый класс полностью описан, множество его непосредственных базовых классов остается неизменным, следовательно, остается неизменным множество ребер графа наследования, ведущих из соответствующей ему вершины, а так как все его базовые классы также являются полностью описанными, остается неизменным весь подграф наследования, достижимый из соответствующей вершины. Атрибуты доступности наследования для каждого ребра наследования также не изменяется. Основываясь на этом свойстве языка, перечисленные функции для класса можно вычислить сразу после завершения его определения и значения этих функций остается неизменным до конца работы компилятора. Для того чтобы избежать многократного вычисления этих функций, заранее просчитанные значения сохраняются в структурах данных компилятора.

Для хранения дополнительной информации о классе для каждого класса, определенного в единице трансляции, создается дескриптор этого класса. Дескриптор представляет собой структуру, содержащую следующие поля, описывающие этот класс:

- *self* - ссылка на семантическое слово полного описания класса;
- *bases* - список дескрипторов базовых классов для всех базовых классов;
- *vbases* - список дескрипторов базовых классов для всех виртуальных базовых классов;
- другие поля, рассмотренные во второй главе диссертации.

Следует отметить, что для одного класса, определенного в единице трансляции, в семантических таблицах может быть создано несколько семантических слов ([85], раздел 3.5). Все такие семантические слова ссылаются на общий дескриптор класса.

Дескриптор базового класса представляет собой структуру, содержащую следующие поля:

- *self* - ссылку на семантическое слово базового класса;
- другие поля, рассмотренные во второй главе диссертации.

Дескриптор отношения между классами представляет собой структуру, полями которой являются:

- *derived* - ссылка на производный класс
- *base* - ссылка на базовый класс
- *sub_full* - значение функции *sub(derived,base)*
- *sub* - значение функции *sub*(derived,base)*
- *acc* - значение функции *Acc(derived,base)*

Дескрипторы отношений между классами сохраняются в единой для всех классов хеш-таблице. В качестве хеш-функции используется сумма хеш-значений базового и производного классов. Хеш-значение класса определяется на основе адреса его дескриптора. Время доступа к дескриптору отношений между классами определяется свойствами хеш-таблицы. Число отношений между классами может расти максимально как сумма арифметической прогрессии, то есть, как $O(n^2)$. Коллизии хеш-значений разрешаются в этой таблице с помощью метода "открытой адресации" ([13], раздел 6.4). Во избежание увеличения времени вставки и поиска при достижении коэффициентом заполнености значения *0.8* после очередной вставки строится вдвое большая хеш-таблица. Ожидаемое время поиска в такой хеш-таблице остается ограниченным константной величиной.

Непосредственно после завершения описания класса *A* для этого класса производятся следующие действия:

1. Составляется список bases всех прямых и опосредованных базовых классов. Этот список составляется слиянием списков всех базовых классов его непосредственных базовых классов с добавлением самих непосредственных базовых классов. Информация о непосредственных базовых классах берется из семантических таблиц ([85], раздел 3.5). Список базовых классов базовых классов доступен через дескрипторы базовых классов. Затем этот список сортируется, и из него удаляются повторяющиеся элементы. Как было показано выше, сложность составления такого списка $O(n^2)$.

2. Аналогично составляется список vbases всех прямых и опосредованных виртуальных базовых классов.
3. Для каждого элемента списка bases на основании приведенных выше рекурсивных формул вычисляются значения функций *sub*, *sub** и *Acc*. Результаты вычисления функций сохраняются в полях записи дескриптора отношений между классами. Запись дескриптора отношений между классами заносится в хеш-таблицу.

В дальнейшем, если для вычисления значения функций для производного класса или для выполнения семантической проверки необходимо получить значения этих функций, эти значения ищутся в хеш-таблице. Отсутствие в хеш-таблице записи означает отсутствие отношения наследования между двумя классами.

На рисунке показаны структуры данных описывающие отношения меду классами для следующей программы:

```
class A {};
class B: public A {};
class C: protected virtual A {};
class D: private virtual B, protected C {};
```

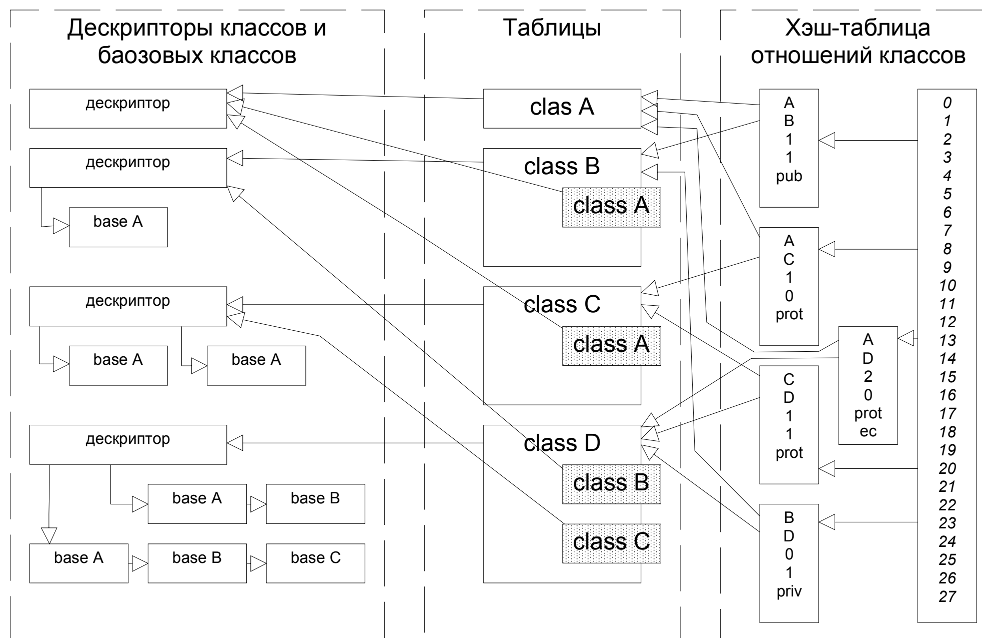


Рис. 1-9. Структуры данных.

В работе [51] также предлагается сохранять промежуточные результаты определения доминирующего определения имени в базовом классе. Такая

реализация возможна в компиляторе с использованием структур поиска имен, описанных в третьей главе диссертации. Но такая возможность не была реализована, так как был найден и реализован более эффективный алгоритм определения доминирования для классов и разработан эффективный метод поиска определения имен.

1.8 Заключение. Выводы главы 1

В этой главе рассматривались наиболее существенные с точки зрения реализации компилятора отношения между классами. Для исследования отношений между классами построены формальные модели – графы наследования и подобъектов, которые также будут использоваться в последующих главах. Понятия однозначности базовых классов и доминирования имен сведены к вычислениям достаточно простых функций, что значительно упрощает предложенные другими авторами формальные модели за счет того, что описанные механизмы не требуют построения графов подобъектов. Также описывается метод реализации проверки отношений между классами.

Основные результаты первой главы могут быть сформулированы следующим образом:

1. Дано формальное определение графа наследования. Изучены свойства наследования классов в языке Си++.
2. Построена и изучена формальная модель (граф подобъектов) строения объектов классов в Си++, учитывающая особенности языка.
3. На основании формальной модели предложен эффективный способ вычисления числа подобъектов базового класса. Дано формальное определение однозначности базового класса.
4. Дано определение доминирующего класса, основанное на формальной модели. Предложен эффективный способ определения доминирующего класса.
5. Предложен способ определения доступности базового класса и члена базового класса на основании графа наследования.
6. Описаны структуры данных компилятора, используемые в реализации семантических проверок отношений между классами.

Глава 2. Реализация классов

2.1 Введение

Данная глава продолжает описание способов реализации механизма классов в компиляторе Си++. В ней рассматривается реализация классов с точки зрения генерации низкоуровневого кода. Семантические проверки, описанные ниже, в основном связаны с проверкой семантики использования виртуальных функций.

Основой для реализации является описанная в первой главе формальная модель подобъектов. При описании механизмов времени компиляции активно используются те же структуры данных, что и в первой главе.

Часть механизмов, описанных в этой главе, нельзя полностью отнести к механизмам генерации низкоуровневого кода (`sizeof`, проверка семантических правил при использовании виртуальных функций).

Основные идеи эффективной реализации виртуальных функций кратко описаны в [22]. В этой главе за основу взят именно этот способ, использующий генерацию «таблиц виртуальных функций». Этот способ используется также и в других компиляторах Си++, так как он позволяет сравнительно легко реализовать механизм виртуальных функций на вычислительных машинах с традиционной архитектурой. Основой реализации механизмов виртуального наследования и определения типов на этапе выполнения (run-time type identification, rtti) также является генерация компилятором статических таблиц, которые являются общими для всех объектов класса и используются во время выполнения программы. При этом основной целью является максимальное упрощение генерируемого низкоуровневого кода, использующего эти таблицы, но не упрощение самих таблиц или частей компилятора их генерирующих.

Следует отметить, что при кажущейся простоте механизма таблиц виртуальных функций необходимо учитывать множество тонкостей языка. Так, в 1996 году автором был предложен пример программы, использующей одновременно виртуальные функции и виртуальное наследование, который в то время обрабатывался некорректно всеми протестированными компиляторами. Из этого был сделан вывод о том, что для описания реализации необходимо предложить максимально формализованную модель и ее описание должно быть максимально полным и аккуратным.

Целью этой главы является рассмотрение методов реализации следующих механизмов языка Си++:

- классы (с точки зрения генерации машинно-независимого промежуточного кода низкого уровня);
- множественное и виртуальное наследование;
- преобразование указателей и ссылок на классы;
- виртуальные функции;
- указатели на члены и функции-члены классов;
- преобразование указателей на члены и функции-члены классов;
- определение типов времени выполнения (run-time type identification, rtti);
- определения типов при обработке исключительных ситуаций.

2.2 Объекты классов

Объекты классов располагаются в непрерывной области памяти (*storage area*) фиксированного размера, равного для всех объектов данного класса. Ниже будет приведен алгоритм вычисления размера. Основой для него служит построение функции *sizeof(x)* (“размер”, *storage size*), дающей размер области памяти, необходимой для размещения объекта *x*. Объекту класса необходимо отводить память для хранения нестатических членов класса.

Функция *sizeof* предполагается уже определенной для всех членов класса и всех базовых классов. Кроме этого, для всех базовых классов предполагается определенной функция *own_size(x)*, дающая “собственный” размер класса (размер без учета виртуальных базовых классов). Можно считать, что размер *sizeof(C)* определяется для всего графа *Sub(C)*, а собственный размер определяется для его фиксированной части – дерева *Sub*(C)*.

Вычисление значения функции *sizeof* реально производится только один раз для каждого класса в процессе его обработки. Затем размер и собственный размер класса сохраняется в дескрипторе класса, описанном в разделе 1.7. В приведенном ниже алгоритме учтена возможность появления “скрытых” членов класса, отсутствующих в определении класса пользователем, но существенных для реализации. В каких случаях добавление “скрытых” членов необходимо, будет сказано ниже, но, забегая вперед, отметим, что для каждого конкретного класса их может появиться не более двух, и их размер известен до начала обработки класса.

Подграф *Sub*(C)* (определенный в первой главе) является деревом, и он целиком, кроме корня, распадается на подграфы, изоморфные деревьям *Sub*(B)*

для своих непосредственных базовых классов *B*. Благодаря этому свойству даже при наличии в языке множественного наследования, но при отсутствии виртуального наследования можно приписать каждому подобъекту фиксированное смещение, не зависящее от того, подобъектом какого наиболее производного класса он являются. При размещении объектов класса существенно то, что "подобъекты" базовых классов (области памяти, отведенные для размещения объектов базовых классов) имеют в точности ту же структуру, что и объекты базовых классов. Если обозначить $offset_A(B)$ смещение подобъекта однозначного базового класса *B* в объекте класса *A*, выполнялось бы равенство $offset_A(C)=offset_A(B)+offset_B(C)$ для любых трех классов $A \Rightarrow B \Rightarrow C$ таких, что $sub^*(A,B)=sub^*(B,C)=sub^*(A,C)=1$. Это позволяло бы существенно упростить реализацию, так как достаточно было бы определить смещения только для случаев непосредственного наследования. В этом случае можно было бы приписывать смещения ребрам графа *Inh*. Полное смещение подобъекта опосредованного базового класса можно бы было вычислить как сумму смещений по пути в графе *Inh*.

В случае виртуального наследования такая простая реализация невозможна. Допустим, что класс *C* является виртуальным базовым классом классов *B1* и *B2*. Было бы весьма сложно подбирать такие смещения подобъекта класса *C* в базовых классах *B1* и *B2*, чтобы для всех возможных классов *A* таких, что $A \Rightarrow B1$ и $A \Rightarrow B2$ выполнялось равенство $offset_A(B1)+offset_{B1}(C)=offset_A(B2)+offset_{B2}(A)$. При этом возникали бы дополнительные сложности, например необходимость «выравнивания» подобъектов, при котором часть отведенной для хранения объекта памяти не использовалась бы, и, что более существенно, не все классы, удовлетворяющие условиям, наложенным на класс *A*, обязаны быть описаны в одной единице компиляции. Вместо этого для виртуальных базовых классов используются нефиксированные смещения: смещение подобъекта класса *C* в объекте класса *B1* не является фиксированным, а зависит от того, в объекте какого класса *A* объект класса *B1* является подобъектом.

Дополнительные сложности возникают, когда базовый класс *C*, сам не являясь виртуальным базовым классом для *A*, является базовым классом другого виртуального базового класса *B*. В этом случае его динамическое смещение складывается из динамического смещения *B* в *A* и известного статически смещения класса *C* в *B*.

Рассмотрим следующий пример графов наследования:

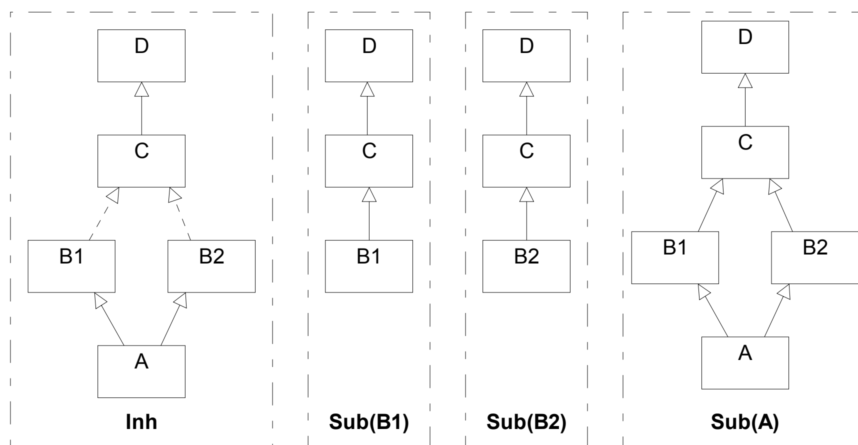


Рис. 2-1. Пример смещений подобъектов.

В приведенном примере смещения базовых классов могут быть, например такими:

$offset_C(D)=1$

$offset_{B1}(C)=1$ $offset_{B2}(C)=1$

$offset_{B1}(D)=offset_{B1}(C)+offset_C(D)=2$ $offset_{B2}(D)=offset_{B2}(C)+offset_C(D)=2$

$offset_A(B1)=1$ $offset_A(B2)=2$

$offset_A(C)=3$ $offset_A(D)= offset_A(C)+ offset_C(D)=4$

Таким образом, наиболее оптимальным представляется способ реализации, использующий простое размещение подобъектов для фиксированной части *Sub*(C)* и использующий дополнительные приемы для вычисления смещений подобъектов виртуальных базовых классов, которые, как указано выше, не могут быть вычислены статически, на этапе компиляции, если статически не известен динамический тип объекта.

Размещение в памяти объектов класса в простейших случаях полностью совпадает с размещением в памяти структур ставших классическими языков программирования Си и Pascal. Кроме этого, размещение объектов простейших классов Си++**,** в точности так же, как структур языка С - весьма существенная возможность для пользователя языка (так как язык Си++ является развитием языка Си, и разработчики языка стремились сохранить совместимость с программами на Си).

В следующих разделах будет описан алгоритм обработки класса во время компиляции. Этот алгоритм применяется к каждому классу, определенному в единице трансляции. В процессе работы алгоритма каждому нестатическому члену класса и каждому базовому классу будет приписываться их “смещение”

(*offset*) в классе. Оно означает смещение области памяти, отведенной для размещения нестатического члена класса или базового класса относительно области памяти, отведенной для размещения объекта класса целиком. Будем обозначать смещение *offset(m)*, где *m* – путь в графе *Inh* из класса С в базовый класс *B*, либо *m* является нестатическим членом класса С. Смещение не может считаться функцией от класса, так как один и тот же класс *B* может иметь различные смещения в различных своих прямых производных классах. Как было сказано в первой главе, при реализации компилятора мы избегали построения графа *Sub(C)* в явном виде, поэтому смещение, приписанное ребру или, в общем случае, пути, реально размещается в одном из полей дескриптора отношений между классами, описанного в разделе 1.7. В графе *Inh* может существовать несколько путей из вершины производного класса в вершину базового класса. Смещение подобъекта имеет смысл определять только для тех случаев, когда базовый класс является однозначным, иначе каждое обращение к члену базового класса в объекте производного будет семантически некорректным, и нет необходимости генерировать в этом случае машинный код, а, следовательно, нет необходимости в корректном определении смещений. Смещение нестатических членов класса сохраняется в их семантических словах ([85]).

Также будет определяться функция *index(l)*, заданная для элементов списков виртуальных базовых классов и элементов списков виртуальных функций.

2.3 Члены класса

В семантических таблицах компилятора ([85]) для каждого описанного класса *C* доступна информация о его членах (*class members*) *attr(C)*, невиртуальных функциях-членах (*class member functions*) *meth(C)*, виртуальных функциях членах (*virtual class member functions*) *virt(C)*. Для простоты дальнейшего изложения будем считать, что *attr(C)*, *meth(C)* и *virt(C)* являются списками. В этой главе будем считать, что операторы, определенные в классе, также являются функциями-членами класса, так как правила языка, относящиеся к функциям-членам, также распространяются на пользовательские операторы.

Существенным для дальнейшего изложения является понятие совпадения сигнатур функций. *Сигнатура функции* определяется как ее имя и типы ее явных параметров. Будем говорить, что две функции имеют одинаковую сигнатуру, если совпадают их имена и типы параметров с учетом их порядка. Тип возвращаемого значения не входит в сигнатуру. Две функции могут иметь одинаковую сигнатуру и различные типы возвращаемых значений, при этом существенным является то,

что в каждом конкретном классе не может быть определены две функции с одинаковыми сигнатурами.

Также потребуется дать понятие максимального производного класса. Каждый раз, когда создается объект класса, в тексте программы явно указывается его тип. Затем с объектом можно работать через указатели или ссылки на его подобъекты, при этом изначальный тип объекта далеко не всегда может быть выяснен на этапе трансляции. Тип объекта при его создании будем называть максимальным типом, а соответствующий типу класс – максимальным производным классом.

Введем отношение доминирования для виртуальных функций. Неформально оно изложено в разделе 10.3 стандарта Си++ [1], где говорится, что отношение доминирования повторяет с небольшими оговорками отношение доминирования для имен, с тем отличием, что при обсуждении доминирования имен говорилось о совпадающих именах, а в сейчас мы говорим о совпадающих сигнатурах функций. Повторяя рассуждения о том, что отношение доминирования имен индуцирует отношение доминирования для классов (раздел 1.4), мы придем к следующему определению доминирования для виртуальных функций. Будем говорить, что виртуальная функция *f* класса *D* (которую ниже будем обозначать как *D::f*) *доминирует над* виртуальной функцией класса *A (A::f)* с той же сигнатурой, если класс *D* доминирует над классом *A*. Будем считать, что функции с различными сигнатурами не доминируют друг над другом. Отношение доминирования для виртуальных функций несущественно для семантической проверки программ, но является существенным на этапе выполнения и при генерации таблиц виртуальных функций.

Следует также отметить, что если в самом классе *C* определена функция *f,* то она по определению доминирует над всеми функциями базовых классов с совпадающей сигнатурой. Это свойство называют *замещением* (*overriding*).

2.4 Механизмы времени компиляции

В этом разделе рассматриваются основне механизмы времени компиляции, использованные в компиляторе для поддержки механизма классов. Описывается генерация таблиц, которые используются описанными в следующем разделе механизмами времени выполнения. Также обсуждаются возможные оптимизации этих таблиц.

2.4.1 Обозначения

Введем следующие обозначения

$\|s\|$	- количество элементов списка *s*;
C	- обрабатываемый класс;
S[i]	- *i*-ый элемент списка *s*, будем считать, что элементы нумеруются с единицы;
{X}	- список, состоящий из единственного элемента *X*;
$e \in l$	- полагаем, что элемент *e* принадлежит списку *l*, если $\exists i, 1 \le i \le \|l\|): l[i]=e$;

2.4.2 Построение списка всех виртуальных функций

Строится упорядоченный список *all_vf(C)* всех виртуальных функций-членов всех базовых классов. Строго говоря, этот список не является списком в классическом понимании этой структуры данных. Эта структура скорее является динамическим массивом, так как в дальнейшем будет использоваться двоичный поиск в этом списке, что подразумевает прямой доступ к элементам списка по индексу. Но, тем не менее, в дальнейшем мы будем называть ее списком.

На первом шаге сливаются списки *all_vf(B)*, $C \Rightarrow B$. Так как в процессе построения этого списка в него включаются, как будет показано ниже, все доминирующие элементы списков для всех базовых классов, достаточно на самом деле сливать списки виртуальных функций только для непосредственных базовых классов.

На следующем шаге список сортируется. Основная идея сортировки состоит в том, чтобы сгруппировать виртуальные функции-члены с совпадающими сигнатурами. Поэтому в качестве первого ключа для сортировки используется лексикографический порядок текстового представления сигнатур функций, так называемых «*испорченных имен*» (*mangled names*), которые доступны из семантических слов функций. Функции с различными именами имеют различные сигнатуры, поэтому такую сортировку можно даже убыстрить, воспользовавшись порядком имен, о котором подробно говорится в третьей главе диссертации. Отдельно рассматриваются виртуальные деструкторы; их имена, а следовательно и сигнатуры, определенные по общим правилам, всегда различны. Будем считать, что сигнатуры различных деструкторов равны, и они «меньше» сигнатуры любой другой функции. Если сигнатуры двух функций совпадают, то элементы списков сравниваются по полю *from*, подробно описанному в разделе 2.4.13. В начало списка выносятся элементы, у которых в этом поле находится ссылка на

дескриптор производного класса (определенного в разделе 1.7). Смысл такой дополнительной сортировки будет объяснен ниже.

Функция–член класса *C* может быть виртуальной, даже если это не объявлено явно в тексте программы (а следовательно, изначально не отражено в семантических таблицах). Функция *f∈meth(C)* неявно является виртуальной, если для одного из базовых классов *B* в списке *all_vf(B)* есть функция с той же сигнатурой ([1], раздел 10.3). Для таких функций атрибут виртуальности выставляется в семантических таблицах, как только виртуальность была установлена. Если по этому правилу делается попытка выставить атрибут виртуальности для функции, явно объявленной как статическая, то генерируется сообщение об ошибке.

На следующем шаге в уже построенный список с сохранением порядка добавляются часть функций из списка *meth(C)*. Функция *f* из *meth(C)* добавляется в *all_vf(C)* только в том случае, если в нем уже есть функция с совпадающей сигнатурой (что легко проверяется, так как при добавлении сохранялся порядок и так или иначе должен производиться поиск места для вставки в упорядоченный список), таким образом выявляется неявная виртуальность функций, либо эта функция явно объявлена как виртуальная, что проверяется по ее семантическому слову. Деструктор вставляется в список с учетом приведенных выше правил упорядочивания. Если на этом шаге в список вставляется некоторая функция *f*, то из него удаляются все уже находящиеся там функции с совпадающей сигнатурой. Такое удаление корректно, так как мы хотим иметь в списке только доминирующие функции, а виртуальные функции класса *C* по определению доминируют над всеми виртуальными функциями базовых классов с совпадающими сигнатурами. При добавлении в поле *from* нового элемента списка записывается указатель на дескриптор класса *C*.

Последним шагом является чистка полученного списка и выявление неоднозначностей виртуальных функций. На этом шаге делается проход по построенному списку *all_vf(C)*. По определенному для него отношению порядка функции с одинаковыми сигнатурами в нем расположены последовательно. Смысл чистки состоит в удалении вхождений в список функций, доминируемых какой-либо другой функцией из этого списка, и удалении повторений. По критерию сортировки элементы списка, соответствующие функциям с совпадающими сигнатурами, содержащие в поле *from* производные классы предшествуют элементам, содержащим в поле *from* базовые классы. Таким образом, единственным кандидатом на то, чтобы доминировать над всеми остальными элементами части списка, состоящего из функций с совпадающими сигнатурами,

является ее первый элемент. Проверка производится последовательным проходом по части списка и сравнением ее элементов с первым ее элементом. При этом повторяющиеся вхождения первого элемента удаляются. Если максимально доминирующего элемента не нашлось, то есть после чистки части списка в ней осталось более одного элемента, то соответствующие элементы списка помечаются как *неоднозначные* (*ambiguous*).

Для обозначения неоднозначности виртуальной функции по отношению доминирования для виртуальных функций введем предикат $ambig_C(e)$, который определяется только для элементов списка, но не для самих функций, так как одна и та же функция может быть однозначной в списке *all_vf(C)* одного класса и быть неоднозначной в списке другого класса *all_vf(C').*

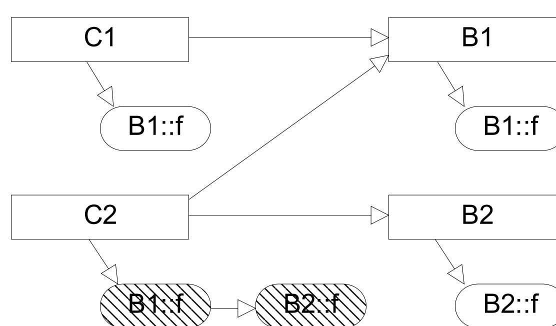


Рис. 2-2. Пример неоднозначности виртуальной функции.

На примере выше показано, что функция *B1::f* может быть неоднозначной для класса *C2*, но однозначной для классов *C1* и *B1*.

Не удаляя из списка функции, для которых установлена неоднозначность, мы избегаем сложностей при построении списков для более производных классов, так как при сливании списков не возникает необходимости рекурсивно обходить все базовые классы, но достаточно сливать списки только непосредственных базовых классов.

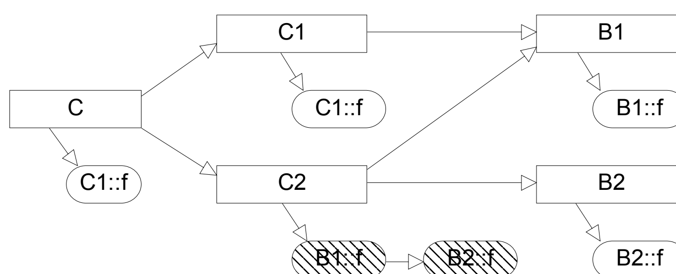


Рис. 2-3. Пример разрешения неоднозначности в производном классе.

В первой главе уже обсуждались неоднозначности, связанные с доминированием имен и неоднозначностью базовых классов. Неоднозначность

виртуальных функций, как видно из приведенных ниже примеров, не сводится к предыдущим двум видам неоднозначностей.

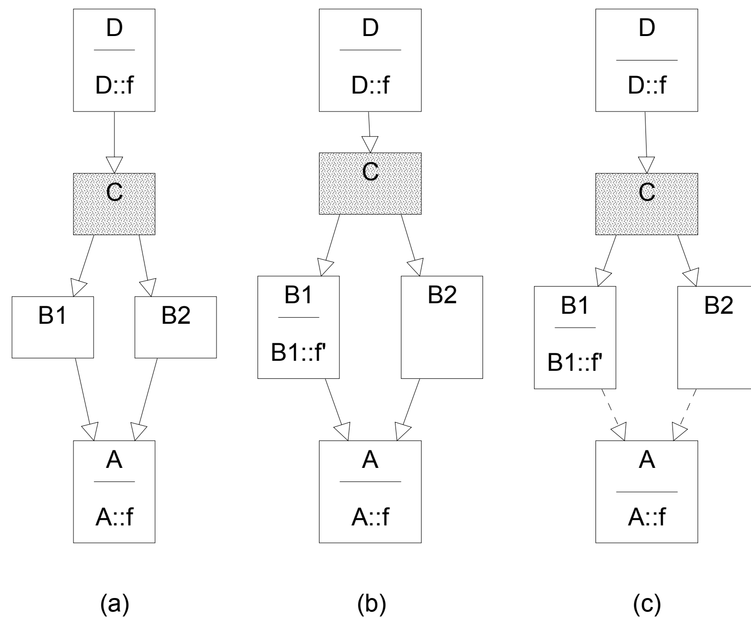


Рис. 2-4. Различные виды неоднозначностей.

На примере (a) показано, что виртуальная функция *A::f* может быть однозначной в классе *C*, при этом класс *A* не является однозначным базовым классом *C*.

На примере (b) показано, что виртуальная функция *B1::f’* может быть однозначной в классе *C*, в то время как имя виртуальной функции неоднозначно в этом классе, если функция *B1::f’* имеет то же имя, что и *A::f*, но другую сигнатуру. Имя *f’* в классе *B1* не доминирует над тем же именем в классе *A*.

На примере (c) показано, что если функции *B1::f’* и *A::f* имеют одинаковые имена, но различные сигнатуры, то имя из класса *B1* доминирует над именем из класса *A*, но при этом обе виртуальные функции являются однозначными в классе *C*.

В некоторых случаях неоднозначности имен или базовых классов для виртуальных функций можно пытаться сводить к неоднозначности виртуальных функций, но последняя фактически проявляются только на этапе выполнения, в то время как первые должны проверяться на этапе трансляции, и они по общим правилам применяются к виртуальным функциям. Но как видно из приведенного выше примера, в некоторых случаях (для объектов максимального производного класса *D* при доступе к нему через указатель на *C*) эти проверки могут оказаться излишне жесткими. Фактически в этих случаях виртуальные функции могли бы разрешать два других вида неоднозначностей на этапе выполнения.

Оценим сложность этой части алгоритма обработки класса.

Пусть $bvf = \sum_{B<C} |all_vf(B)|, \quad svf = |meth(C)|$, тогда слияние списков базовых классов требует *O(bvf)* операций, его сортировка требует *O(bvf* ln *bvf)* сравнений. Поиск места для вставки каждого элемента *meth(C)* требует *O(*ln *(bvf+svf))* сравнений. Получившаяся после вставки элементов *meth*(*C*) длина списка не превосходит *bvf*+|*meth*(*C*)|. Столько же сравнений потребуется выполнить при чистке списка. Если удаление производилось при добавлении элемента, соответствующего элементу *meth*(*C*), то длина списка при его чистке уменьшится на один элемент, что не изменит суммарную сложность этой части алгоритма, которую можно оценить как *O(bvf* ln *bvf* + *svf* ln *(bvf+svf))* или более грубо как *O(nN* ln *nN)*, где *n* – число виртуальных функций в программе, а *N* – число классов.

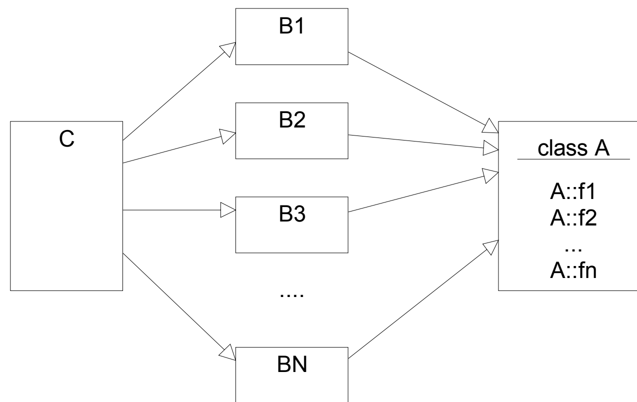


Рис. 2-5. Пример точности оценки сложности построения списка виртуальных функций.

Как видно из приведенного выше примера, даже такая грубая оценка оказывается точна. Для каждого из классов *Bi* его список виртуальных функций содержит *n* элементов (копию списка для класса *A*), а список для класса *C* после слияния списков базовых классов содержит *nN* элементов. Для сортировки последнего списка уже потребуется *O(nN* ln *nN)* сравнений.

2.4.3 Выявление чистых виртуальных функций и абстрактных классов

Виртуальные функции могут быть объявлены как *чистые виртуальные*, информация об этом доступна в семантических таблицах. Класс, в котором объявлена хотя бы одна виртуальная функция, или унаследовавший от базового класса без перекрытия хотя бы одну чистую виртуальную функцию, называется *абстрактным*.

Определить абстрактность класса, которая по определению является атрибутом класса, можно пройдя по списку *all_vf(C)* и проверив является ли хотя бы одна из функций чистой виртуальной. Наличие атрибута чистой виртуальности проверяется по семантическим таблицам. Эта проверка требует *O(|all_vf(C)|)* проверок функций.

2.4.4 Подбор основного базового класса

Этот шаг можно считать опциональным. Подбор основного базового класса является оптимизаций, позволяющей классу *C* и подобъекту одного из его непосредственных базовых классов использовать общую таблицу смещений виртуальных базовых классов.

Подбор основного базового класса не всегда возможен. Если подбор произвести не удается, то будем считать, что у класса *C* нет основного базового класса.

Подбор заключается в выборе среди непосредственных невиртуальных базовых классов класса *C* класса, имеющего наибольшее число виртуальных базовых классов. Основной базовый класс будем обозначать как *B_major(C)*. Если у класса нет непосредственных невиртуальных базовых классов, то *B_major(C)=nil*. Если у класса *C* нет виртуальных базовых классов, то выбор основного базового класса не имеет смысла.

Сложность этой части алгоритма *O(|bases(C)|)*. Точность такой оценки очевидна.

2.4.5 «Раскладка» класса

Раскладка класса состоит в приписывании статических смещений базовым классам и членам класса. Эти смещения, как было указано выше, имеют смысл только для объектов наибольшего производного класса *C*. В классах, производных от класса *C,* виртуальные базовые классы могут иметь другие относительные смещения в подобъекте класса *C*. По этой причине будут последовательно размещаться сначала объекты невиртуальных базовых классов, затем члены класса *C*, и только затем подобъекты виртуальных базовых классов.

В результате работы этой части алгоритма также будет вычислен размер области памяти, необходимый для хранения объекта класса *C*, собственный размер объекта класса *C* и требуемое для объектов выравнивание.

При составлении этой части алгоритма также приходится учитывать, что все действия должны производится над уже существующими структурами данных компилятора, в частности, если бы компилятор в явном виде строил граф *Sub(C),*

смещения было бы удобно приписывать ребрам этого графа. Поэтому мы будем работать только со списком непосредственных невиртуальных базовых классов (частью списка непосредственных базовых классов) и списком всех виртуальных базовых классов. Так как в каждом из списков непосредственных базовых классов и всех виртуальных базовых классов каждый класс встречается не более одного раза, не существует опасности приписать одному и тому же подобъекту различные смещения.

Положим переменные целочисленного типа `current_offset`=*0,* `current_alignment`=*1.*

Определим вспомогательную процедуру `разместить`(*a*), параметром которой может выступать базовый класс или нестатический член класса.

`current_alignment` = НОК(`current_alignment`, *alignment*(*a*))

На этом шаге для параметра *a* учитываются требования по выравниванию всего объекта класса *C*.

Если `current_offset` mod *alignment*(*a*) ≠ 0

Увеличиваем `current_offset` на *alignment*(*a*) – (`current_offset` mod *alignment*(*a*)).

Таким образом, выравнивается подобъект *a* в объекте класса *C*.

Присваиваем *offset(a)*=`current_offset`.

То есть, приписываем *a* смещение в объекте класса *C*.

Увеличиваем `current_offset` на *own_size(a)* если *a* – это базовый класс, или на *sizeof(a),* если это член класса.

То есть в объекте класса *C* отводим память для *a*.

Если определен *B_major*(C)

Выполняем `разместить` (*B_major*(C)).

После этого подобъект класса *B_major*(C) получает нулевое смещение в объекте класса *C*.

В цикле по всем элементам *B* списка непосредственных невиртуальных базовых классов *bases(C)* кроме *B_major*(C) выполняем

`Разместить` (*B*)

В цикле по всем элементам *A* списка *attr(C)* нестатических членов класса *C* выполняем

`Разместить` (*A*).

Если список виртуальных базовых классов класса *C* (*all_vbases(C)*) непуст

Создаем “скрытый” член класса *C*: *vbase_ptr(C)* указательного типа.

Если определен *B_major(C)*

Присваиваем *offset*(*vbase_ptr*(*C*)) = *offset*(*vbase_ptr*(*B_major*(*C*))).

Иначе

Выполняем `разместить`(*vbase_ptr*(*C*)).

Если список виртуальных функций класса *C* (*all_vf(C)*) непуст

Создаем “скрытый” член класса *C* : *vf_ptr(C)* указательного типа.

Выполняем `разместить`(*vf_ptr(C))*

Присваиваем “собственный” размер класса *C* равным `current_offset`: *own_size(C)=current_offset*.

В цикле по всем элементам *B* списка виртуальных базовых классов *all_vbases*(C).

Выполняем `разместить`(B).

Присваиваем *alignment*(C)=`current_alignment`.

Учитываем найденные требования по выравниванию класса *C*.

Если `current_offset` mod `current_alignment` ≠ 0

Увеличиваем `current_offset` на `current_alignment` - `current_offset` mod `current_alignment`.

Полагаем размер класса *C* равным `current_offset`: *sizeof(C)*=`current_offset`.

Можно считать, что эта процедура, вызванная для максимально производного класса, приписывает смещения ребрам графа *Sub(C)*. При этом если *B* – некоторый базовый класс, то изоморфизм подграфа *Sub*(C)* и графа *Sub*(B)* сохраняет смещения.

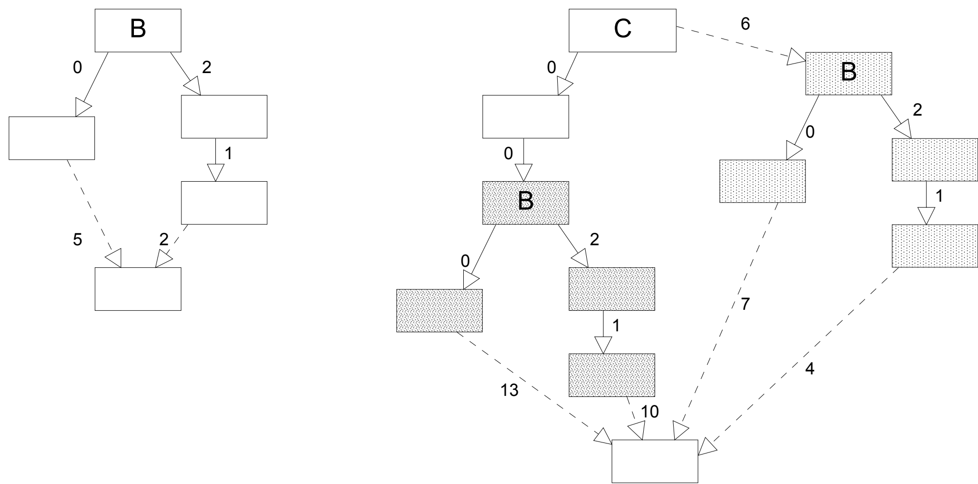


Рис. 2-6. Пример изоморфизма подграфов, сохраняющего смещения ребер.

На приведенном рисунке граф *Sub(C)* содержит два подграфа, изоморфных графу *Sub*(B)*. Ребра, соответствующие виртуальному наследованию, выделены пунктиром.

Процедура приписывания смещений определена таким образом, что для любых ребер *p* и *q* графа *Sub*(S)* из того, что *m(p)=m(q),* следует, что приписанные этим ребрам смещения совпадают. Таким образом, можно перенести смещения для ребер в граф наследования *Inh*, приписав ребрам невиртуального наследования смещения, равные смещениям их прообразов в графе *Sub*(S)*.

Для виртуальных базовых классов это свойство выполнено не всегда, но, тем не менее, в графе *Sub(C)* сумма смещений по путям, ведущим в подобъект базового класса, одинакова для всех путей, включая пути, содержащие ребра соответствующие виртуальному наследованию. Эта сумма, очевидно, равна смещению подобъекта виртуального базового класса, и смещение, приписанное последнему ведущему в вершину виртуального базового класса ребру, подбирается соответствующим образом. При генерации таблицы смещений виртуальных базовых классов такой подбор неявно будет производиться.

Сложность этой части алгоритма: *O(|bases(C)|+|all_vbases(C)|+|attr(C)|)*. Очевидно, что эта оценка точна.

2.4.6 Создание таблицы смещений виртуальных базовых классов

Эта таблица создается для поддержки механизмов времени выполнения. В отличие от всех построенных ранее списков, она явным образом помещается в коде, получаемом в результате работы компилятора. Использование этой таблицы будет описано ниже, при описании механизмов времени выполнения.

Таблица смещений виртуальных базовых классов создается не более одного раза для каждого класса. В генерируемом коде она помечается уникальной меткой, имеющей внутреннее связывание. Элементами таблицы являются целые числа.

В процессе построения таблицы виртуальным базовым классам приписываются индексы в классе *C*. Индексы сохраняются как атрибуты в таблице отношений между классами.

Таблица смещений виртуальных базовых классов и таблицы смещений виртуальных базовых классов для подобъектов базовых классов строятся только для тех классов, определенных в единице компиляции, для которых в этой единице компиляции генерируются код конструкторов. Но приписывание индексов и сортировка списков выполняется для всех классов в единице трансляции.

Положить целочисленную переменную `current_index`=*0*

Если $B_major(C) \neq nil$

> В цикле по всем элементам *B* списка *all_vbases(B_major(C))* в порядке возрастания $index_{B_major(C)}(B)$
>
> > Присвоить $index_c(B)=current_index$.
> >
> > По построению $\forall B \in all_vbases(B_major(C))\ B \in all_vbases(C)$.
> >
> > Увеличить *current_index* на 1.
> >
> > Таким образом, индексы виртуальных базовых классов в классе *C* повторяют индексы виртуальных базовых классов его основного базового класса.
>
> Если $|all_vbases(C)|=|all_vbases(B_major(C))|$ и для *B_major(C)* таблица смещений виртуальных базовых классов уже построена
>
> > Построение таблицы смещений виртуальных базовых классов прерывается, таблица для класса *C* разделяется с таблицей для класса *B_major(C)*.

В цикле по всем элементам *B* списка *all_vbases(C)*

> Если $B_major(C)=nil \vee B \notin all_vbases(B_major(C))$
>
> > Положить $index_c(B)=current_index$
> >
> > Увеличить *current_index* на 1.

Список *all_vbases(C)* сортируется в порядке возрастания индексов.

Генерируется метка начала таблицы.

В цикле по всем элементам *B* списка *all_vbases(C)*

Генерируется элемент таблицы, равный $offset_C(B)$. При этом статическое смещение находится из списка виртуальных базовых классов, так как базовый класс *B* может быть неоднозначным базовым классом.

Таблица построена. Так как *B_major(C)* имеет статическое смещение 0, в объекте класса *C* статические смещения виртуальных базовых классов класса *B_major(C)* в классе *C* совпадают со статическими смещениями тех же виртуальных базовых классов в классе *C*.

$\forall\ B \in all_vbases(B_major(C))\ \ offset_C(B)=offset_{B_major(C)}(B),\ index_C(B)=index_{B_major(C)}(B)$.

Благодаря этому свойству на следующем шаге алгоритма можно не строить таблицу смещений базовых классов для базового класса *B_major(C)*, а использовать таблицу, построенную для самого класса *C*.

Для того, чтобы посчитать сложность этой части алгоритма придется воспользоваться тем, что $|all_vbases(B_major(C))| \le |all_vbases(C)|$, при этом равенство достижимо. Тогда для поиска в списке *all_vbases(C)* элементов, соответствующих элементам аналогичного списка *B_major(C),* потребуется $O(|all_vbases(B_major(C))|\ \ln\ |all_vbases(C)|) \le O(|all_vbases(C)|\ \ln\ |all_vbases\ (C)|)$ сравнений. Сравнимое число операций потребуется для сортировки списка для класса *C*. Наконец, собственно генерация таблицы потребует $O(|all_vbases(C)|)$ шагов цикла. Таким образом, окончательная сложность этой части алгоритма есть $O(|all_vbases(C)|\ \ln\ |all_vbases(C)|)$. Точность оценки следует из точности оценки сложности сортировки списка.

2.4.7 Создание таблицы смещений виртуальных базовых классов для подобъектов базовых классов

Как говорилось ранее, смещения виртуальных базовых классов для базовых классов максимального класса *C* могут не совпадать со смещениями, построенными для них как максимальных базовых классов. Поэтому для каждого подобъекта базового класса, имеющего виртуальные базовые классы, необходимо построить таблицу смещений, отражающую статические смещения в классе *C*. Сложность такого построения состоит в том, что базовые классы класса *C* могут быть неоднозначными, но к каждому из них может быть получен доступ через указатель путем последовательных преобразований, каждое из которых является семантически корректным преобразованием от указателя на производный класс к указателю на базовый. Поэтому следующие таблицы строятся не для базовых

классов, а для подобъектов класса *C*. Различные подобъекты одного и того же базового класса получают различные таблицы.

С точки зрения реализации эти таблицы строятся как продолжение таблицы, построенной в предыдущем подпункте. Доступ к ним производится как к частям большой таблицы, относительно ее метки.

Так как таблицы строятся для подобъектов, а граф *Sub(C)* компилятор в явном виде не строит, наиболее естественным путем их построения является рекурсивный обход, эмулирующий обход дерева *Sub(C)*.

Функция `DumpBaseVbaseOffsets (C,B,o)`

Параметры:

Ссылка на дескриптор максимального класса *C*,

Ссылка на обрабатываемый базовый класс *B*,

Дополнительное смещение *o* – целое.

В цикле по всем элементам *D* списка *all_vbases(B)* в порядке возрастания $index_B(D)$, то есть в том порядке, в котором они находятся после выполнения алгоритма обработки для класса *B*

Поместить в код целое число $offset_C(D)$-*o*.

В цикле по всем элементам *D* списка непосредственных невиртуальных базовых классов *bases(B)*

Выполнить `DumpBaseVbaseOffsets`(*C,D,o+*$offset_B(D)$).

Конец функции `DumpBaseVbaseOffsets`.

Запуск этой функции при обработке класса *C* производится следующим образом:

В цикле по всем элементам *B* списка непосредственных невиртуальных базовых классов *bases(C)*

Если $B \neq B_major(C)$

Выполнить `DumpBaseVbaseOffsets`(*C,B,*$offset_C(B)$)

После этого шага построены все таблицы для подобъектов подграфа *Sub*(C)*, если учесть, что некоторые таблицы разделяются производными и основными базовыми классами.

В цикле по всем элементам *B* списка виртуальных базовых классов *all_vbases(C)*.

Выполнить `DumpBaseVbaseOffsets`(*C,B,*$offset_C(B)$).

На этом шаге таблицы строятся для остальной части вершин графа *Sub(C)*.

В терминах графа подобъектов в таблицу для подобъекта базового класса попадают смещения, приписанные путям, ведущим из подобъекта базового класса к подобъекту виртуального базового класса в графе *Sub(C)*. Свойство равенства сумм смещений по различным путям выполняется и в этом случае. В качестве параметра *o* в процедуру передается сумма смещений по уже пройденному участку пути к подобъекту базового класса.

Сложность функции `DumpBaseVbaseOffsets` можно оценить как *O(|all_vbases(B)| + |bases(C)|)*. Эта функция вызывается для почти каждого узла (всех кроме пар «производный - основной базовый») графа *Sub(C)*. Из этого можно сделать предположение о том, что общая сложность этой части алгоритма ограничена только экспонентой от числа базовых классов класса *C*. Следующий пример демонстрирует возможность такого случая. Каждый класс *Di* может иметь только один основной базовый класс, и процедура будет вызвана рекурсивно для двух других классов.

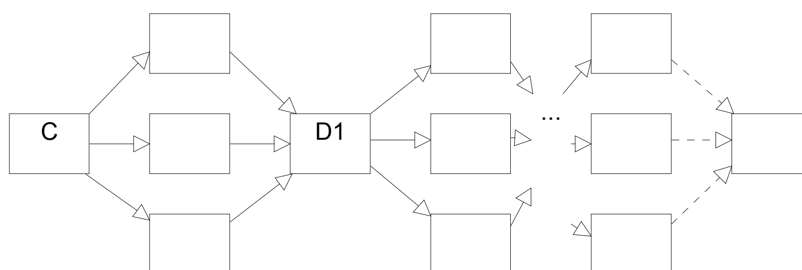


Рис. 2-7. Пример неэффективности оптимизаций таблиц смещения виртуальных базовых классов.

Таким образом, сложность этой части алгоритма есть $O(2^N)$. Оценка размера генерируемых для подобъектов класса *C* таблиц будет такой же. Как будет показано ниже, такой же будет оценка времени работы во время выполнения конструктора для класса *C,* так как конструктор устанавливает указатели на сгенерированные таблицы для подобъектов. К счастью, на практике подобные иерархии встречаются редко.

Ниже будет рассматриваться генерация информации для определения типа объекта времени выполнения, пользуясь которой, можно построить другой метод для поддержки нефиксированных смещений виртуальных базовых классов, не требующий генерации таблиц потенциально экспоненциального размера. Но такой способ реализации был отвергнут, так как описанный способ лучше подходит для реальных программ на Си++.

2.4.8 Создание таблицы виртуальных функций

Способ генерации таблицы виртуальных функций во многом повторяет способ генерации таблицы смещений виртуальных базовых классов. В качестве смещений виртуальных базовых классов выступают смещения подобъектов, в которых определена доминирующая виртуальная функция. Так же, как и виртуальным базовым классам, виртуальным функциям будет приписываться индекс в таблице.

Таблица состоит из пар, первый член которых – адрес функции, второй – целое число, которое затем будет использоваться как смещение.

Можно также определить механизм, аналогичный выбору основного базового класса. В качестве такого базового класса может выступать базовый класс, имеющий нулевое смещение в классе *C*. Но в этой части он рассматриваться не будет, так как его описание сделает изложенные ниже алгоритмы более громоздким.

Аналогично таблицам смещений виртуальных базовых классов таблицы виртуальных функций строятся только для тех классов, для которых в единице трансляции генерируется код хотябы для одного из конструкторов.

Присвоить целочисленной переменной `current_index` значение 0.

В цикле по всем элементам с уникальной сигнатурой *F* из списка *all_vf(C)*

> Присвоить функции *F* и всем следующим за ней функциям с той же сигнатурой индекс `current_index`.
>
> Пропустить все функции с той же сигнатурой, что и *F*. В списке они находятся непосредственно за *F*.
>
> Увеличить `current_index` на 1.

В цикле по всем элементам с уникальной сигнатурой *F* из списка *all_vf(C)*

> Если *ambig(F)* или *from(F)* является неоднозначным базовым классом для класса *C*.
>
> > Поместить в таблицу адрес вспомогательной функции *ambig_vf* и целое число 0.
>
> Иначе
>
> > Поместить в код адрес функции *F* и целое число *offset(from(F))*.

Функция *ambig_vf* является вспомогательной функцией библиотеки времени выполнения. Будучи вызвана, она аварийно завершает выполнение программы. Это происходит в тех случаях, когда во время выполнения происходит попытка вызвать неоднозначную виртуальную функцию.

Сложность этой части алгоритма, очевидно, равна *O(|all_vf(C)|)*.

2.4.9 Создание таблиц виртуальных функций для подобъектов базовых классов

Таблицы виртуальных функций для подобъектов базовых классов генерируются аналогично таблицам смещений виртуальных базовых классов для базовых классов; в частности, они генерируются для отдельных подобъектов базовых классов. Для некоторых базовых классов может потребоваться генерация нескольких таблиц, часть из которых используется только на этапе создания и разрушения объекта. Далее будет приведена полная процедура генерации таблиц виртуальных функций для базовых классов, и будут указаны условия, при которых часть этих таблиц строить не требуется.

Так как отношение доминирования определяется для классов, являющихся базовыми для класса *C,* и оно определяет выбор виртуальных функций, вызываемых во время выполнения, может возникнуть ситуация, когда функция с сигнатурой *f* оказывается неоднозначной в классе *C*, но однозначной в его базовых классах.

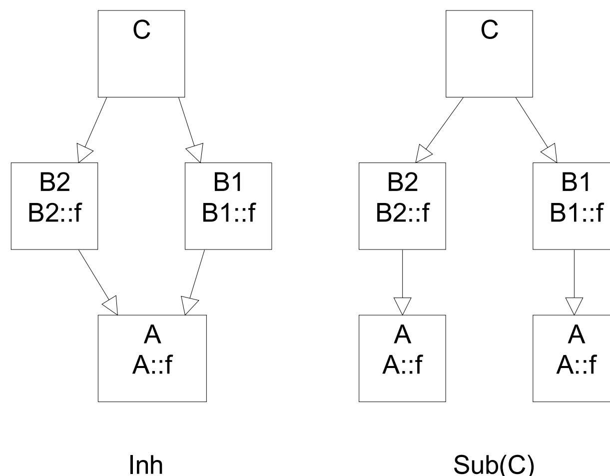


Рис. 2-8. Пример частичного доминирования виртуальных функций.

В приведенном выше примере ни одна из функций *B1::f* и *B2::f* не доминирует над другой в классе *C*, но они доминируют над функцией *A::f*, при этом в различных подобъектах класса *A* доминируют различные функции. Такие случаи

абсолютно корректны с точки зрения семантики языка, и их необходимо учитывать при генерации таблиц виртуальных функций для базовых классов.

Функция `DumpBaseVfTableForBaseInPB(C,PB).`

Параметры:

Ссылка на дескриптор максимального класса *C*.

Непустой список пройденных базовых классов *PB*, состоящий из пар (базовый класс, смещение).

В списке *PB* выделим последний элемент *(B,o)*.

В цикле по всем элементам с уникальной сигнатурой *F* из списка *all_vf(B)*

Присвоить *F'=nil*

В цикле по всем элементам $(E, offset_{PB}(E))$ списка *PB*

В списке *all_vf(E)* найти первую функцию *G* с той же сигнатурой, что и *F*.

Если *G* является однозначной виртуальной функцией, и *from(G)* является однозначным базовым классом для класса *E*

присвоить *F'=G*

прервать выполнение ближайшего объемлющего цикла.

Если *F'=nil*

Поместить в таблицу адрес вспомогательной функции *ambig_vf* и целое число 0.

иначе

Поместить в таблицу адрес функции *F'* и смещение, равное

$$\begin{cases} offset_C(from(F')) - o, & from(F') \in all_vbases(E) \\ offset_E(from(F')) - o + offset_{PB}(E), & from(F') \notin all_vbases(E) \end{cases}$$

Смещение подобъекта вычисляется на основе смещений в классе *C* и равно смещению подобъекта класса *from(F')* за вычетом смещения, приписанного подобъекту класса *B* в списке *PB*. В терминах графа подобъектов оно равно разнице суммы смещений, приписанных ребрам графа *Sub(C)* по пути в вершину текущего подобъекта класса *B*, и суммы смещений, приписанных пути в вершину объекта класса *from(F')*. Если в этих путях нет ни одного ребра виртуального

наследования, то разницу смещений можно вычислять в графе *Sub*(E)* как $offset_E(from(F'))$-$offset_E(B)$, то есть она не зависит от класса *C*. Смещение подобъекта класса *from(F')*, если он является подобъектом невиртуального базового класса, совпадает с суммой его статического смещением в классе *E* и смещения подобъекта класса *E* в объекте класса *C*, которое известно в списке *PB*.

Список *PB* используется в этой процедуре для реализации частичного доминирования виртуальных функций для базовых классов.

Внешний цикл этой процедуры выполняется |*all_vf(B)*| раз, что не превосходит |*all_vf(C)*|. Внутренний цикл выполняется не более |*all_bases(C)*| раз. Поиск в упорядоченном списке функции с определенной сигнатурой требует $O(\ln |all_vf(E)|) \leq O(\ln |all_vf(C)|)$. Итого сложность этой функции можно оценить как $O(|all_vf(C)|\ |all_bases(C)|\ \ln |all_vf(C)|)$, или, воспользовавшись введенными выше обозначениями, $O(nN \ln n)$.

Процедура `DumpBaseVfTableForBaseInPB` вызывается из процедуры `DumpBaseVfTableForBases`, описанной ниже, которая фактически осуществляет рекурсивный обход части дерева *Sub(C)*.

Функция `DumpBaseVfTableForBases(C,PB)`.

Параметры:

Ссылка на дескриптор максимального класса *C*.

Список пройденных базовых классов *PB*, состоящий из пар (базовый класс, смещение).

В списке *PB* выделим последний элемент *(B,o)*.

Выполнить `DumpBaseVfTableForBaseInPB` с параметрами `(C,PB)`.

В цикле по всем непосредственным невиртуальным базовым классам *E* класса *B*

Рекурсивно выполнить `DumpBaseVfTableForBases` с параметрами `(C,PB+(B,o+`$offset_B(E)$`))`.

Содержательный смысл этой функции – генерация таблицы виртуальных функций для базового класса *B* в объекте класса *C,* используемой при создании или удалении подобъекта класса *D,* который является первым элементом списка *PB*. При этом должны использоваться функции, доминирующие в списке виртуальных функций для класса *D*, но смещения виртуальных базовых классов, в которых виртуальные функции могут быть определены, должны быть корректными для подобъектов класса *C*.

Эта функция, очевидно, будучи вызвана со вторым параметром *PB*, выполняет полный обход подграфа *Sub*(C),* имеющего корнем вершину класса *B*. Общее число рекурсивных вызовов не превосходит числа вершин графа *Sub*(C)*, которое, как было доказано в первой главе, может расти экспоненциально по отношению к *N*, а именно как 2^{N-1}. Пример точности этой оценки для числа вершин графа *Sub*(C)* также является примером графа, рекурсивный обход которого требует посещения 2^{N-1} узлов (рис. 2-7). Следовательно, сложность этой процедуры можно оценить как $O(2^N)$ вызовов процедуры `DumpBaseVfTableForBaseInPB`. Максимальная глубина рекурсии этой функции ограничена *N*.

Следующая процедура также рекурсивно обходит дерево подобъектов, строя таблицы виртуальных функций для различных вариантов начала списка *PB*.

Функция `DumpBaseVfTable(C,B,o)`.

Параметры:

Ссылка на дескриптор максимального класса *C*.

Ссылка на базовый класс *B*, для которого генерируется таблица

Смещение базового класса *o*.

Выполнить `DumpBaseVfTableForBases` с параметрами `(C,{(B,o)})`.

В цикле по всем непосредственным невиртуальным базовым классам *E* класса *B.*

Выполнить `DumpBaseVfTableForBases` с параметрами `(C,{(B,o),(E,o+`$offset_B(E)$`)})`.

Рекурсивно выполнить `DumpBaseVfTable` с параметрами `(C,B,o+`$offset_B(E)$`)`.

В цикле по всем классам *E* списка *all_vbases(B)*

Выполнить `DumpBaseVfTableForBases` с параметрами `(C,{(B,o),(E,`$offset_C(E)$`)})`.

Аналогично предыдущей процедуре, сложность этой процедуры оценивается как $O(2^N)$ вызовов процедуры `DumpBaseVfTableForBases`, и максимальная глубина рекурсии ограничена *N*..

При обработке класса *C* последняя процедура вызывается последовательно для самого класса *C* с параметрами *(C,C,*`0`*)* и для всех его

виртуальных базовых классов *all_vbass(C)* с параметрами $(C,B,offset_C(B))$, то есть максимум *O(N)* раз.

Теперь можно подсчитать сложность всей процедуры генерации таблиц виртуальных функций для подобъектов базовых классов. Это будет $O(N)O(2^N)O(2^N)O(nN \ln n)=O(N^2 4^N n \ln n)$. Такова же оценка для размера построенных процедурой таблиц. Пример (рис. 2-7) экспоненциального роста графа подобъектов *Sub(C)* также является примером, иллюстрирующим точность этой оценки.

2.4.10 Пример генерации таблиц

Описанный выше алгоритм генерации таблиц довольно сложен, поэтому уместно будет привести пример генерируемых им таблиц. Рассмотрим следующий граф наследования. Он также интересен тем, что «наивная» реализация таблиц виртуальных функций (как, например, описанная в [22]) генерирует для данного примера некорректные таблицы.

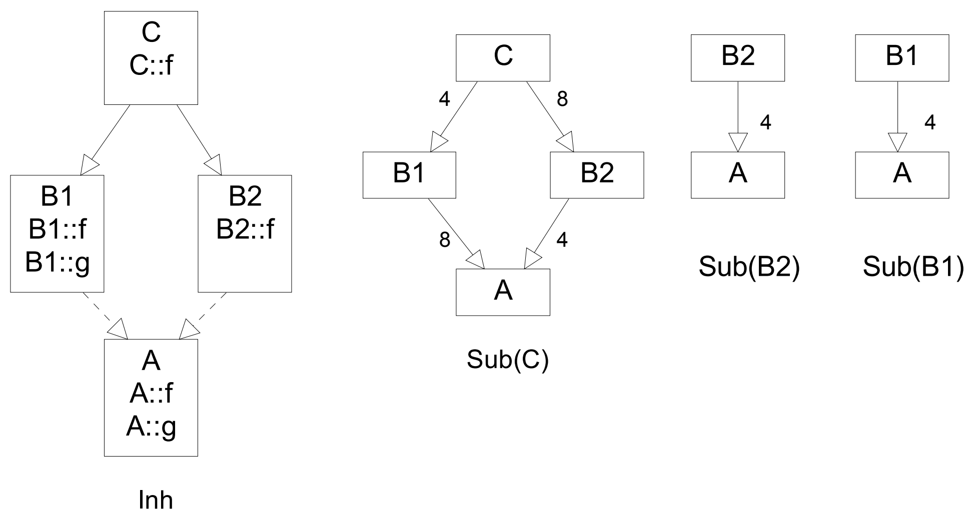


Рис. 2-9. Пример построения таблиц.

Построим все таблицы для этого примера:

Таблица смещения виртуальных базовых классов для классов *B1*, *B2*, *C* состоят из единственного элемента каждая, что соответствует наличию единственного виртуального базового класса *A*. Таблицы для этих классов выглядят, соответственно, как `(4)`, `(4)`, `(12)`.

Таблицы смещений виртуальных базовых классов для подобъектов базовых классов строятся только для класса *C*, так как только он имеет базовые классы, в свою очередь, имеющие виртуальные базовые классы. Для пары *(C,B1)*

таблица состоит из единственного элемента (8), для пары *(C,B2)* - из единственного элемента (4).

Таблица виртуальных функций для класса *A* состоит из двух элементов – пар {(A::f,0),(A::g,0)}.

Таблицы виртуальных функций для классов *B1* и *B2* состоят двух пар каждая, {(B1::f,0), (B1::g,0)} и {(B2::f,0), (A::g,4)}, соответственно. Для подобъектов базовых классов *A* строятся таблицы также состоящие из двух элементов каждая {(B1::f,-4), (B1::g,-4)} и {(B2,-4), (A::g,0)}.

Для базовых классов класса *C* строится больше таблиц; для наглядности приведены параметры функций, построивших эти таблицы.

Параметры функции DumpBaseVfTable	Параметры функции DumpBaseVfTableForBases	Параметры функции DumpBaseVfTableForBaseInPB	Сгенерированная таблица
	(C,{(C,0)})	(C,{(C,0)})	(C::f,0) (B1::g,4)
		(C,{(C,0),(B1,4)})	(C::f,-4) (B1::g,0)
		(C,{(C,0),(B2,8)})	(C::f,-8) (B1::g,-4)
		(C,{(C,0), (A,12)})	(C::f,-12) (B1::g,-8)
	(C,{(B1,4)})	(C,{(B1,4)})	(B1::f,0) (B1::g,0)
		(C,{(B1,4),(A,12)})	(B1::f,-8) (B1::g,-8)
	(C,{(B2,8)})	(C,{(B2,8)})	(B2::f,0) (A::g,4)
		(C,{(B2,8), (A,12)})	(B2::f,-4) (A::g,0)
(C,A,12)	(C,{(A,12)})	(C,{(A,12)})	(A::f,0) (A::g,0)

Табл. 2. Пример построения таблиц виртуальных функций для базовых классов

Некоторые реализации Си++ ошибочно строят таблицы виртуальных функций. Приведенный выше пример интересен тем, что на нем ошибочную генерацию легко заметить. Типичной ошибкой является генерация таблицы функцией DumpBaseVfTableForBaseInPB (C,{(B1,8)},{A,12}), совпадающей с таблицей, построенной DumpBaseVfTableForBaseInPB (B1,{(B1,0)}). Также встречается использование второй таблицы в тех случаях когда требуется первая.

2.4.11 Создание объектов класса type_info

Класс type_info специфицирован в стандарте Си++ ([1], раздел 18.5.1) как один из библиотечных классов. Реализация этого класса может расширять

данную спецификацию, добавляя в класс новые скрытые члены. Описываемая в данной работе реализация расширяет описанный в стандарте класс одним дополнительным членом – указателем на первый элемент массива описателей базовых классов. Этот член имеет ненулевое значение только для объектов класса `type_info`, сгенерированных для классовых типов. Описатели специально подобраны таким образом, что для объектов полиморфных классов на их основе становится возможным реализация операции `dynamic_cast`, а также динамическая идентификация возможности приведения типа при обработке исключительных ситуаций. При генерации таблиц виртуальных функций непосредственно перед генерацией самой таблицы генерируется указатель на объект `type_info` класса *D*, являющегося первым элементом списка *PB,* и целое число, равное разнице смещения, приписанного классу *D* в списке *PB,* и текущего смещения (смещения последнего элемента в списке *PB*).

Массив описателей базовых классов состоит из однотипных элементов. Отдельный элемент генерируется для каждого объекта в графе *Sub(C)*. Каждый элемент является структурой, содержащей следующие поля:

- ссылка на объект типа `type_info` для базового класса;
- смещение, приписанное подобъекту в графе *Sub(C);*
- флаг, говорящий о том, что данный подобъект является подобъектом виртуального базового класса;
- флаг, показывающий доступность подобъекта базового класса в объекте класса *C*.

В этой главе уже приводились примеры процедур полного обхода графа *Sub(C)*. Процедура генерации описателей подобъектов базовых классов реализована аналогично предыдущим и имеет сложность $O(2^N)$. При этом максимальная глубина рекурсии не превышает *N*.

2.4.12 Оптимизация таблиц виртуальных функций

Как видно из описанного выше способа генерации таблиц виртуальных функций для подобъектов базовых классов, эти таблицы для сложных иерархий классов могут получаться чрезвычайно большими. Поэтому отдельный интерес представляет изучение случаев, когда для совпадающих пар базовых классов (*D*,*B*) генерируются одинаковые таблицы виртуальных функций. В этих случаях таблицы виртуальных функций для различных базовых классов могут разделяться между различными классами *C* или различными подобъектами базового класса в классе *C*. Наиболее общий подход к такого рода оптимизации можно реализовать

следующим образом: при генерации таблица не сразу генерируется в результирующем промежуточном коде программы, а формируется как внутренняя структура компилятора. При следующей генерации таблиц для той же пары классов таблица также генерируется как внутренняя структура, и если она в точности совпадает с первой, то вторая сгенерированная таблица уничтожается и вместо нее используется первая (таблица разделяется между двумя различными подобъектами). Также часто встречаются случаи, когда таблица для пар классов (*D*,*B_major(B)*) совпадает с частью (началом) таблицы для пары (*D*,*B*). Это свойство также может быть использовано для оптимизации таблиц. По причинам, изложенным выше, а именно из-за способа доступа к объектам типа `type_info`, в данной реализации мы не пытаемся разделять таблицы для пар (*D1*,*B*) и (*D2*,*B*), где *D1* и *D2* – различные классы. Но при модификации подхода к доступу к объектам класса `type_info` такую реализацию можно осуществить.

Функция `DumpBaseFVTableInPB` может генерировать для базового класса *B* (последнего класса в списке *PB*) идентичные таблицы виртуальных функций для всех максимальных производных классов *C*. Достаточным условием этого является то, что при поиске доминирующих функций в списке *PB* ни для одной из найденных функций *F'* класс *from(F')* не является виртуальным базовым классом класса *B*. В этом случае фактически доминирование и все смещения определяется в подграфе, изоморфном графу *Sub*(E)*, так как изоморфизм для таких подграфов сохраняет смещения, приписанные ребрам. Достаточным условием для такого свойства является отсутствие у класса *D* (первого элемента списка *PB*) виртуальных базовых классов. Параметр *C* не участвует в генерации таблиц в функции `DumpBaseVfTableForBases`, если класс *C* не имеет виртуальных базовых классов.

Функция `DumpBaseVfTableForBaseInPB`, будучи вызвана с параметрами (C,{(C,0)}), генерирует таблицу, идентичную таблице, сгенерированной для самого класса *C* (раздел 2.4.8).

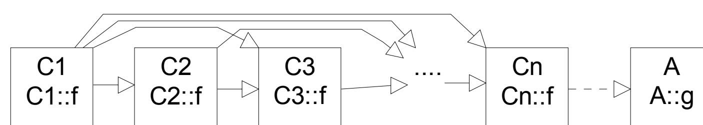


Рис. 2-10. Пример неэффективности оптимизаций таблиц виртуальных функций.

Приведенный выше пример иллюстрирует тот факт, что описанные выше оптимизации таблиц не позволяют избежать экспоненциального роста их числа.

Компилятор фирмы Microsoft генерирует таблицы виртуальных функций несколько иначе: в таблицах присутствует только адрес функции, но отсутствует смещение. В тех случаях, когда в терминах таблиц описываемой реализации смещение ненулевое, этот компилятор генерирует фиктивную функцию, выполняющую смещение скрытого параметра (фактически просто прибавляя к нему константу) и безусловный переход на начало настоящей доминирующей виртуальной функции. Такой способ генерации таблиц не только сокращает размер таблиц, но и увеличивает скорость выполнения программы, так как во время выполнения не производится дополнительное косвенное обращение к таблице, что существенно для современных вычислительных машин, у которых задержки, связанные с доступом к памяти, существенно влияют на производительность.

Реализация GNU CC ([54]) генерирует таблицы в виде, сходном с описываемой реализацией. Но в них также иногда используются скрытые сгенерированные компилятором функции. Эти функции генерируются в случае, когда доминирующая функция выбрана из виртуального базового класса, в этих случаях скрытая функция выполняет выравнивание скрытого параметра `this` с использованием приемов, аналогичных механизму `dynamic_cast`, то есть фактически вычисляя смещение на этапе выполнения, а не выбирая его из таблицы. Генерация таких скрытых функций позволяет генерировать меньшее число таблиц для подобъектов базовых классов, но их число по-прежнему растет экспоненциально, как $O(2^N)$. Вычисление смещения на этапе выполнения также требует больших затрат во время выполнения программы.

2.4.13 Структуры данных, используемые при обработке класса и генерации таблиц

На этом разделе описание механизмов времени компиляции заканчивается. При описании этих механизмов мы неявно и довольно свободно расширяли описанные в первой главе структуры данных, описывающие классы и отношения между ними. Сейчас есть возможность явно описать структуры данных, использованные при описании этих механизмов. Часть из них является простым расширением структур данных, уже описанных в первой главе: дескрипторов классов, дескрипторов базовых классов и дескрипторов отношений между классами.

Кроме перечисленных в первой главе членов, дескриптор класса содержит следующие дополнительные поля:

- список виртуальных функций *all_vf*;
- размер *sizeof*;
- собственный размер *own_size*;
- выравнивание *alignment*;
- ссылки на семантические слова скрытых членов класса *vf_ptr*, *vbase_ptr*;
- ссылки на имена сгенерированных таблиц виртуальных функций, смещений виртуальных базовых классов, объекта типа `type_info`.

Дескриптор базового класса содержит следующие дополнительные поля:

- ссылка на дескриптор отношения между классами.

Дескриптор отношений между классами содержит следующие дополнительные поля:

- смещение *offset*, имеющее смысл только для однозначных базовых классов;
- индекс *index,* имеющий смысл только для элементов списка виртуальных базовых классов;
- список таблиц перенумерации виртуальных базовых классов и виртуальных функций.

Элемент списка виртуальных функций содержит следующие поля:

- ссылка на семантическое слово функции *F*;
- ссылка на дескриптор класса, в котором функция определена *from*;
- индекс *index*;
- флаг неоднозначности виртуальной функции.

2.5 Механизмы времени выполнения

В данном разделе обсуждается реализация механизмов времени выполнения и генерация кода, релизующего механизмы классов и виртуальных функций в компиляторе. Эта реализация основана на таблицах, генерация которых была описана в предыдущем разделе. Также обсуждаются возможные оптимизации, позволяющие упростить генерируемый код и сделать его более эффективным по сравнению с кодом, генерируемым по общим правилам.

2.5.1 Создание и уничтожение объектов классов

В этом разделе обсуждается реализация генерации кода для конструкторов и деструкторов.

Объект класса *C* при его создании в процессе выполнения программы занимает область памяти размером *sizeof(C),* выровненную по границе *align(C)* байт. При его создании происходит неявная (не определяемая пользователем) инициализация скрытых членов *vbase_ptr(C)* и аналогичных членов для базовых классов. Инициализация производится адресами первых элементов таблиц смещений виртуальных базовых классов. Такая инициализация не является частью кода, сгенерированного для конструктора класса *C*, так как класс *C* сам может выступать в качестве базового класса и в этом случае его скрытый член должен инициализироваться адресом другой таблицы, а его конструктор вызывается для инициализации подобъекта и не должен переустанавливать указатель на таблицу. Код для инициализации этих скрытых членов генерируется процедурой, аналогичной описанным выше, выполняющим рекурсивный обход вершин графа *Sub(C)* с вычислением их смещения.

Более сложный код генерируется для инициализации скрытых указателей на таблицы виртуальных функций *vf_ptr*. Такой код генерируется не только при создании объекта класса, но и как часть деструктора. При этом случай деструктора является более сложным. Деструктор класса в качестве дополнительного скрытого параметра получает флаг, сигнализирующий о том, что этот деструктор вызван для максимального динамического типа объекта и должен вызвать деструкторы для своих подобъектов (при этом для деструкторов подобъектов этот флаг не взводится). Если флаг взведен, деструктор также берет на себя работу по инициализации скрытых членов *vf_ptr* перед вызовом деструкторов для подобъектов базовых классов. Деструкторы для членов класса вызываются всегда, независимо от значения флага, и при их вызове передаваемый им флаг взведен.

Инициализация объекта заключается в вызове конструкторов для всех подобъектов его базовых классов и членов. Порядок вызова конструкторов строго оговорен в стандарте; для нас лишь важно, что по правилам языка для любого поддерева *Sub*(D)* конструктор вызывается для вершины только после того, как конструкторы были вызваны для всех вершин достижимых из нее и конструкторы виртуальных базовых классов вызываются до конструкторов подобъектов непосредственных невиртуальных базовых классов. Порядок вызова деструкторов противоположен порядку вызова конструкторов.

Аналогично всем описанным выше функциям, выполняющим обход графа *Sub(C),* реализована функция обхода вершин графа в порядке вызова для них конструкторов. Эта функция генерирует код для последовательного вызова конструкторов подобъектов; каждый вызов конструктора для вершины предваряется кодом, устанавливающим указатели на таблицы виртуальных функций для всех вершин, достижимых из нее. Порядок генерации присваиваний указателей повторяет порядок генерации таблиц виртуальных функций для базовых классов (двойной рекурсивный обход дерева *Sub(C)*) и имеет сложность $O(4^N)$. Как говорилось выше при обсуждении оптимизаций таблиц виртуальных функций, такую установку можно опустить в тех случаях, когда для конструктора установлено, что из него опосредованно не вызываются виртуальные функции. При этом случай непосредственного вызова виртуальной функции из конструктора или деструктора можно рассматривать как квалифицированный вызов функции, не нуждающийся в поиске по таблице виртуальных функций. Также, как было показано в разделе 2.4.12, в некоторых случаях можно убедиться в том, что таблицы виртуальных функций, используемые для самого класса, можно использовать как таблицы виртуальных функций для него как для подобъекта другого класса во время его создания и разрушения. В таких случаях код инициализирующий указатели на таблицы виртуальных функций для этого класса и его подобъектов вносится непосредственно в код конструктора, что позволяет избегать его многократного повторения для производных классов. В таком случае установка таблицы виртуальных функций для виртуальных базовых классов должна производиться с вычислением динамического смещения на основе таблицы смещений виртуальных базовых классов по правилам, описанным в следующем разделе.

2.5.2 Преобразования указателей на классы

Указатель на производный класс по правилам языка может быть преобразован к указателю на однозначный базовый класс. С точки зрения генерации кода эта операция сводится к сдвигу указателя на некоторое смещение. В терминах графа подобъектов *Sub(S)* операция преобразования является поиском суммы смещений, приписанных ребрам графа по пути *p* от некоторой вершины класса *C* к единственной достижимой из нее вершине класса *B*. Класс *S* является максимальным классом объекта, на подобъекты которого указывает преобразуемый указатель. Генерируемый для преобразования код должен корректно выполнять смещение независимо от класса *S*, который может и не

совпадать с *C*, поэтому для преобразования далеко не всегда можно пользоваться статическими смещениями (смещением подобъекта класса B в объекте класса *C*).

Рассмотрим фиксированную часть пути *q=fix(m(p))*. Эта часть пути не содержит ребер виртуального наследования. Ребрам, являющимся прообразом пути *q* приписаны смещения, совпадающие со смещениями, приписанным ребрам пути *q,* и их сумма равна сумме смещений, приписанных ребрам пути *q* графа *Inh*, следовательно не зависит от класса *S*.

Если *m(p)=q*, то независимо от класса *S* преобразование производится смещением на фиксированное число байт. Иначе путь *q* начинается в вершине *R*, являющейся вершиной виртуального базового класса для класса *C*. В списке *all_vbases(C)* ему приписан некоторый индекс *i*. Смещение в этом случае состоит из двух частей – смещения, равного сумме смещений по пути *q* и смещения *R*, которое в зависимости от класса *S* может принимать различные значения, но для подобъекта класса *C* оно может быть извлечено из таблицы смещений виртуальных базовых классов по индексу *i*. Доступ к самой таблице осуществляется через указатель на ее первый элемент, находящийся в члене *vb_ptr(C)*. Доступ к этому члену производится через указатель на объект класса *C* (преобразуемый указатель) по фиксированному смещению.

Таким образом, код преобразования указателей на классы в самом сложном случае состоит из прибавления константы, равной смещению, приписанному пути *p*, операции сложения для получения адреса указателя *vb_ptr*, разыменования указателя для доступа к таблице смещений, прибавления индекса базового класса и разыменования для доступа к динамическому смещению виртуального базового класса. При этом преобразуемый указатель может использоваться дважды: один раз для сложения с константой, второй раз для доступа к указателю на таблицу.

Дополнительное требование языка для преобразования указателей – нулевой указатель всегда должен преобразовываться в нулевой указатель. Это правило реализуется путем генерации условного выражения, сравнивающего указатель с 0.

Ссылки на базовые классы преобразуются аналогичным образом.

Частные случаи, в которых генерируемый код преобразования можно упростить перечислены ниже:

- Проверка на нулевой указатель не делается при преобразовании ссылок на классы.
- Проверка на нулевое значение не генерируется, если код преобразования сводится к прибавлению константного нулевого смещения.

- Если компилятору известен динамический тип объекта, на который указывает преобразуемый указатель, то преобразование сводится к прибавлению константы, равной разнице смещения подобъектов.
- Доступ к члену класса или вызов функции-члена класса также не требует проверки на нулевое значение.
- При преобразовании указателя `this` не генерируется проверка на нулевое значение.

Очевидно, указанные частные случаи допускают генерацию упрощенного кода; выявление этих достаточно распространенных в реальных программах ситуаций желательно проводить по результатам анализа потока данных. Результаты анализа потока данных позволяют выявлять такие частные случаи, даже если они присутствуют в компилируемой программе неявно.

2.5.3 Статические члены класса

Статические члены класса реализуются как обычные глобальные объекты, неявно имеющие внешнее связывание. Синтаксически доступ к ним может производиться как непосредственно, с использованием квалифицированного имени члена, так и с помощью операций доступа к члену класса. В последнем случае код, генерируемый для доступа, состоит из вычисления выражения, стоящего в левой части операции доступа, и кода, аналогичного коду доступа к глобальной переменной.

2.5.4 Доступ к нестатическим членам классов

Синтаксически доступ к нестатическим членам класса реализуется с помощью операций доступа к членам классов. Случай доступа к члену из тела функции-члена по имени можно считать частным случаем использования операции доступа `->`, у которой в левой части находится указатель `this`. Код для доступа генерируется на основании статического смещения члена в объекте класса.

Согласно семантике языка, член однозначного базового класса может использоваться так же, как и член, определенный в самом классе. В этом случае перед генерацией кода доступа неявно генерируется код преобразования указателя или ссылки, стоящей в левой части операции доступа, к указателю или ссылке на класс, в котором этот член определен. После этого код для доступа генерируется, как и в первом случае, на основании смещения члена.

2.5.5 Вызов функций-членов класса

Функции-члены класса могут быть статическими, виртуальными и обычными. Вызовы этих трех типов функций рассматриваются отдельно так как для их вызова код генерируется по различным правилам.

Статические функции-члены реализуются как глобальные функции, неявно имеющие внешнее связывание. Код для их вызова ничем не отличается от кода вызова обычной глобальной функции.

Обычные функции-члены вызываются посредством операции доступа к члену класса, включая случай неявного использования операции доступа с указателем `this` в левой части. В теле функции-члена объект класса, для которого функция-член вызвана, доступен через указатель `this`. Аналогично членам классов функции-члены однозначных базовых классов доступны в производном классе. Код для их вызова содержит преобразование указателя или ссылки из левой части операции доступа к указателю или ссылке на базовый класс. При генерации вызова указатель, возможно неявно преобразованный, передается в функцию в качестве первого скрытого параметра. В теле функции этот скрытый первый параметр используется как указатель `this`.

Код для вызова виртуальной функции-члена генерируется во многом аналогично генерации кода вызова обычной функции-члена. Указатель из левой части операции доступа аналогичным образом может преобразовываться к указателю на базовый класс. Так же, как в обычную функцию-член в качестве первого скрытого параметра передается указатель `this`. Для обычной функции-члена адрес вызываемой функции является константой, известной на этапе компиляции (для генерации объектного кода достаточно знать ее внешнее имя). Для виртуальной функции-члена это не так: адрес вызываемой функции берется из таблицы, на этапе компиляции известен только индекс в таблице. Доступ к самой таблице производится через скрытый член *vf_ptr* объекта, стоящего в левой части операции доступа. Таким образом реализуется зависимость вызываемой функции от динамического типа объекта, находящегося в левой части. Этот механизм часто называют механизмом *позднего связывания*, подразумевая, что он напоминает механизм связывания, но отрабатывает на этапе выполнения. Даже если в левой части операции доступа к функции-члену находится указатель `this`, динамический тип объекта может быть неизвестен, так как вызывающая функция сама может быть вызвана как функция подобъекта базового класса. Первый (скрытый) параметр также нуждается в дополнительной обработке. В вызываемой функции он должен указывать на объект класса *L*, в котором

определена функция-член, а не на объект класса *M*, для которого эта функция вызвана. При этом возможны ситуации, когда $M \Rightarrow L$, равно как возможны ситуации, когда $L \Rightarrow M$. Поэтому из таблицы виртуальных функций кроме самого адреса вызываемой функции, берется дополнительное смещение, которое прибавляется к указателю, стоящему в левой части операции доступа. Дополнительное смещение может быть как положительным, так и отрицательным, в зависимости от расположения классов в графе наследования. Полученная сумма и передается в качестве первого скрытого параметра.

Псевдокод для вызова виртуальной функции–члена выглядит следующим образом: `ptr->f(par)` – исходный код вызова преобразуется в `((ptr->vf_ptr)[i].fun)(ptr+(ptr->vf_ptr)[i].offset,par)`.

Как видно из приведенного псевдокода, дополнительные операции, которые выполняются для вызова виртуальной функции, по сравнению с вызовом обычной функции-члена, включают сложение и разыменование для вычисления общего подвыражения `(ptr->vf_ptr)`, разыменование и сложение для вычисления адреса вызываемой функции, разыменование и два сложения для вычисления скрытого параметра.

В некоторых частных случаях, например в телах конструкторов и деструкторов, динамический тип объекта, именуемого указателем `this`, известен на этапе компиляции. В таких случаях виртуальные функции можно вызывать как обычные функции-члены.

2.5.6 Указатели на члены и функции-члены классов

Указатель на члены классов реализуется как структуры, содержащие два поля *index* и *offset* целочисленного типа. Значение типа указатель на член класса *A::b*, где *B* – класс, в котором определен член *b*, в первом поле содержит индекс виртуального базового класса *[p]**, где *p* – некоторый путь из вершины *A* в вершину *B* в графе *Inh*. Если *[p]*=A*, то в этом поле содержится специальное значение –1. Во втором поле содержится сумма смещений, приписанная ребрам пути *fix(p),* и смещения $offset_B(b)$. Специальное нулевое значение указателя на член класса представляется в виде пары *(-1,-1)*. Благодаря тому, что для реальных членов класса поле *offset* всегда содержит неотрицательное число, нулевое значение всегда можно отличить от указателя на существующий член класса.

Для генерации кода доступа к члену класса через указатель на член необходимо найти смещение именуемого члена в объекте класса *A*. Это

смещение вычисляется в процессе выполнения как сумма $offset + \begin{cases} 0, index = -1 \\ vbase[index], index \neq -1 \end{cases}$

где *vbase[index]* - элемент таблицы смещений виртуальных базовых классов с номером *index*. Доступ к указателю на таблицу смещений виртуальных базовых классов производится по смещению $offset_A(vbase_ptr(A))$ в объекте класса *A*. Если класс *A* не имеет виртуальных базовых классов, то поле *index* никогда не используется, и его можно исключить из генерируемого кода, а также из самой структуры.

Указатели на функции-члены классов реализуются как структуры *(index,offset,func,vindex)*. Для функции-члена *A::b,* определенной в классе *B,* поля структуры содержат следующие значения:

- *vindex* – индекс виртуальной функции в таблице класса *A*, если функция-член является виртуальной, или –1 в противном случае.
- *func* – адрес функции-члена *B::f* если эта функция невиртуальная или нулевое значение в противном случае.
- *offset* и *index* – значения определяемые по тем же правилам, что и для указателей на члены класса.

Нулевое значение указателя на функцию-член содержит произвольные значения в полях *vindex*, *func*, *index* и –1 в поле *offset*.

При обращении к функции-члену через указатель ее адрес во время выполнения вычисляется по следующей формуле $\begin{cases} vftbl[vindex].func, vindex \neq 1 \\ \begin{cases} func, offset \neq -1 \\ 0, offset = -1 \end{cases}, vindex = -1 \end{cases}$,

а дополнительное смещение скрытого параметра по формуле $\begin{cases} vftbl[vindex].offset, vindex \neq -1 \\ offset + \begin{cases} 0, index = -1 \\ vbase[index], index \neq -1 \end{cases}, vindex = -1 \end{cases}$.

2.5.7 Преобразования указателей на члены и функции-члены классов

По правилам языка указатели на члены базовых классов могут быть преобразованы в указатели на члены однозначных производных классов. Если указатель на член базового класса преобразуется в указатель на член производного класса, то необходимо учитывать различные варианты

преобразования в зависимости от отношений между классами и от значения полей указателя.

Если преобразование производится от указателя на член базового класса *B* к указателю на член класса *D*:

Если $index \neq -1$, то при преобразовании изменяется только поле *index*. Значение этого поля в преобразованном указателе выбирается из таблицы, создаваемой компилятором для преобразования указателей на члены этой пары классов. *i*-ым элементом таблицы является $index_D(all_vbases(B)[i]))$, $0 \leq i < |all_vbases(B)|$. Это определение корректно, так как все виртуальные базовые классы класса *B* также являются виртуальными базовыми классами класса *D*. Такие таблицы также поддаются оптимизациям аналогичным оптимизациям таблиц виртуальных функций смещений виртуальных базовых классов. Например, если *B_major(D)=B, i*-ый элемент таблицы будет равен *i,* и фактически перенумерация по таблице не требуется.

Иначе, если *index=-1*, то способ преобразования зависит от отношений между классами *B* и *D*. Рассмотрим путь p из вершины *D* в вершину *B* в графе *Inh*. При преобразовании к полю *offset* прибавляется сумма смещений, приписанных ребрам пути *fix(p)*. Если *[p]** совпадает с *B*, то поле *index* не изменяется, иначе в ему присваивается значение $index_D([p]^*)$.

При преобразовании указателей на функции-члены дополнительно выполняется преобразование поля *vindex*. Так же, как и перенумерация индексов виртуальных базовых классов, перенумерация индексов виртуальных функций производится по таблицам, которые строятся на этапе компиляции. При этом индекс *–1* всегда преобразуется в *–1*, для этого в таблице перекодирования генерируется дополнительный элемент.

2.5.8 Реализация операции `typeid`

Оператор `typeid` ([1], раздел 5.2.8) возвращает ссылку на объект класса `type_info`, соответствующего типу аргумента операции. Для неполиморфных типов эта операция имеет очень простую реализацию, достаточно просто вернуть ссылку на статический, сгенерированный компилятором объект. Такие объекты

генерируются в процессе компиляции для каждого типа, объекты которого являются операндами операции `typeid`.

Если операндом является объект полиморфного типа, операция должна вернуть ссылку на объект класса `type_info`, соответствующий динамическому типу объекта. Доступ к этому объекту производится через таблицу виртуальных функций аргумента (по определению полиморфного типа он имеет виртуальные функции, а следовательно, в рассматриваемой реализации имеет таблицу виртуальных функций).

Так как во время создания и разрушения объектов для конструкторов и деструкторов базовых классов используются отдельные таблицы виртуальных функций, семантика этой операции, специфицированная в разделе 12.7 стандарта [1], легко удовлетворяется во время создания и разрушения подобъектов,.

2.5.9 Реализация операции `dynamic_cast`

Оператор `dynamic_cast` ([1], раздел 5.2.7) представляет известную сложность для реализации; для ее реализации потребовалось использовать несколько функций библиотеки поддержки времени выполнения. Промежуточный код, генерируемый для оператора `dynamic_cast<d>(exp)`, зависит от типа *d* и типа выражения *exp*.

Простейшим является случай, когда тип *d* является указателем или ссылкой на класс *D*, а тип *exp* – указатель или ссылка на класс *B,* совпадающий или являющийся однозначным базовым для класса *D*. В этом случае семантика операции ничем не отличается от семантики преобразования указателей на классы, способ реализации которого уже описан выше.

Более сложный случай операции – преобразование к указателю на тип `cv void*`. В этом случае операция применима только к указателям на полиморфные классы. Объекты этих классов имеют таблицы виртуальных функций, через которые легко получить доступ к смещению подобъекта, сгенерированному как описано выше в разделе 2.4.7. Правила языка требуют, чтобы нулевое значение указателя преобразовывалось в нулевое. Таким образом, в этом случае для преобразования генерируется следующий Си-подобный псевдокод:

```
exp ? exp+exp->vtbl_ptr->subobject_offset : 0.
```

Следующий случай – преобразование указателя на класс *B* к указателю на производный класс *D*. В этом случае возникают сложности, так как класс *B* может быть виртуальным или неоднозначным базовым классом класса *D*. В этом случае

необходимо воспользоваться информацией из дескрипторов подобъектов, доступных через объекты типа `type_info`.

На приведенном ниже примере графов *Sub(M)* показаны различные варианты корректных преобразований выполняемых `dynamic_cast`. Предполагается, что во всех случаях наследование имеет public доступность.

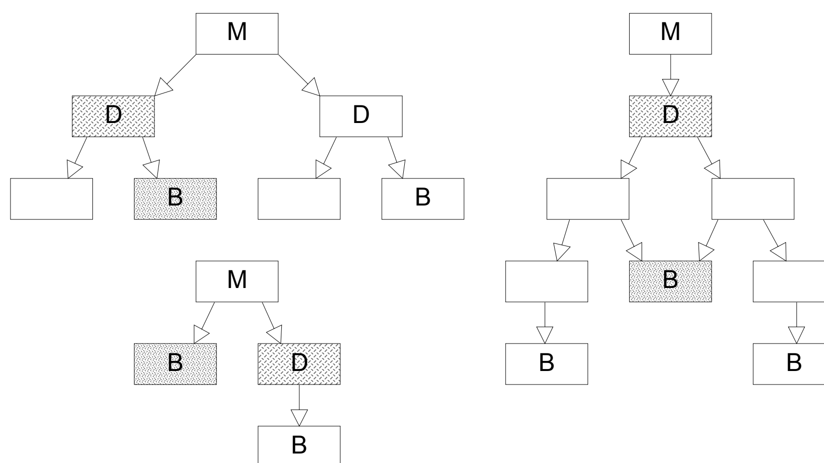


Рис. 2-11. Различные варианты возможных преобразований `dynamic_cast`.

Дополнительные требования, которые накладываются на доступность подобъектов базовых классов, можно сформулировать следующим образом: если динамическим типом объекта является *M*, то либо в объекте класса *D* преобразуемый объект *B* является public доступным, либо подобъект класса *B* является public доступным в объекте класса *M* и найденный подобъект класса *D* является public доступным в классе *M*.

Приведем следующий пример:

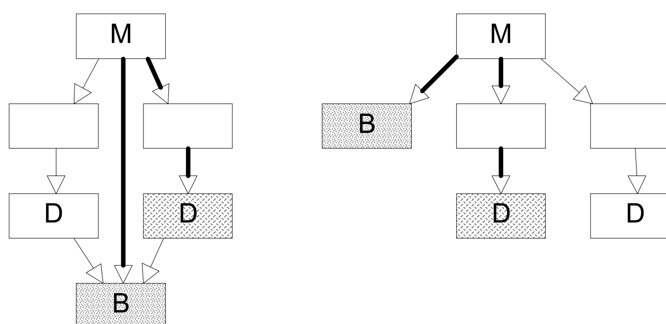


Рис. 2-12. Отношение доступности и оператор `dynamic_cast`.

В приведенном примере ребра соответствующе public наследованию выделены более толстыми линиями. Как видно из примера отношение доступности может разрешать неоднозначности при выполнении оператора `dynamic_cast`.

Для выполнения преобразования требуется доступ к объектам типа `type_info` трех классов: *B*, *D* и *M*. Первые два класса известны во время компиляции, а значит доступ к соответствующим объектам `type_info` не представляет сложности. Доступ к объекту типа `type_info` класса *M* возможен так же как при выполнении операции `typeid(exp)`. Через объекты типа `type_info` можно получить доступ к спискам описателей их подобъектов, в которых содержится информация о смещении и доступности подобъектов. Этой информации достаточно, чтобы реализовать функцию библиотеки поддержки времени выполнения, выполняющую преобразование. Параметры этой функции: `exp, &(exp->vtbl), typeid(B), typeid(D)`.

Эта функция реализует преобразование для указателей на классы. Аналогичная функция преобразования для ссылок отличается от функции преобразования указателей лишь тем, что в том случае, когда первая функция возвращает 0, вторая генерирует исключение типа `bad_cast`.

2.5.10 Реализация определения типов исключительных ситуаций

Для Си++ известно два основных способа реализации исключительных ситуаций ([43], [44]). Один из них называют методом динамической регистрации ([45]). В компиляторе использован аналогичный метод; подробно его реализация описана в [73]. В этом разделе сам метод и его особенности не будут рассматриваться, но будет описано как во время выполнения передается информация о типах активных исключений и метод их сравнения во время перехвата.

Во время активизации исключения значение исключения копируется в буфер, в буфере также сохраняется информация о типе исключения, а именно сохраняется указатель на объект `type_info` типа исключения. Значение и указатель становятся доступны при перехвате исключения. После перехвата необходимо сопоставить тип исключительной ситуации с набором типов, определенных в заголовках оператора перехвата. Прямое сравнение типов в такой ситуации не удовлетворяет семантике языка, так как при перехвате исключения возможны некоторые виды неявных преобразований, в том числе преобразование указателя на производный класс к указателю на базовый класс.

Неявные преобразования, допустимые при перехвате исключительной ситуации представляют сложность только в случае исключений имеющих типы классов или указателей на классы. Во всех остальных случаях для типа стоящего в заголовке оператора перехвата можно указать конечное множество типов

подходящих для этого оператора. Для указательных и ссылочных типов это множество получается удалением из типа имеющихся cv-квалификаторов (подробнее операции с типами описаны в четвертой главе этой диссертации), для остальных типов это множество состоит из одного элемента. Таким образом, сопоставление типов можно выполнить использовав несколько сравнений на равенство для объектов `type_info` из множества и объекта `type_info`, описывающего тип активного исключения.

В случае объектов классов или указателей на классы реализация несколько сложнее, так как у любого класса может быть неограниченное число производных классов, и не все производные классы могут быть известны в обрабатываемой единице трансляции. В этих случаях для сопоставления типов используется вспомогательная функция библиотеки времени выполнения, которая принимает в качестве аргументов указатели на объекты `type_info` текущего активного исключения и типа требуемого обработчиком. Эта функция проходит по списку описателей подобъектов базовых классов, доступному через первый объект, и сравнивая их со вторым. При этом легко учитываются требования стандарта языка предъявляемые к однозначности и доступности базового класса. Результатом работы этой функции является ответ на вопрос о соответствии типов и смещение искомого подобъекта в объекте текущего активного исключения.

2.6 Заключение. Выводы главы 2

В этой главе обсуждался способ реализации механизмов классов и виртуальных в компиляторе языка Си++. Предложеный способ основан на генерации на этапе компиляции таблиц, которые затем используются механизмами выполнения. Такой подход позволяет генерировать достаточно простой и эффективный код, реализующий семантику конструкций языка Си++.

Основные результаты второй главы могут быть сформулированы следующим образом:

1. Разработан метод реализации объектов классов в компиляторе Си++ с точки зрения генерации машинно-независимого низкоуровневого промежуточного кода, основанная на предложенной в первой главе модели.
2. Предложен способ реализации множественного и виртуального наследования. Реализация виртуального наследования основана на генерации таблиц смещений виртуальных базовых классов.
3. Описан метод генерации кода для доступа к членам классов и членам базовых классов.

4. Разработан метод реализации преобразований указателей и ссылок на классы учитывающий особенности множественного и виртуального наследования.
5. Исследован метод реализации виртуальных функций основанный на генерации таблиц.
6. Дано определение доминирования для виртуальных функций и описан способ его выявления.
7. Предложены методы определения абстрактности классов и неявной виртуальности функций.
8. Описан метод реализации указателей на члены классов и их преобразований.
9. Предложены методы реализации операторов `typeid` и `dynamic_cast` с использованием объектов расширенного класса `type_info`.
10. Определена сложность алгоритмов генерации таблиц и даны оценки их размера.
11. Описан метод реализации динамического определения типов для исключительных ситуаций.
12. Разработаны структуры данных компилятора, используемые механизмами времени компиляции.

Глава 3. Поиск имен

3.1 Введение

В том или ином виде механизмы поиска имен присутствуют во всех компиляторах ЯВУ [88]. Язык Си++ обладает достаточно сложным набором правил поиска имен. Основные правила поиска имен сформулированы в стандарте [1] в разделах 3.4 и 10.2.

Основной задачей поиска имен в компиляторе является сопоставление идентификатору сущности, *именуемой* этим идентификатором. В случае рассматриваемой реализации, важнейшей задачей механизма поиска имен является сопоставление идентификатору исходного текста программы элемента семантических таблиц (*семантического слова*), описывающего сущность, *именуемую* этим идентификатором. Такое сопоставление, выполненное на ранних этапах трансляции (во время или до построения деревьев), позволяет существенно упростить все дальнейшие фазы трансляции. В некоторых случаях для разрешения синтаксических неоднозначностей поиск имен должен производиться даже до синтаксического разбора ([85]).

В качестве примера рассмотрим следующий фрагмент исходного текста на языке Си++:

```
int i;
void f(int &i)
{
        i=1;
}
```

В данном примере определяется две переменных с именем *i*. Задачей механизма поиска имен в данном примере является сопоставление идентификатору *i*, используемому в выражении в теле функции, элемента семантических таблиц, соответствующего этому вхождению идентификатора.

Первые два вхождения идентификатора *i* в этом примере является частями описания сущностей (глобальной переменной и параметра). Такие вхождения ниже будем называть *определяющими*, а последующие, ссылающиеся на уже объявленные объекты, будем называть *использующими*.

В стандарте языка определяется принципиальное для описания процесса поиска имен понятие *области действия* (scope) [1, раздел 3.3]. В рассматриваемом компиляторе этому понятию соответствует *блок* семантических таблиц ([85]). Далее считается, что ссылку на дескриптор блока можно использовать как уникальный идентификатор области действия.

При проектировании компилятора было принято решение о разделении механизмов поиска имен и механизмов построения семантических таблиц. Первая и основная причина такого решения заключается в сложности и нерегулярности правил языка. Можно предположить, что отделение механизма поиска имен от механизма построения и управления семантическими таблицами позволит упростить и сделать более регулярным как поиск имен, так и сами семантические таблицы, которые освобождаются от необходимости хранить большое количество дополнительной информации.

Второй причиной является другое проектное решение, связанное с местом механизма поиска имен в общей структуре компилятора. Это решение заключается в том, что поиск имен происходит, если это возможно, на очень ранней стадии обработки исходного текста, а именно – непосредственно после лексического анализа. В тех случаях, когда такой поиск невозможно осуществить полностью, то есть невозможно сопоставить имени некоторую сущность, на ранних этапах трансляции поиск производится частично. Это в свою очередь обусловлено сложностью синтаксиса языка и наличием в нем неоднозначностей. Подробное описание взаимодействия механизма поиска имен с другими частями компилятора приводится в главе 1 [85].

Механизм поиска имен тесно связан и активно использует механизм семантических таблиц, в частности активно используются понятия «текущего контекста», «блока» и «дисплея», которые не описываются в данной работе достаточно подробно (даются лишь краткие пояснения к этим понятиям необходимые для описания механизма). Более подробно эти механизмы описаны в главах 1 и 3 работы [85].

Основной задачей этой главы является разработка и анализ механизма поиска имен для компилятора Си++. Механизм поиска включает в себя алгоритмы поиска и структуры данных, используемые этими алгоритмами. Механизм должен обеспечивать, во-первых, возможность поиска имен для всех возможных в языке Си++ случаев и соответствовать всем требованиям, накладываемым стандартом. Во вторых, должен быть достаточно эффективным для того, чтобы быть пригодным для использования в промышленном компиляторе, что, в свою очередь, требует от механизма возможности эффективно работать с большими исходными текстами, содержащими большое количество идентификаторов. Третьей целью является максимальная, насколько это возможно, простота самого механизма, минимальная его зависимость от языка Си++ с той точки зрения, что построенный на описанных ниже принципах механизм может применяться и в

других компиляторах, в том числе в компиляторах с других языков программирования.

Как и в первой главе, основное внимание при описании разработанного механизма будет уделяться используемым в нем структурам данных.

3.2 Особенности поиска имен в языке Си++

Ниже будут рассмотрены основные отличия правил поиска имен в Си++ от правил поиска имен в других языках программирования, таких как Паскаль и Си.

Правила поиска имен в языке Си++ сформулированы в разделах 3.4 и 10.2 стандарта [1]. Обсуждая эти правила, мы не будем пытаться переформулировать их, но сосредоточимся на следствиях из этих правил, которые позволят наиболее полно представить сложность задачи и необходимость формального подхода к ее решению.

В большинстве существующих языков программирования имеются понятия *области действия* имени, *видимости* имен и *скрытия* имен. Наиболее распространенный способ определения этих понятий основан на концепции *вложенных областей действия*. Вложенные области действия имеют различный синтаксис в различных языках программирования, и в терминологии различных языков программирования называются по-разному, но в то же время обладают весьма сходной семантикой ([88]). Далее, во избежание именно терминологических трудностей будет использоваться терминология языка Си++ [22].

Для всех определений вложенных областей действия в различных языках программирования характерны следующие свойства.

A1. *Лексическая ограниченность*. Для каждой области действия в программе можно указать где (после какой лексемы) область действия начинается и где заканчивается.

A2. *Вложенные области действия образуют дерево вложенности*. А именно, существует единственная *глобальная* область действия, началом которой является начало исходного текста единицы трансляции и концом – конец единицы трансляции. Логично считать эту область действия корнем дерева. Для любой области действия существует единственная непосредственно *объемлющая* область действия. Ребра дерева направлены от *объемлющей* области действия к *вложенной*. Языки программирования, как правило, не накладывают никаких ограничений на максимальную высоту или арность такого дерева.

A3. *Для любой именованной сущности в единице компиляции может быть указана наиболее вложенная область действия, в которой эта сущность определена*. Как правило, в таких случаях имя сущности *действует* с момента своего определения в своей области действия до конца этой области действия.

A4. *Имя сущности, описанной во вложенной области действия, скрывает имя сущности из объемлющей области действия.* Таким образом, для каждой именованной сущности можно говорить об *области видимости*, той части программы, в которой имя сущности действует и не скрыто именем из вложенной области.

A5. *Имя сущности, описанной во вложенной области действия никак не отражается на использовании того же имени в объемлющей области действия.* Другими словами, с завершением области действия, в которой имя описано, оно становится недоступным, либо для доступа к сущности по ее имени используется специальный синтаксис, например операция доступа к члену структуры.

A6. *Язык накладывает ограничение на определение одноименных сущностей в одной области действия.* Как правило, в языках программирования не допускается определение в одной области действия двух различных одноименных сущностей, либо эти сущности должны принадлежать к различным *пространствам имен*.

A7. *Поиск имен на этапе семантического анализа.* В некоторых случаях определение области действия, в которой необходимо производить поиск имен, не может быть выполнено на этапе лексического или синтаксического анализа. Например, в случае доступа к члену класса (полю структуры) необходимо знать тип выражения в левой части операции доступа.

Для Си++ эти общие свойства многих языков программирования лучше всего проиллюстрировать на примере вложенных блоков. Блок образует область действия. Он начинается с открывающей операторной скобки **{** и заканчивается закрывающей операторной скобкой **}**. Любое имя, определенное в блоке, принадлежит его области действия. Действие имени в блоке начинается непосредственно за лексемой, определяющей именованную сущность. Блоки могут быть вложенными; в этом случае две сущности, описанные в разных блоках, могут иметь одинаковые имена. Во вложенном блоке имя связывается с сущностью, объявленной в наиболее вложенном блоке, содержащем текущий. Например:

```
{
        struct A {
              int a;
        } *ap
        int I;
        int X;
        {
              int I;
              // Имя I в этом блоке означает переменную,
              // объявленную в этом блоке
              // Имя I из объемлющего блока скрыто
              int X=X;
              // Действие имени X во вложенном блоке
              // начинается до лексемы '=' и потому
              // переменная инициализируется своим
              // собственным, еще не определенным значением.
        }
        ap->a = 1;
        // для доступа к члену структуры по имени
        // используется оператор ->,
        // Область действия, в которой производится поиск
        // имени a, определяется на основании типа выражения
        // в левой части оператора ->
}
```

Проиллюстрировав общие для многих языков программирования свойства, которыми также обладает Си++, перейдем к свойствам, специфичным для этого языка.

B1. *Различные виды областей действия*. В языке Си++ есть несколько видов областей действия, которые различаются по своим свойствам. Виды областей действия перечислены в разделе 3.3 стандарта [1].

B2. *Ограничители области действия.* Большинство областей действия в Си++ ограничены лексемами { и }, это верно не только для области действия локального блока. Исключения составляют глобальная область действия, ограниченная началом и концом файла, и область действия имен параметров функции, ограниченная лексемами (и). Лексемы (,), { и } по синтаксису Си++ используются не только для ограничения областей действия. Таким образом, нужно признать необходимость того, что управлять переключением текущих областей действия должен механизм, работающий на основе информации, которая доступна только после или во время синтаксического анализа. Из этого следует, что лексический анализатор, и тесно связанный с ним механизм поиска имен, не могут получить всю необходимую для корректного поиска имен информацию самостоятельно, в частности не могут самостоятельно управлять дисплеем (структурой, отражающей вложенностью блоков, [85]). Пример:

```
class A {    // в данном случае область действия
```

```
        int a;      // ограничена лексемами { }
};

int f (int x, int y);
// в данном случае область действия
// ограничена лексемами ( )

enum E { e1, e2 };
int a[] = {1, 2};
// в данных случаях лексемы { и } не
// образуют отдельной области действия

int (*p)[10];
// в данном случае лексемы ( и )
// не образуют отдельной области действия
```

В3. Ограничения на вложенность различных видов областей действия. Области действия имен в Си++ не могут вкладываться друг в друга произвольным образом. В частности, область действия имен параметров функции может содержать в себе только область действия имен параметров другой функции.

В4. Некоторые именованные сущности, объявленные во вложенной области, действуют, как будто их имена объявлены в объемлющей области действия. Примерами могут служить объявление метки во вложенном блоке в теле функции. Областью действия имени метки является вся функция. Таким образом, в языке Си++ есть исключения из свойств А4 и А6, описанных выше. Пример:

```
void f ()
{
        {
                label: ;
        }
        goto label;
        // Имя метки, объявленной во вложенном
        // блоке, действует в объемлющем блоке.
}
```

В5. В языке Си++ существует несколько видов пространств имен. Так, имена классов и перечислений находятся в пространствах имен, отдельных от пространства имен переменных и функций. Метки в теле функции также имеют отдельное пространство имен.

```
class A {
};
int A;          // объявляется переменная с тем же именем,
                // что и класс

A: ;            // Объявляется метка с тем же именем,
                // что и класс и переменная.
```

В6. В языке Си++ допускается существование нескольких одноименных сущностей в одной области действия в одном пространстве имен.

Примером могут служить имена совместно используемых («перегруженных», overloaded) функций. При этом нельзя сказать, что имя функции, описанной лексически последней, скрывает имена всех функций описанных до нее. Имена функций действуют одновременно.

```
void f(int);
void f(char);      // объявление двух одноименных функций

f(1);         // Вызов f(int)
f('c');       // Вызов f(char)
              // Имена функций не скрывают друг друга
```

В7. *Использование имен из различных пространств имен может быть идентичным по синтаксису.* Так, в одной области действия могут быть описаны класс, а затем переменная с тем же именем. Имена этих сущностей принадлежат к различным пространствам имен. При этом использование имени после описания класса, но до описания переменной, означает класс, но после описания переменной означает переменную. Имя переменной скрывает имя класса при неуточненном доступе. Из этого также следует, что скрытие имени может произойти даже в той области действия, в котором имя описано.

```
class A {
};
A a;          // A разрешается как имя класса
int A;        // объявляется сущность с тем же именем,
              // что и класс
…
A = 1;        // A разрешается как имя переменной.
              // Имя переменной скрывает имя класса.

class A a1;
// имя класса доступно для уточненного доступа
void f ()
{
      int A;
      class A a;
      // имя класса доступно для уточненного доступа даже из
      // вложенной области действия
}
```

В8. *Области действия сами могут быть именованными.* Наиболее интересные для последующих рассмотрений типы областей действия, а именно область действия класса и область действия namespace сами являются подчиненными именованным сущностям, а именно к классам и namespace. Такое отношение можно назвать дуальным ([85], глава 3):

```
class A {
              // Класс имеет собственную область
              // действия, в то же время он является
              // именованной сущностью объемлющей
              // области действия
```

```
};

namespace A {
            // namespace имеет собственную область
            // действия, в то же время является
            // именованной сущностью объемлющей
            // области действия
}

class {
} xA;
namespace {
};      // Классы и namespace также могут быть
        // безымянными
```

В9. *Возможен доступ к имени, объявленному в другой области действия.* Для такого доступа есть множество ограничений. Существенным является тот факт, что такой доступ возможен из области действия, не являющейся вложенной в область действия, в которой имя определено, при помощи явной синтаксической конструкции, указывающей сначала область действия, по имени, а уже затем указывающее имя для поиска в этой области действия.

```
namespace A {
      int a;
};
namespace B {
class A {
      static int a;
};

::A::a = 1; // Доступ к переменной a члену namespace A
A::a = 1;   // Доступ к члену a класса A
};
```

В10. *Доступ к имени из другой области действия возможен без использования специального синтаксиса.* В теле функции-члена класса, описанной вне тела класса, возможен доступ по имени к другим членам класса. При этом такой доступ синтаксически никак не отличается от доступа к локальной переменной функции.

```
class A {
      int x;
      void f();
};

void A::f()
{
      int y;
      x = 1;      // доступ к члену x класса A
                  // синтаксически ничем не
      y = 1;      // отличается от доступа к локальной
                  // переменной y.
}
```

В11. Определение одного пространства имен может быть лексически разрывным. Это возможно только в случае именованного namespace; все имена, описанные в продолжении уже объявленного именованного namespace, находятся в той же области действия, что и имена, объявленные во всех предыдущих частях namespace.

В12. *using-директивы и using-объявления позволяют ссылаться на имена, описанные в другой области действия.* Различные варианты этой директивы могут вводить в область действия, в которой директива использована, отдельное имя из другой области действия, либо все имена из другой области действия.

```
namespace A {
      void f (int);
};

namespace B {
      using A::f;
      void g() { f(1); }
            // используется функция A::f,
            // имя которой доступно благодаря
            // использованию using-объявления
};

namespace C {
      using namespace A;
      // все имена, определенные в namespace A
      // доступны в текущем namespace C.
      class f { };
      // имя класса f скрывает имя функции f
};

namespace A {
      // продолжение области действия
      // namespace A
      void f (char);
};

namespace B {
      void h() { f('a'); }
      // используется A::f(char) несмотря на то,
      // что определение этой функции дано
      // позднее using-объявления.
      // Таким образом, директива делает видимыми
      // имена, но не сущности по их имени.
};
```

В13. *Использование имени может оказаться неоднозначным.* В некоторых случаях нельзя указать последовательность областей действия, в которых нужно искать определение имени. Так, если бы в языке Си++ правила для областей действия ограничивались правилами А1-А7, можно было бы сказать,

что поиск имен должен происходить последовательно во всех областях действия, начиная с текущей, вверх до корня дерева. Примером неоднозначности имени может служить определение одноименных членов в двух базовых классах и попытка получить доступ по этому имени в теле функции-члена производного класса. Этот случай неоднозначности отличается от другого вида неоднозначности, связанной с неоднозначностью самого базового класса, при которой имя может разрешаться однозначно.

```
class A {
        int a;
        static int b;
};
class B {
        int a;
};

class C: A, B {
        void f() { a=1; }
        // имя a неоднозначно в классе C
};

class D1: A {};
class D2: A {};
class E: D1, D2 {
        void f() { a=1; b=1; }
        // имена a и b не являются неоднозначными;
        // базовый класс A является неоднозначным,
        // поэтому использование имени a ошибочно,
        // но использование имени b допустимо
};
```

B14. *Последовательность поиска имени во вложенных областях действия совпадает с порядком их вложенности*. Однако из этого не следует, что поиск не ведется ни в каких других областях действия. Более того, даже если последовательность поиска строго определена, и имя не может быть неоднозначным, такая последовательность может быть нетривиальной в том смысле, что после поиска во вложенной области действия и перед поиском в непосредственно лексически объемлющем ее блоке поиск может производиться в других областях действия.

```
class A {
        void g();
};
void A::g() { x = 1; }
        // имя x сначала ищется в области действия
        // функции-члена A::g, затем в области
        // действия класса A, и лишь затем в
        // глобальной области действия
```

B15. *Область действия для поиска имени может определяться уже на этапе семантического анализа.* Правило поиска имен, называемое Koeing lookup ([1]

раздел 3.4.2), требует производить поиск имени функции в области действия, не объемлющей текущую даже в случае отсутствия из областей действия, объемлющих текущую, любых ссылок на область действия, содержащую функцию.

```
namespace A {
      class B {};
      void f(B);
};
A::B x;
void g() { f(x); }
      // поиск имени функции f производится в
      // namespace A, так как тип параметра A::B
      // определяется в области действия
      // namespace A.
```

B16. Использования имени лексически может предшествовать его определению. В случае определения функции-члена класса в теле класса все имена членов и типов, описанные в том же классе, должны быть доступны в теле функции-члена.

```
int a;
class A {
      void f () { a=1; }
      // использование имени a члена класса
      // лексически предшествует его определению
      int a;
};
```

B17. *У одной сущности может быть несколько определяющих вхождений имени.* Примерами могут служить предварительные объявления классов или внешние (external) объявления объектов.

```
class A;
extern int x;
class A {};
int x;
```

Можно также указать некоторые свойства программ, написанных на языке Си++, явно не сформулированные в стандарте языка, но характерные для многих программ и обусловленные сложившимся стилем программирования на этом языке.

C1. Частое использование имен из другой области действия. Поддерживаемый в языке Си++ объектно-ориентрованный подход предполагает частое использование имен, объявленных в области действия класса, вне этого класса. Кроме того, во вложенных областях действия часто используются имена из глобальной области действия или области `namespace std`, где

определены все сущности, составляющие стандартную библиотеку Си++ ([1], глава 17).

C2. *Имена функций-членов классов часто совпадают в различных классах.* Так, полиморфизм - одно из основных понятий объектно-ориентированного программирования - поддерживается в Си++ посредством механизма виртуальных функций. Для этого механизма существенно, что перекрывающие функции производных классов имеют те же имена, что и перекрываемые функции базовых классов.

C3. *Неравномерность распределения имен по областям действия.* Количество имен, объявленных в различных областях действия, распределено очень неравномерно, а именно большое количество имен определяется в глобальной области действия и области действия namespace, а в областях действия локальных блоков во многих случаях не объявляется ни одного имени.

C4. *Большое количество областей действия.* В языке Си++ имеется отдельная область действия для имен параметров каждой функции, даже в том случае, когда задается только предварительное объявление (прототип) функции. В то же время стиль программирования на языках Си и Си++ предполагает наличие в единице компиляции большого количества предварительных объявлений функций, большая часть которых в единице трансляции реально не используется. Объектно-ориентированное и обобщенное программирование предполагает использование в программах большого количества классов, шаблонов классов и относительно небольших функций-членов (accessors), единственное назначение которых – обеспечить доступ (по чтению или по записи) к приватным членам классов. Кроме этого такие конструкции как циклы и условные операторы имеют отдельные области действия типа локальный блок.

3.3 Реализация механизма поиска имен в одной области действия

После обсуждения требований, предъявляемых к механизму поиска имен, и важнейших свойств и особенностей структуры программ на Си++ можно перейти к обсуждению реализации этого механизма.

3.3.1 Традиционный способ поиска имен и его недостатки

Для того чтобы наиболее полно рассмотреть предложенный механизм, имеет смысл рассматривать также его отличия от механизмов поиска имен, часто применяемых в компиляторах других языков программирования и подробно описанного в классической литературе [14].

Первый способ основан на понятии вложенных областей действий. Для каждого разбираемого в процессе компиляции блока или другого типа области действия создается небольшая хеш-таблица, в которую по мере разбора заносятся имена сущностей, определяемых в разбираемом блоке. Синтаксическая вложенность блоков отражается с помощью структуры "*дисплея*" - стека ссылок на хеш-таблицы блоков. Коллизии в хеш-таблице разрешаются с помощью "метода цепочек". При поиске идентификатора сначала вычисляется его хеш-значение (определяемое его литеральным представлением). Затем последовательно перебираются блоки, ссылки на которые в настоящий момент находятся в дисплее, начиная с вершины стека, и происходит поиск имени в хеш-таблицах, соответствующих этим блокам. Так как коллизии разрешаются методом цепочек, производится сравнение литеральных представлений имен для каждого имени находящегося в цепочке.

Для иллюстрации рассмотрим следующий пример:

```
void p(int a, int i, int m)
{
      {
            int i, b;
            // на рисунке представлено состояние
            // дисплея и
            // хеш-таблиц в момент, когда разбор
            // находится в этой точке исходного
            // текста программы
      }
}
```

Предположим для определенности, что хеш-значения идентификаторов `h(a)=1, h(b)=2, h(i)=h(m)=4, h(p)=6.`

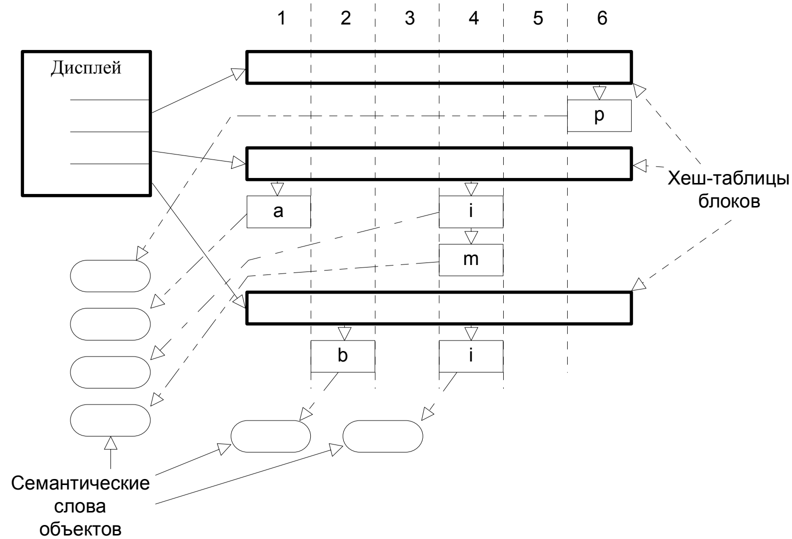


Рис. 3-1. Классическая схема поиска идентификаторов

При выходе из лексического блока ссылка на хеш-таблицу блока удаляется из дисплея, после чего возможно и удаление самой таблицы. В случае обычного блока в теле функции в процессе дальнейшей трансляции поиск идентификаторов в закончившемся блоке более не потребуется. В случае области действия структуры ссылка на хеш-таблицу блока сохраняется в одном из полей семантического слова, описывающего эту структуру.

Такая схема позволяет хорошо отобразить блочную структуру языков программирования, обеспечивает единый способ реализации поиска имен в обычных синтаксических блоках и областях действия структур.

К сожалению, такая схема реализации очень неэффективна для случая большого числа идентификаторов в одной области действия (свойство С3) по причине того, что все хеш-таблицы имеют одинаковый размер, который не может быть достаточно большим по причине наличия большого количества областей действия (свойство С4), и, следовательно, большого числа коллизий в хеш-таблице.

Другой недостаток такого механизма связан с тем, что при поиске имени в хеш-таблице производится литеральное сравнение строк как ключей поиска в хеш-цепочке. А это также является малоэффективным в связи с большим количеством блоков, в которых необходимо произвести поиск имени (свойство С1).

Оба эти недостатка частично устраняются более изощренной реализацией. Наиболее очевидными улучшениями являются: использование хеш-таблиц различного размера для различных областей действия, использование более развитого метода разрешения коллизий, сравнение хеш-значений до сравнения литеральных строк. В процессе разработки первой версии компилятора такие

улучшения были опробованы, но в процессе анализа выяснилось, что такая изощренная реализация сильно усложняет реализацию, возможность ее понимания и внесения изменений, не устраняя полностью описанных выше недостатков.

3.3.2 Использование единой хеш-таблицы

Другим вариантом описанного выше механизма является использование единой хеш-таблицы для всех идентификаторов в лексических блоках. Аналогичные способы реализации таблицы символов рассматривается в [14, раздел 7.6], [15, глава 7]. Эта схема в некотором смысле представляет собой инверсию описанной выше схемы: вместо того, чтобы создавать единый стек для последовательного просмотра многих хеш-таблиц, создается единая хеш-таблица, а уже в ней создается множество стеков для поддержки вложенных лексических блоков.

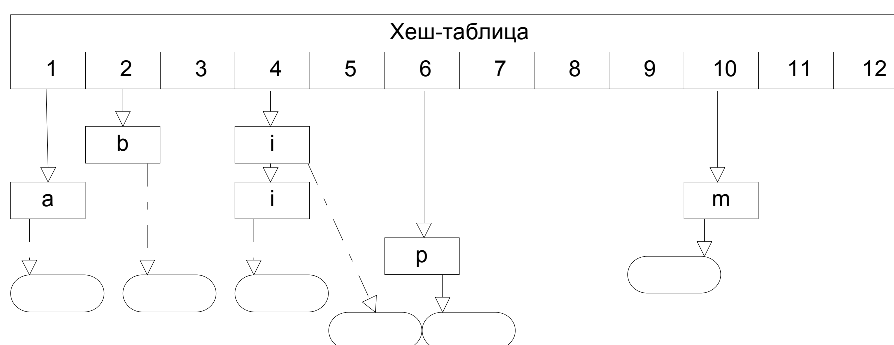


Рис. 3-2. Применение единой хеш-таблицы

При такой схеме реализации коллизии разрешаются при помощи хеш-цепочек, при этом допускается наличие в одной цепочке нескольких совпадающих идентификаторов.

Добавление в такую структуру нового имени происходит на основании его хеш-значения в начало хеш-цепочки. Таким образом, имя сущности из наиболее вложенного лексического блока оказывается ближе к началу хеш-цепочки, чем имя сущности, объявленной в объемлющей области действия. Предполагается, что при завершении разбора области действия все имена, в ней объявленные, из хеш-таблицы удаляются.

При поиске имени искомым является первая сущность в цепочке с именем, совпадающим с искомым идентификатором.

Наличие единой хеш-таблицы позволяет увеличить ее размер, использовать большое число различных хеш-значений и тем самым уменьшить число коллизий и неудачных литеральных сравнений идентификаторов.

Такая схема очень хорошо работает для случая вложенных блоков, но для реализации поиска имен в блоках структур такая схема не подходит, так как искать объявленное в структуре имя приходится уже после завершения обработки ее синтаксического блока. Таким образом, для поиска имен членов классов необходимо применять дополнительный механизм, либо усложнять правила поиска. Этот недостаток не очень существенен в языках Паскаль и Си, где лексический контекст точно определяет необходимость поиска идентификатора по общим правилам или поиска идентификатора в уже описанной структуре (в последнем случае идентификатору должна предшествовать лексема "`.`" или "`->`"), но является существенным для Си++, таким свойством не обладающим (свойство B10).

Кроме этого при такой схеме реализации довольно сложно реализовать поиск имен в глобальной области действия, что необходимо для реализации конструкции квалифицированного имени, аналога этой конструкции Си++ в языках Паскаль и Си нет (свойство B9).

3.3.3 Использование «перевернутой» таблицы

Для более полного отражения особенностей Си++ предложена следующая схема, при которой поиск идентификаторов производится “наизнанку”: в качестве основной структуры для поиска идентификаторов используется не таблица, описывающая какие идентификаторы объявлены в блоке, а таблица описывающая в каких блоках объявлен идентификатор. Таким образом, мы стремимся к минимизации числа сравнений литеральных представлений идентификаторов, но увеличиваем число сравнений указателей на структуры, описывающие блоки. Этот метод можно считать развитием предыдущего.

При обработке очередного идентификатора производится его поиск в единой на весь процесс трансляции хеш-таблице, в которой хранятся все встретившиеся до этого идентификаторы, вне зависимости от того было ли до этого дано объявление сущности с таким именем и вне связи с лексическими блоками. После этого поиска литеральное сравнение идентификаторов более не производится, все действия производятся с использованием указателя на структуру, представляющую идентификатор в хеш-таблице.

Затем поисковый алгоритм последовательно определяет, объявлен ли такой идентификатор в конкретных блоках. Для того чтобы определить, в каких блоках производить поиск идентификатора используется *дисплей*, описанный выше, но он используется только как вспомогательная структура, указывающая

лишь на лексическую вложенность блоков, и используется в основном при поиске неквалифицированных имен.

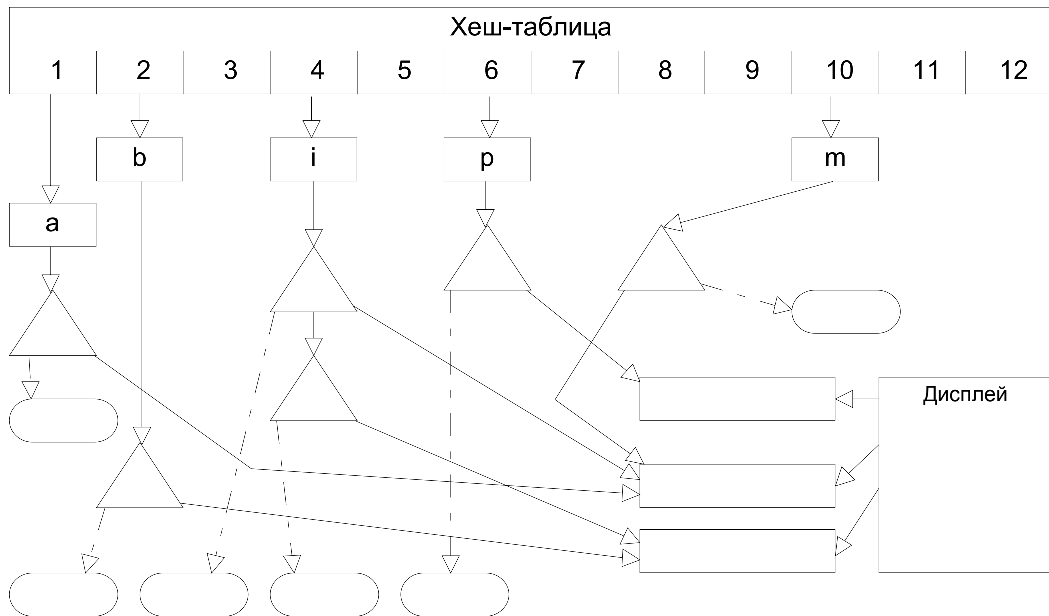


Рис. 3-3. Использование "перевернутой" таблицы

На рисунке сущности обозначены овалами, дескрипторы блоков - прямоугольниками, дескрипторы имен именованными - прямоугольниками. Треугольниками обозначены *узлы* - структуры описывающие привязку имени, дескриптора блока и сущности. Эти структуры содержат:

1. ссылку на блок block;
2. ссылку на семантическое слово object;
3. две ссылки на другие узлы left, right.

Узлы образуют направленное двоичное дерево (*дерево поиска,* [12]), ключом для этого дерева служит ссылка на блок.

Дескриптор идентификатора ссылаются на корень дерева узлов. Наличие узла означает, что идентификатор, который на него (возможно опосредованно через другие узлы) ссылается, определен в блоке block и означает сущность описанную семантическим словом object. В дереве узлов может быть несколько узлов ссылающихся на один и тот же блок, это означает, что в данном блоке описано несколько сущностей с данным именем.

Имя, определенное в области действия *A* с использованием директивы `using` (но не `using namespace`) ссылающееся на имя определенное в области действия *B* представляется узлом по общим правилам, но ссылается такой узел на фиктивное семантическое слово, созданное специально для этой директивы в

области действия *A*. Фиктивное семантическое слово ссылается на имя определенное в области *B*. Следует отдельно отметить, что ссылка на другую область действия производится именно на имя (в терминах описываемого механизма на узел), а не на именованную сущность (семантическое слово). Это объясняется правилами работы с директивами using сформулированными в разделе 7.3.3 стандарта [1], по этим правилам при поиске имени в области действия *A* через директиву `using`, а следовательно, с использованием фиктивного семантического слова, могут быть доступны сущности, определенные в области действия *B* лексически после директивы `using` (свойство B12). Если директива ссылается из области действия класса *A* на имя, определенное в базовом классе *B*, при доступе к нему действуют специальные правила проверки доступности. По этим правилам проверка доступности производится так, как будто бы вместо директивы `using` было бы дано определение члена с тем же именем. В семантическом слове созданном для директивы `using` в одном из полей хранится его доступность.

3.3.4 Алгоритмы добавления и поиска

Добавление в такую структуру новой именованной сущности (допустим, эта сущность блока *B* с именем *A,* описываемая семантическим словом *C*) производится следующим образом:

1. На основании литерального представления идентификатора *A* вычисляется его хеш-значение. По этому хеш-значению производится поиск дескриптора идентификатора в хеш-таблице. Если дескриптор не найден, (это означает, что в исходном тексте программы такой идентификатор *A* еще не встречался) дескриптор идентификатора создается.
2. Создается новый узел *N,* ссылающийся на семантическое слово *C* и блок *B,* и не ссылающийся ни на какие другие узлы.
3. Если дерево пусто узел *N* становится новым корнем дерева и дальнейшие шаги не выполняются.
4. В дереве узлов на основании ключа *B* (ссылки на блок) происходит спуск до тех пор, пока не будет обнаружено, что следующий шаг спуска невозможен по причине того, что ссылка вниз не ссылается на узел (является нулевой).
5. Из узла, в котором прекратился поиск в направлении, в котором этот поиск мог бы быть продолжен, устанавливается ссылка на узел *N*.

Пока мы не можем описать полностью процесс поиска идентификатора, но уже можно описать процедуру проверки наличия в блоке *B* сущности с именем *A*.

В случае наличия такой сущности процедура возвращает ссылку на нее, в случае ее отсутствия возвращает нулевую ссылку. Этот алгоритм приводится лишь для иллюстрации и реально в компиляторе не используется.

1. На основании литерального представления идентификатора *A* вычисляется его хеш-значение. По этому хеш-значению производится поиск дескриптора идентификатора в хеш-таблице. Если такой дескриптор не найден, возвращается пустая ссылка.
2. Иначе происходит поиск по дереву по ключу *B* первого узла, ссылающегося на *B*. Если такой узел не найден, возвращается пустая ссылка, иначе возвращается ссылка на семантическое слово из поля object этого узла.

Такой простой алгоритм поиска может быть использован в компиляторе, но это было бы неэффективно по следующим причинам:

- Достаточно часто поиск идентификатора в первом же подлежащем проверке блоке дает отрицательный результат, то есть сущность оказывается не найденной. После этого следует проверка на наличие этого идентификатора в другом блоке. При этом, так как первый шаг описанного выше алгоритма поиска не зависит от блока *B*, нет необходимости его повторять.
- Даже в том случае, когда поиск завершился успешно, то есть идентификатор был найден, часто нужно повторить поиск в том же блоке для поиска других сущностей с тем же именем, описанных в том же блоке (свойства B5, B6, B7).

Таким образом, в качестве промежуточной информации есть смысл использовать ссылку на узел дерева; в таком случае не возникает необходимости производить повторный спуск по дереву.

Два последних замечания нисколько не усложняют работу с данной таблицей, так как структура хеш-таблицы может быть скрыта минимальным числом интерфейсных функций позволяющих:

1. Находить дескриптор узла по идентификатору (1-ый шаг алгоритма поиска приведенного выше). В компиляторе эта функция выполняется на очень ранних стадиях, непосредственно после того, как лексический анализатор выделил очередной идентификатор. При этом хеш-функция для идентификаторов также используется для ускорения распознавания ключевых слов. Вся дальнейшая работа с идентификатором внутри компилятора производится только через его дескриптор. В частности, из семантического слова сущности ссылка, условно называемая «ссылкой на

идентификатор» [85], на самом деле ссылается на дескриптор идентификатора.
2. По дескриптору идентификатора *A* и блоку *B* найти ссылку на первый узел, соответствующий этой паре.
3. По ссылке на узел найти ссылку на следующий узел, соответствующей той же паре (*A*, *B*). Как видно из описанной выше структуры для поиска идентификаторов, этой функции в качестве параметра не нужно передавать *A* или *B*: информации содержащейся в узле, достаточно для поиска следующего узла.
4. По ссылке на узел вернуть ссылку на семантическое слово. Эта функция фактически возвращает поле object структуры, ссылку на которую ей передали в качестве параметра.

Использование уникальных дескрипторов идентификаторов также ускоряет работу других частей компилятора, не связанных непосредственно с поиском имен в тех случаях, когда необходимо сравнить на равенство два идентификатора. Сравнение двух указателей на дескрипторы идентификаторов существенно быстрее сравнения литеральных строк.

3.4 Свойства областей действия

Прежде чем перейти к детальному описанию алгоритмов, необходимо обсудить некоторые свойства областей действия. В языке Си++ есть несколько типов областей действия, каждый из которых обладает своими особенностями.

1. Область действия *namespace*.
2. Область действия класса (*class*).
3. Область действия функции (*function*).
4. Область действия локального блока (*local block*).
5. Область действия прототипа функции (*function prototype*).

Существуют ограничения на вложение областей действия различных типов. Области действия могут вкладываться друг в друга, как показано на рисунке.

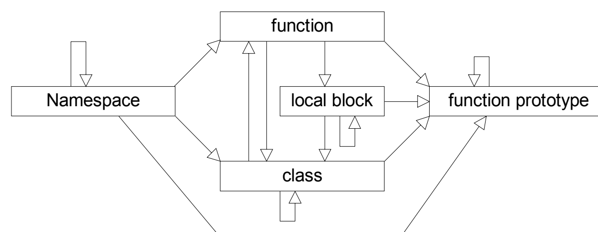


Рис. 3-4. Допустимые вложения областей действия

По определению языка глобальная область действия является областью типа namespace.

Нас будут интересовать следующие свойства областей действия:

1. *Лексическая замкнутость***.** Все типы практически все областей действия (кроме глобальной области действия и локальных областей действия циклов и условных операций) начинаются с лексем открывающей скобки ("{" или "(") и заканчиваются парными закрывающими скобками. Все типы областей действия, кроме области namespace обладают тем свойством, что, будучи однажды закрыты закрывающей скобкой, дальше по тексту программы в эти области действия новые имена не попадают. Области действия namespace и только они этим свойством не обладает (свойство B12).
2. *Время жизни***.** Некоторые области действия обладают тем свойством, что после их закрытия у пользователя нет возможности сослаться на имя из этой области. Этим свойством обладают области действия функции, локального блока и прототипа функции. В то же время, на имена, объявленные в области действия класса или namespace, можно ссылаться из других областей действия. Например:

```
struct A {
      static int a;
};
void f () { int i = A::a; }
// в функции f мы ссылаемся к имени объявленном в
// области действия класса.
void g() {
            // из функции g нет возможности сослаться на имя
            // переменной i объявленной в области действия
            // другой функции.
}
```

Так как область действия namespace может быть вложена только в другую область действия того же типа, на любое имя, объявленное в namespace, можно сослаться из любой точки программы, лексически следующей за объявлением имени. Аналогичным свойством обладают и области действия классов, вложенных в области действия namespace. Области действия классов объявленных в локальных блоках таким свойством не обладают, так как нет возможности сослаться на имя самого класса.

Заметим также, что все области действия, кроме локального блока, "принадлежат" некоторым сущностям, и эти сущности сами могут иметь имя, доступное в другой области действия. Свойство "времени жизни" никак не связанно с наличием у сущности имени: время жизни области действия

именованной функции локально, в то время как время жизни области действия неименованного класса нелокально. Например:

```
void f ()
{
  struct { // неименованный класс
    int i;
  } x;
  x.i = 1; // обращение к имени принадлежащему области
           // действия класса вне этого класса
}
```

Таким образом, мы готовы построить формальную модель для определения действия имен. Допустим X - множество областей действия. $>(X,X)$ - отношение "быть вложенным в", определенное на областях действия; $\Rightarrow(X,X)$ - транзитивное замыкание отношения $>$ и $\rightarrow(X,X)$ - отношение лексического следования областей действия (для прямо или косвенно вложенных областей отношение $\rightarrow$ считаем неопределенным). K - глобальная область действия из X. Тогда граф $G(X,>)$ является деревом, и K является его корнем.

Для любой вершины x по определению $K \Rightarrow x$.

Отношение $x \rightarrow y$ определено тогда и только тогда, когда в графе G нет путей из вершины x и в вершину y, то есть $\neg\, x \Rightarrow y \;\&\; \neg\, y \Rightarrow x$ (таким образом, для любой $x \in X$ отношения $K \rightarrow x$ и $x \rightarrow K$ не определены).

Отношения $\rightarrow$ и $\Rightarrow$ являются транзитивными антирефлексивными, следовательно, отношениями частичного порядка.

Каждый элемент из X имеет свой тип: один из пяти типов областей действия. Введем новое обозначение: $x \Rightarrow^* y$, если $x \Rightarrow y$ и в графе $G(X,>)$ в пути по вершинам из вершины x в вершину y встречается только вершины, имеющие типы "namespace" или "class". Это отношение вводится для формального описания оператора доступа к контексту `::`, с помощью которого можно обращаться только к именованным сущностям, определенным в классах и namespace.

Допустим, A и B - две произвольные вершины из X. Имена из области действия A потенциально достижимы из области B, если $A \Rightarrow B \vee (A \rightarrow B \;\&\; K \Rightarrow^* A)$. Для отношения потенциальной достижимости введем обозначение $R(A,B)$.

Введенное понятие потенциальной достижимости не обязательно означает, что в области действия B может существовать синтаксическая конструкция, ссылающаяся на имя, объявленное в области A. Например:

```
void f(int x)
{
        {
                int x;
        }
```

}

Пусть A - область действия функции, а B - область вложенного действия блока. В этом случае имена области действия A потенциально достижимы из области B, но нет никакой синтаксической конструкции языка, позволяющей сослаться на имя x из области A.

Отрицание отношения потенциальной достижимости имеет более сильный смысл. Как будет показано ниже, если имена из области A не являются потенциально достижимыми в области B, то при разборе и поиске имен в области B, всех последующих и вложенных в нее областей имена из области A остаются недостижимыми и, следовательно, нет необходимости в их дальнейшем хранении структурах данных компилятора, используемых для поиска имен.

Утверждение 1. Если $A \to B$ & $B \to C$ & $\neg R(A,B)$, то $\neg R(A,C)$.

$A \to B$ & $B \to C$ влечет $A \to C$ ($\to$ является отношением частичного порядка)

$\neg R(A,B)$ влечет $\neg K \Rightarrow^* A$, иначе, пользуясь определением R, прейдем к противоречию).

Таким образом, вторая альтернатива в определении $R(A,C)$ невозможна.

Тогда допустим $A \Rightarrow C$, но это противоречит определенности $A \to C$.

Утверждение 2. Если $A \to B$ & $B \Rightarrow C$ & $\neg R(A,B)$, то $\neg R(A,C)$.

Допустим $A \Rightarrow C$. По условию утверждения $B \Rightarrow C$, значит либо $A \Rightarrow B$, либо $B \Rightarrow A$, что противоречит $A \to B$.

$A \to B$ & $B \Rightarrow C$ влечет, что отношение $A \to C$ определено и верно, из чего следует невозможность $A \Rightarrow C$.

Воспользовавшись $\neg K \Rightarrow^* A$ опровергается вторая альтернатива для $R(A,C)$.

Утверждение 3. Если $K \Rightarrow^* A$ & $A \to B$, то $R(A,B)$.

Прямо следует из определения $R (A,B)$.

Утверждение 4. Если $A \Rightarrow B$ & $A \to C$ & $\neg R(A,C)$, то $\neg R(B,C)$.

Из того, что $A \to C$ и $A \Rightarrow B$ следует, что $B \to C$, а следовательно, невозможно $B \Rightarrow C$.

Так как $\neg R(A,C)$ и $A \to C$ то невозможно $K \Rightarrow^* A$, а следовательно, невозможно $K \Rightarrow^* B$, так как путь из корня K в вершину B проходит через вершину A.

Приведенные утверждения дают исчерпывающий ответ на вопрос о том, в какой момент информация об именах в блоке *A* становится несущественной. Если *A* - локальный блок, блок функции или блок прототипа функции, то, как только $A \rightarrow B$ (а это все блоки, лексически следующие за блоком *A*) все имена блока *A* перестают быть потенциально доступными.

Если *A* - блок класса, то его имена могут оставаться потенциально доступными до конца трансляции (утверждение 3), если этот класс не вложен в локальный блок. Если же класс вложен в локальный блок, то имена становятся недоступными как только становятся недоступными имена из содержащего его локального блока (утверждение 4), а следовательно, как только становится недоступным имя этого класса.

3.5 Поиск неквалифицированных и квалифицированных имен

Теперь мы готовы обсуждать собственно поиск имен в различных областях действия. Основной структурой данных будет граф *N(X,I)*. Отношение *I* является объединением отношения наследования для классов и отношения, индуцируемого конструкцией `using namespace`. Последняя конструкция делает все имена, определенные в одной области действия типа namespace также доступными в другой области действия. Действие этой конструкции распространяется только на часть программы, лексически за ней следующей, то есть в процессе трансляции граф *N* может изменяться, в нем могут появляться новые ребра, соединяющие две уже существующие вершины. Также эта конструкция может создавать в графе *N* циклы.

По определению отношения инцидентности в графе *N* все вершины, не соответствующие областям действий классов и namespace являются изолированными. Пользуясь этим графом, можно описать основной алгоритм поиска имен – поиск неквалифицированного имени, а затем остальные алгоритмы, относящиеся к поиску имен.

3.5.1 Поиск неквалифицированного имени

Допустим, в некоторый момент в процессе разбора встретилось неквалифицированное имя *x*. В этот момент в дисплее находились ссылки на области действия $d_1, d_2, \ldots d_n = K$. Поиск производится последовательно по всем областям действия $d_1, d_2, \ldots d_n$ и завершается, если имя найдено в очередной

области действия. Поиск завершается неудачно, если имя не найдено ни в одной из областей действия.

Для каждой области d_i действия осуществляется поиск с использованием обхода в глубину без повторений графа *N*, начинающийся из вершины, соответствующей d_i. Тип искомой сущности определяется контекстом, в котором встретилось имя, например, имя *x* может быть именем метки или следовать за ключевым словом `class`, `struct`, `enum`, `union`, `namespace`, в таком случае искомая сущность должна быть соответственно меткой, именем типа или namespace.

Эффективный метод поиска определения имени в одной конкретной области действия, основанный на использовании хеш-таблицы, описан выше.

В процессе поиска может обнаружиться, что из вершины d_i достижимо несколько различных именованных сущностей с именем *x* и подходящего типа. В этом случае выполняется проверка на доминирование для имен членов класса или проверка того, что все using-имена на самом деле ссылаются на одну сущность. Если проверка не может выявить единственной доминирующей сущности, генерируется сообщение об ошибке неоднозначности использования имени.

В дисплее может находиться несколько областей действия типа namespace, в которых до этого могли использоваться конструкции using namespace, ссылающиеся на совпадающие namespace, то есть для различных d_i и d_j может существовать непустое множество вершин $S \subset X$, достижимых из обеих вершин *d*. Очевидно, что если для первой из них в множестве *S* не найдено подходящего определения имени, то для второго также подходящее имя не будет найдено. А значит, метки, присвоенные вершинам графа при их посещении при обходе в глубину, можно сохранять в графе *N* во время всего процесса поиска, и не сбрасывать при каждом переходе к новому значению d_i.

Имена меток, определенных в функции, даже если они определены в локальных блоках, считаются определенными в области действия имен самой функции, и в процессе поиска меток можно пропустить все элементы дисплея, соответствующие локальным блокам.

3.5.2 Поиск квалифицированного имени

Поиск квалифицированного имени, начинающегося с операции доступа к глобальной области действия, ведется без использования дисплея, а непосредственно начинается с глобальной области действия $d=K$.

При поиске квалифицированного имени `A1::A2::..::Am` поиск также производится без использования дисплея, а выполняется последовательно для имени A_i в области действия, соответствующей результату поиска A_{i-1}. Поиск начинается с поиска имени A_1, который производится по общим правилам, то есть может быть поиском неквалифицированного имени или поиском квалифицированного имени с доступом к глобальной области действия, в зависимости от того, использовалась ли конструкция доступа к глобальной области действия. Определение того, имеет ли именованная сущность подходящий тип, выполняется только для результата самого последнего поиска. При выполнении промежуточных шагов поиска всегда ищутся сущности типа class или namespace.

В случае поиска имени члена класса после оператора доступа правила поиска отличаются от всех предыдущих случаев. Правила поиска имен для этого случая описаны в разделе 3.4.5 стандарта [1]. В этом случае может дважды использоваться алгоритм поиска квалифицированного имени: один раз по общим правилам и второй раз, начиная с вершины графа *N*, соответствующей классу, стоящему в левой части операции доступа.

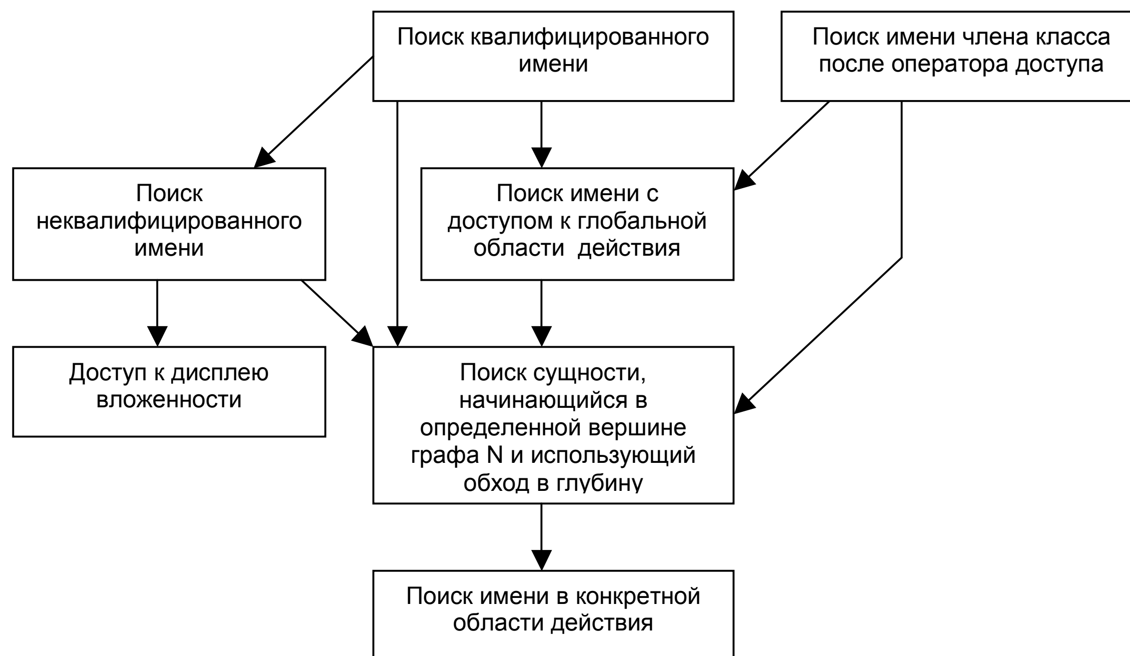


Рис. 3-5. Взаимодействие различных частей механизма поиска имен.

На рисунке изображено взаимодействие различных алгоритмов поиска имен. На нем отсутствуют части, относящиеся к проверке типа сущности, диагностике ошибок, проверке однозначности и сбросу флагов выставленных в процессе обхода графа *N*.

3.6 Заключение. Выводы главы 3

В этой главе были рассмотрены особенности поиска имен в языке Си++, описаны недостатки использования в компиляторе этого языка традиционных методов реализации поиска имен и предложен более эффективный способ реализации, который, также может использоваться и в компиляторах других языков программирования.

Основные результаты третьей главы могут быть сформулированы следующим образом:

1. Исследованы особенности поиска имен в языке Си++ и отличия правил поиска имен в этом языке от правил поиска имен в других языках программирования.
2. Предложен эффективный способ проверки наличия определения имени в различных областях действия с использованием хеш-таблицы.
3. Описана реализация механизма поиска имен, эффективно реализующая правила языка, использующая граф областей действия и дисплей областей действия.
4. Изучены условия, при которых объявленные в области действия имена становятся недоступными и могут быть удалены из хеш-таблицы.
5. Предложен реализация конструкций `using` и `using namespace`.
6. Описано взаимодействие частей компилятора, реализующих различные варианты поиска имен.
7. Описано взаимодействие механизма поиска имен с другими частями компилятора.

Глава 4. Реализация системы типов и проверки типов

4.1 Введение

В большинстве современных языков программирования имеется понятие типов данных. Большая часть таких языков программирования предполагают проверку типов данных на этапе компиляции программы (статически типизированные языки) [88]. Детали определения типов и их проверки в выражениях отличаются в различных языках программирования, и в том же время содержат сходные общие черты, такие как:

- наличие конечного числа базовых (или фундаментальных) типов;
- наличие различных способов порождения новых типов, в том числе порождения новых типов от других определенных пользователем (пользовательские типы);
- при объявлении объектов программы им приписывается определенный тип;
- операторы в выражениях требуют наличия операндов типов удовлетворяющих определенным языком требованиям;
- определяется способ приписыванию выражению определенного типа на основании типов приписанных его подвыражениям;
- в некоторых случаях результаты подвыражений неявно преобразовываются к другим типам.

Язык Си++ является языком со статической проверкой типов. Часть правил проверки типов, связанная с проверкой типов для простейших арифметических выражений является стандартной, свойственной многим языкам программирования. Правила проверки типов для более сложных выражений, особенно выражений содержащих операции, работающие с объектами классов, гораздо сложнее, и плохо поддаются формальному описанию с помощью стандартной приемов, описанных, например в [14, глава 6].,

В этой главе обсуждаются особенности типов в Си++, основные требования к реализации системы типов в компиляторе, возможные варианты реализации и реализация одного из вариантов. Также излагаются основные идеи алгоритма проверки типов, учитывающего особенности языка.

4.2 Общие требования к реализации системы типов

Язык Си++ предоставляет достаточно большой набор стандартных *фундаментальных* типов и гибкие возможности создания новых типов на основе уже определенных.

Набор фундаментальных типов включает в себя символьные типы набор целочисленных типов различного размера, булевский тип, три типа чисел с плавающей точкой различной точности и пустой тип `void`.

К способам образования новых типов относятся:

- определение перечислимого типа;
- создание объединений, структур и классов;
- несколько способов «легковесной» модификации существующих типов, таких как определение указателей и ссылок на тип, определение массива объектов данного типа, добавление квалификаторов константности (`const`) и «подвижности» (`volatile`), образование типов функций и указателей на члены класса;
- кроме перечисленных выше способов существует возможность задания шаблонов классов и функций. Шаблоны напрямую не участвуют в системе типов объектов Си++ и не учитываются в процедурах проверки типов, а участвуют лишь конкретные экземпляры классов и функций, сгенерированных по шаблону.

Строить типы можно применяя произвольное число модификаций описанными выше способами, но семантически допустимыми являются не все комбинации способов построения типов. Так, например нельзя создавать массивы ссылочных типов.

Добавление квалификаторов константности и подвижности может применяться к типу более одного раза, но любое повторное применение такой модификации не меняет семантики этого типа, существенным является лишь качественный результат: получен ли данный тип из другого применением хотя бы одной операции квалификации.

Для программ на языках Си++ и Си характерно активное использование «легковесных» модификаций типов. Легковесные модификации типов могут быть многоуровневыми. Многие стандартные операторы языка определены не только для фундаментальных типов, но и для типов, определенных пользователем (примером может служить хорошо известная адресная арифметика языка Си).

При разработке реализации системы типов приходится учитывать тот факт, что для проверки типов часто приходится установить каким способом и из какого

типа был получен данный тип, является ли данный тип терминальным, к какой группе типов данный тип относится.

Под группой типов подразумеваются все типы обладающие некоторым свойством, один и тот же тип может принадлежать нескольким группам. Ниже перечислены наиболее важные с точки зрения реализации группы:

1. указательные и массивные типы
2. фундаментальные типы
3. арифметические типы
4. перечислимые типы
5. классовые типы
6. целочисленные типы
7. типы чисел с плавающей точкой
8. скалярные типы
9. функциональные типы
10. константные и подвижные типы
11. ссылочные типы
12. указатели на члены классов

С точки зрения реализации необходимо иметь возможность быстро определять принадлежность типа произвольной группе.

Во многих случаях необходимо иметь возможность определить тип, на основании которого при помощи добавления константности, подвижности и ссылок был создан данный, так как семантика многих операций не зависит от наличия квалификаторов типа.

Часто по типу необходимо определить размер объектов данного типа.

Еще одной операцией над типами, которая часто используется при проверке типов в выражениях, является проверка типов на совпадение.

Проверка возможности преобразования значения одного типа к значению другого типа является существенной с точки зрения реализации компилятора. Следует отдельно отметить, что такая проверка не может быть выполнена на основании только информации о типах. В некоторых случаях необходимо иметь информацию о том является ли преобразуемое значение lvalue или rvalue. (Например lvalue типа `int` может быть преобразовано к типу «ссылка на `int`», но rvalue такого типа не может быть преобразовано к тому же типу).

Для генерации отладочной информации в наиболее популярных современных форматах, таких как STABS [66] и DWARF [64] желательно иметь возможность занумеровать все использованные в единице трансляции типы. При этом допускается присваивать совпадающим типам различные номера, но такая

нумерация не выглядит естественной и может существенно увеличить объем генерируемой отладочной информации.

4.3 Методы реализации типов

В этом разделе рассматриваются способы реализации типов, использованные в первой и второй версии компилятора. Более подробно описывается метод использованный во второй версии, так как он является более эффективным.

4.3.1 Реализация типов с помощью типовых цепочек

В первой версии компилятора использовалась реализация типов при помощи «типовых цепочек» аналогичных цепочкам, описанным в [14, глава 6]. Каждый тип представляется в виде цепочки (реализуемой как массив) кодов типов. Тип `const int *` таким способом представляется с помощью следующей цепочки: *(код *, код const, код int)*. Массивные типы представляются цепочкой *(код массива, число элементов массива, цепочка типа элемента массива)*. Классовые и перечислимые типы представляются в цепочке парой *(код типа, указатель на семантическое слово типа)*. Тип указателя на члены классов представляется цепочкой *(код указателя на член класса, указатель на семантическое слово класса, цепочка типа именуемого данным указателем)*. Самое сложное представление имеют функциональные типы. Для их представления в виде цепочки создается «фиктивное» семантическое слово: *(код функционального типа, указатель на семантическое слово фальшивой функции)*. При этом тип, возвращаемый функцией и типы ее параметров доступны только через семантическое слово фиктивной функции, само же семантическое слово создавалось с единственной целью – хранить информацию о типе возвращаемого значения и типах параметров.

При такой схеме для того, чтобы определить принадлежность типа к определенной группе необходимо просмотреть начало его цепочки (для различных групп типов необходимо пройти различное число элементов цепочки).

Для сравнения типов на совпадение приходится проходить по всей цепочке, последовательно сравнивая коды типов. Функциональные типы сравниваются с помощью последовательного сравнения числа и типов параметров, а также типа возвращаемого значения.

При таком методе реализации для создания нового типа на основании имеющегося не допускается добавлять новые коды в начало цепочки и отрезать первые коды цепочки, так как сами цепочки создаются в динамической памяти. Во

всех случаях, когда это необходимо, создается копия цепочки или ее части. Кроме этого не разрешается повторное использование уже существующих цепочек. Так, если объявлено две переменные типа указатель на `const int` семантические слова этих переменных ссылаются на две различные цепочки. Для проверки типов в выражениях, использующих эти переменные, создаются копии их типовых цепочек.

Такой подход может показаться весьма расточительным с точки зрения использования компилятором памяти. Тем не менее это не вызывает особых проблем. Дело в том, что в процессе проверки типов в выражениях типовые цепочки, которые создаются для хранения типов подвыражений, используются лишь как временные объекты, и после проверки выражения освобождаются. Следить за тем, что во всех возможных выражениях все возникшие цепочки освобождаются довольно сложно, и велика вероятность того, что в результате неаккуратной реализации в некоторых случаях теряются ссылки на созданные цепочки.

Как отмечалось выше, цепочки типов представляющие тип подвыражения освобождаются. Поэтому после проверки типов в выражении невозможно определить какой тип имело то или иное подвыражение. Для компилятора, как такового, это не является критичным – вся необходимая для генерации кода информация остается в узлах дерева. Но в процессе развития компилятора и использования его частей не только как составной части компилятора, но и в качестве составной части других систем, в частности как составной части виртуальной машины, становится существенным наличие информации о типе подвыражения в каждом узле дерева выражения.

Существенным недостатком такого метода реализации оказались также большие расходы на копирование типовых цепочек. При проверке типов в выражении, при обработке узла переменной компилятор вынужден копировать тип этой переменной и передавать выше именно копию типа, так как по общему механизму эта типовая цепочка должна быть, в конечном счете, освобождена, но мы не имеем права освобождать типовую цепочку самой переменной, так как переменная может быть использована позднее в другом выражении. Этот пример нагляднее всего демонстрирует возникающие проблемы эффективности: компилятор выполняет слишком большое число копирований цепочек типов.

Другая проблема, для которой не было найдено приемлемого решения в рамках такой реализации типов это нумерация типов при генерации отладочной информации. Присвоив типу некоторый номер хотелось бы во всех случаях, когда потребуется определить номер того же типа вернуть уже присвоенный номер. Увы,

установить по типовой цепочке был ли ей ранее присвоен номер невозможно без дополнительных структур.

4.3.2 Реализация системы типов при помощи таблицы

Ниже будут рассмотрены различные варианты реализации типов, частично свободные от перечисленных выше недостатков реализации на основе цепочек. Все варианты так или иначе предлагают способ обеспечить «уникальность» каждого типа. Каждый тип представляется в виде некоторой структуры, при этом все эквивалентные типы ссылаются на одну структуру.

Начнем с описания всех допустимых в языке типов. Приведенное ниже описание полностью повторяет содержание раздела 3.9 стандарта [1], но более сжато и формализовано. Также ниже приводятся семантические ограничения на построения новых типов, описанные в других разделах стандарта.

Зададимся следующей системой *терминальных* типов: фундаментальные типы (void, char, unsigned char, signed char, signed short int, unsigned short int, signed int, unsigned int, signed long int, unsigned long int, float, double, long double, bool, wchar_t), пользовательские и встроенные классовые типы, пользовательские перечислимые типы.

При таком определении все возможные типы можно построить с помощью следующей системы правил:

1. Терминальный тип есть тип.

2. Добавление квалификаторов.

Если T - тип, то const T, volatile T, const volatile T есть типы. Такие типы будем называть *квалифицированными вариантами* типа T, а const и volatile квалификаторами. Если тип T получен с помощью произвольного правила, кроме правила добавления квалификаторов, то его будем называть неквалифицированным, иначе квалифицированным.

3. Создание указателя.

Если T - тип, то *T (указатель на T) есть тип. Тип, созданный по этому правилу, будем называть *указательным*.

4. Создание ссылки.

Если T - тип, то &T (ссылка на T) есть тип. Тип, созданный по этому правилу, будем называть *ссылочным*.

5. Создание массива.

Если T - тип n - натуральное число, то T[n] (массив из n элементов типа T) есть тип. Тип, созданный по этому правилу, будем называть *массивным* типом.

6. Создание функции.

Если T_0, T_1, …, T_k (k≥0) типы, c, v и e - булевы значения, L - некоторое имя из множества, содержащего по крайней мере два имени ("C" и "C++"), то T_0 $(T_1,…T_k,[…]_e)_L$ $const_c$ $volatile_v$ есть тип. (Функция принимающая k параметров типов $T_1…T_k$, возвращающая значение типа T_0, с переменным числом параметров если e - истина, с соглашениями о вызовах языка L. c и v имеют смысл только для функций-членов классов и определяют возможность вызова функции только для константных или подвижных объектов). Типы построенные по этому правилу будем называть *функциональными*.

7. Создание указателя на член класса.

Если T тип, C - класс, то T C::* (указатель на член класса C типа T) есть тип. Такие типы будем называть *указателями на члены классов*. Заметим, что в этом определении C является классом, а не классовым типом.

При записи традиционно используются скобки для явного указания последовательности применения правил образования типов. Следуя правилам Си++ будем опускать скобки в тех случаях, когда тип построен применением правила 5 к типу, полученному по правилу 3.

Тип, построенный по перечисленным выше правилам, может оказаться семантически некорректным. Ниже перечислены семантические ограничения на возможность применения различных правил:

2. Правило 2 не должно применяться к типам, полученным по правилам 2, 4, 5, 6.
3. Правило 3 не должно применяться к типам, полученным по правилу 4.
4. Правило 4 не должно применяться к терминальному типу void, и типам, полученным по правилам 4, 6.
5. Правило 5 не должно применяться к терминальному типу void, и типам, полученным по правилам 4, 6.
6. Правило 6 не должно применяться к типам, полученным по правилам 2, 5, 6. Кроме этого типы $T_1..T_k$ не могут быть терминальным типом void.
7. Правило 7 не должно применяться к терминальному типу void, и типам, полученным по правилу 4.

Ограничения на применение правила 5 могут показаться излишне жесткими. Тем не менее, они точно соответствуют семантике языка, определенной в стандарте. Так, при объявлении функции допускается указывать void в списке

параметров, но такое объявление семантически эквивалентно объявлению с пустым списком параметров (k=0).

Также в объявлениях функций допускается использование массивных типов в качестве типов параметров и типа возвращаемого значения, но такие объявления семантически эквивалентны объявлениям, в которых все последние использования правила 5 для типов $T_0 .. T_k$ заменены на применения правила 3. Например:

`void (int [10])` эквивалентно `void (int *)`.

При объявлении функции также можно использовать функциональные типы в качестве типов параметров и возвращаемого значения, но такие объявления семантически эквивалентны объявлениям, полученным применением правила 3 к функциональным типам. Например:

```
typedef int T (int);
```

`T f(T)` эквивалентно `T *f(T*)`;

Использование квалифицированного типа в качестве типа возвращаемого значения допускается, но оно эквивалентно использованию неквалифицированной версии того же типа. Квалификации для типов параметров нельзя полностью опускать только в случае полного определения функции, такие квалификации играют роль только при проверке типов внутри самой функции, но не сказываются при работе с ней как с объектом функционального типа. Например:

```
void f (const int i)
{
        i = 1;  // ошибка, вызванная квалифицированностью типа параметра
}
void (*p)(int) = f;  //  квалификация параметра не учитывается
```

Таким образом, квалификаторы следует учитывать только в типе параметра функции как самостоятельного объекта, но не как часть типа функции.

Такие атрибуты функции, как имена ее параметров, значения параметров по умолчанию и throw-спецификации не являются частью функционального типа.

Семантически корректные типы T_1 и T_2 будем называть эквивалентными, если они построены применением одинаковых правил образования типов к одинаковым терминальным типам. Потребовав корректность от типов мы освобождаем себя от необходимости определять эквивалентность типов полученных двумя последовательными применениями правила 2.

Если рассмотреть все возможные способы построения типов, отбросив возможность построения функционального типа, и изображать типы как вершины графа, а построение нового типа с помощью некоторого правила как ребро графа (ориентированного от производного типа к базовому), то получится направленный

лес, ориентированный к корню. Корнями леса будут служить определенные нами выше терминальные типы.

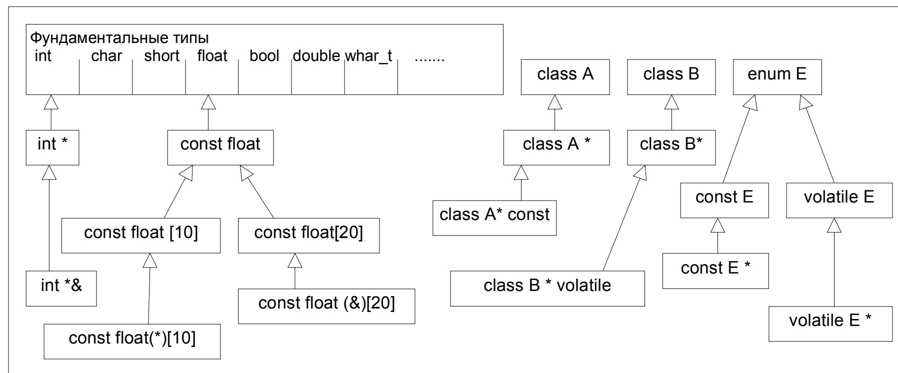


Рис. 4-1. Типы как лес, ориентированный к корню.

Функциональный тип строится на основании нескольких типов (тип возвращаемого значения и типы параметров). Если следовать приведенным выше соглашениям о представлении типов как вершин графа, то граф перестает быть лесом: из вершины функционального типа исходит несколько ребер. Граф при этом остается ациклическим направленным.

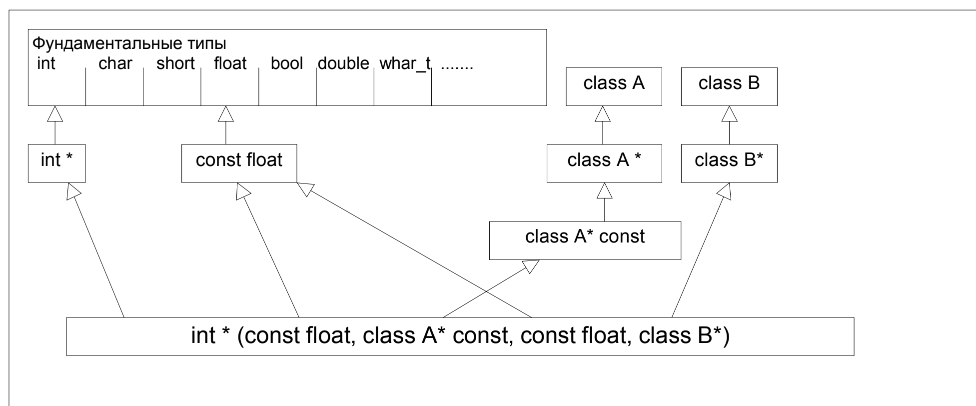


Рис. 4-2. Функциональные типы, как DAG.

Следует дать пояснение по поводу выбора системы терминальных типов: фундаментальные и перечислимые типы не строятся на основе других типов, поэтому их «терминальность» выглядит вполне естественной. Классовые типы, за исключением вырожденного случая пустого класса, строятся на основании других типов - типов членов класса. При этом член класса может иметь тип, который основан на типе самого класса. Простейший пример, приведенный ниже, иллюстрирует такую возможность. Подобные классы часто используются в программах на Си++ для реализации динамических структур данных, например списков:

```
struct List {
      struct List *next;
};
```

В приведенном выше примере тип члена класса ссылается на сам класс, то есть если не считать класс терминальным типом и попытаться изобразить такой тип в виде графа, то этот граф будет содержать цикл. К счастью все правила проверки типов не принимают во внимание типы полей классовых типов, а эквивалентность классовых типов в языке Си++, как и во многих других языках, именная, а не структурная.

4.3.3 Реализация типов с помощью структур

Элементами структуры, представляющей тип являются:

- тег типа (для быстрой классификации типов)
- размер типа
- хеш-значение или указатель на справочник производных типов
- информация, необходимая для генерации кода
- информация, необходимая для генерации отладочной информации
- базовый тип или базовые типы (если тип построен на основе других) и другая информация, зависящая от способа образования типа (такая как размер массива)

Сравнение типов на эквивалентность при такой реализации возможна простым сравнением указателей на структуры, описывающие тип, при условии, что для каждого неэквивалентного типа существует единственная структура.

Наиболее часто встречающаяся операция над типами – определение типа, на основе которого построен данный, выполняется заглядыванием в соответствующее поле структуры. Класс типа определяется по его тегу.

Следующие вопросы, на который необходимо ответить: каким образом будет обеспечиваться уникальность структур типов и как осуществляется построение производного типа. Эти вопросы тесно связаны друг с другом. При поиске производного типа сначала определяется, существует ли уже в таблице типов структура для данного типа. Если она существует, то возвращается указатель на нее; если нет, то соответствующая структура создается.

Уникальность терминальных типов легко обеспечивается созданием структур фундаментальных типов на этапе инициализации компилятора (до начала обработки пользовательской программы) или созданием структур по ходу обработки пользовательской программы, но с сохранением указателей них. (Ввсего в Си++ около двух десятков встроенных и фундаментальных типов).

Указатель на структуру в таблице типов относящуюся к классовому и перечислимому типу удобно хранить в семантическом слове соответствующей сущности (класса или перечисления).

Предложено два способа быстрого поиска производного типа. Оба способа основаны на рекурсивных правилах построения производных типов.

Первый способ основан на использовании хеш-функции, которая каким либо образом задана на терминальных типах и рекурсивно строится для производных. Например:

```
H(T*) = H(T)*5+21;
H(T&) = H(T)*7+37;
H(const T) = H(T)*11+53;
H(T[n]) = H(T)*(10+n*3)+71;
….
H(T0(T1, T2,…)) = H(T0)+3*H(T1)+5*H(T2)+…
```

При таком способе зная хеш-значение типа (которое хранятся в структуре, описывающей тип) по приведенной выше формуле вычисляется хеш-значение производного типа и на основании этого хеш-значения в таблице ищется производный тип. Колизии в такой хеш-таблице разрешаются методом цепоцек ([13], раздел 6.4).

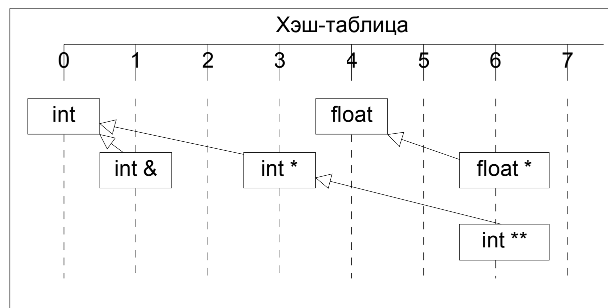


Рис. 4-3. Таблица типов, как хеш-таблица.

Второй способ заключается в том, что с каждой структурой, представляющей тип, ассоциируется набор указателей на типы, производные от данного. При этом следует иметь ввиду, что для большинства типов Си++ на их основе можно построить бесконечно много корректных производных типов. Поэтому в таком «атласе» производных типов имеет смысл разделять типы, которые получены из данного при помощи различных правил (по некоторым правилам, таким как образование указателя на тип возможно создание только одного с точностью до эквивалентности нового типа).

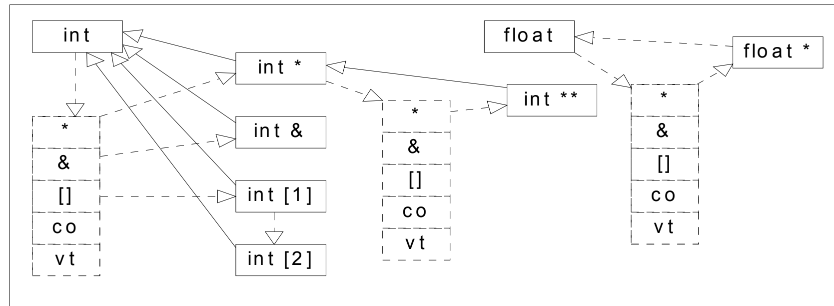


Рис. 4-4. Таблица типов как таблица с атласом.

4.4.4 Квалификаторы `const` и `volatile`

Как отмечалось выше, построение нового типа добавлением к существующему квалификаторов `const` и `volatile` обладает некоторыми особенностями: а именно, если тип `T`' получен из `T` путем добавления одного из квалификаторов и на основе типа `T`' попытаться построить новый тип путем добавления того же квалификатора, то результатом будет тип, эквивалентный `T`'. Другой особенностью построения типов путем добавления квалификаторов является их «коммутативность» в том смысле, что если из типа `T` получен тип `T`' путем последовательного добавления квалификаторов, и получен тип `T`'' путем последовательного добавления тех же квалификаторов, но в другом порядке, то типы `T`' и `T`'' будут эквивалентны.

Еще одной особенностью этих квалификаторов является то, что во многих частях компилятора, связанных с проверкой типов квалификаторы верхнего уровня просто не учитываются, точнее семантика конструкций не зависит от того, получен на вход тип `T` или тип `T`' полученный из `T` путем добавления квалификаторов. В таких случаях процедуры проверки типов вынуждены выполнять дополнительные операции: пропускать квалификаторы верхнего уровня.

Этими особенностями квалификаторов можно воспользоваться при реализации типов. А именно, предлагается не заводить в таблице отдельных структур типов для описания квалифицированных типов, а воспользоваться тем, что на инструментальных платформах структуры описания типов выравниваются в памяти (если появится инструментальная платформа, для которой это не выполняется выравнивание можно производить искусственно) по границе 4 байт. В то же время указатели на эти структуры могут указывать и на невыровненный участок памяти. Тогда, если структура неквалифицированного типа `T` в таблице типов находится по адресу `P`, то младшие два бита в `P` опущены. В одном из битов можно разместить информацию о том, что данный тип был квалифицирован с помощью квалификатора `const`, в другом - `volatile`. Естественно, такие детали

реализации системы типов должны быть скрыты от пользователей реализации системы типов с помощью набора интерфейсных функций, работающих с типами.

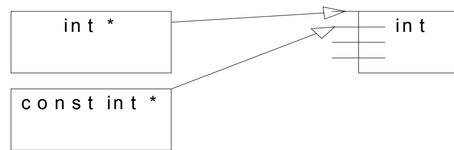


Рис. 4-5. Облегченный вариант реализации квалификаторов

4.4 Дополнительные возможности по работе с типами

Описанные выше методы реализации системы типов ориентированы скорее на улучшение функциональности компилятора, но не на расширение его функциональности. Ниже будут рассмотрены вопросы, связанные именно с дополнительной функциональностью, которую можно реализовать немного изменив реализацию.

После обеспечения уникальности структур типов и вынесение их в дополнительную таблицу, которая существует достаточно независимо от дерева программы и таблиц, без дополнительных усилий мы получаем возможность в каждом узле выражения хранить информацию о типе.

Согласно семантике Си++ `typedef` объявление не вводит новый тип, а лишь именует некоторый уже построенный тип. В целях упрощения компилятора как такового такое определение несколько упрощает реализацию семантики таких объявлений. Но для анализатора Си++ программ может оказаться существенным иметь возможность проверки каким образом некоторый объект получил данный тип: явно или с использованием `typedef`. Например:

```
typedef int *T;
int **p1;
T *p2;
```

С точки зрения компилятора и в смысле данных выше определений типы переменных `p1` и `p2` эквивалентны. Однако анализатору программ желательно знать, что переменная `p2` была объявлена с использованием typedef-имени. Решением данной задачи может быть введение дополнительного типа структур в таблице типов, соответствующего typedef имени. Естественно после такого расширения таблицы нарушается уникальность типов и процедура проверки эквивалентности типов не может быть сведена к простому сравнению указателей.

В объявлениях функций и функциональных типов разрешается именовать параметры, но их имена никак не влияют на семантику программы. В то же время

эквивалентные объявления функциональных типов могут иметь различные имена параметров. Например:

```
int (*p1)(int left, int right);
int (*p2)(int first, int second);
```

С точки зрения компилятора типы переменных `p1` и `p2` эквивалентны, но для целей создания анализатора Си++ программ желательно сохранить литеральные имена параметров. Решение этой проблемы можно свести к предыдущей, вводя «фиктивное» typedef объявление для функциональных типов с именованными параметрами.

4.5 Реализация системы типов

За основу реализации типов во второй версии компилятора взят описанный выше подход, основанный на представлении типов в виде ациклического графа с поиском производных типов с помощью хеширования. Каждый тип представлен как структура. Общие для всех типов поля структур выделены в отдельную структуру, которая является базовым классом для остальных структур, описывающих типы. Поля этой структуры описаны выше, в разделе 4.3.3.

Производными классами являются классы, описывающие фундаментальные типы, классовые типы, перечислимые типы и классы, описывающие каждый из перечисленных выше способов образования производных типов, кроме добавления квалификаторов.

Диаграмма классов такой реализации приводится ниже в нотации UML [30].

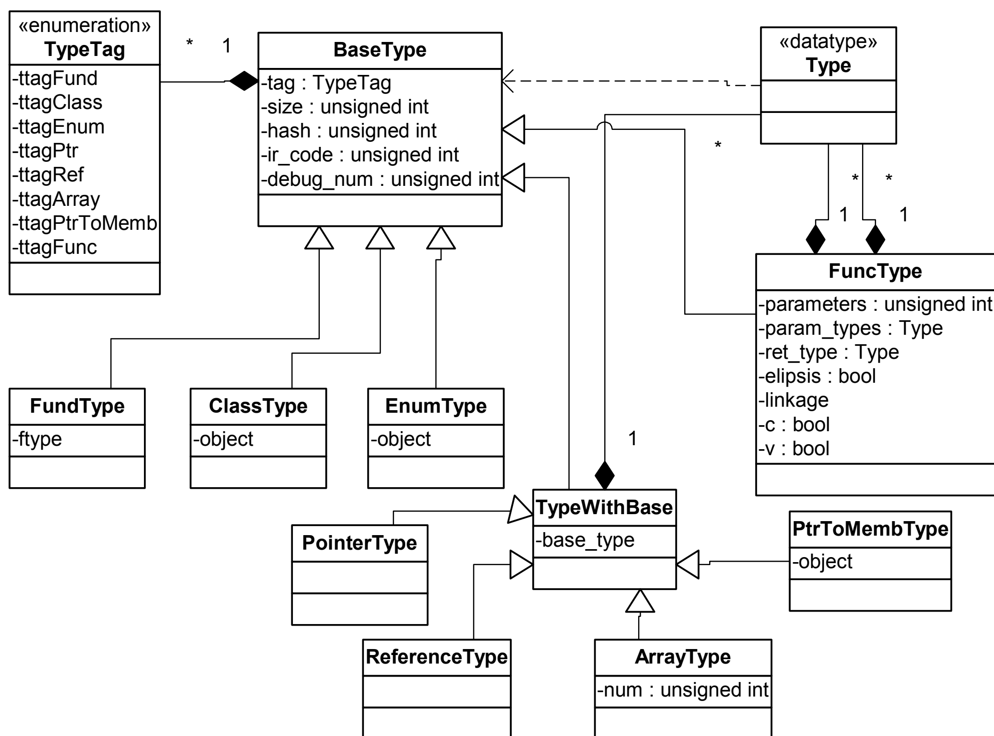


Рис. 4-6. Диаграмма классов реализации системы типов

Выделена дополнительная абстракция типа TypeWithBase - тип, построенный на основе другого типа одним из четырех способов: образованием указателя, образованием ссылки, образованием массива, образованием указателя на член класса. С помощью этой абстракции явно выделяются общие свойства таких способов образования новых типов.

Членами класса BaseType являются:

- tag перечислимого типа TypeTag (ttagFund, ttagClass, ttagEnum, ttagPtr, ttagRef, ttagArray, ttagPtrToMemb, ttagFunc) тег типа, позволяющий определить к какой группе типов принадлежит данный тип;
- size целочисленного типа - размер объектов данного типа;
- hash целочисленного типа - хеш-значение данного типа;
- ir_code - атрибут, описывающий данный тип для целей генерации кода;
- debug_num - атрибут, описывающий данный тип для целей генерации отладочной информации.

Дополнительными атрибутами классов ClassType и EnumType являются указатели object на семантические слова, описывающие данный класс или перечисление.

Дополнительным атрибутом класса TypeWithBase является указатель base_type на структуру базового типа.

Классы PointerType и ReferenceType не имеют дополнительных атрибутов.

Класс ArrayType имеет дополнительный атрибут num - число элементов массива.

Класс PtrToMembType имеет дополнительный атрибут object указывающий на семантическое слово класса.

Класс FuncType имеет несколько дополнительных атрибутов: params - число параметров функции, param_types - массив указателей на типы параметров, ret_type указатель на тип возвращаемого значения, elipsis - флаг, сигнализирующий о том, что функция принимает переменное число параметров, linkage - тип связывания функции, c и v флаги, существенные только для функций-членов классов, указывают на то, что функция описана с квалификаторами const и volatile.

Сами типы в компиляторе представлены как указатели на тип TypeBase. Квалифицированные типы реализуются по "легковесной" схеме со сдвинутыми указателями как описано выше, в разделе 4.4.4.

4.6 Преобразования типов

При реализации типов в компиляторе Си++ особое внимание приходится уделять преобразованиям типов, их семантике и эффективной реализации методов проверки преобразования типов. Это обусловлено в основном тем, что неявные преобразования типов при программировании на этом языке используются очень часто и большинство стандартных операторов являются полиморфными. Также язык поддерживает две формы полиморфизма для определенных пользователем операторов и функций для проверки семантики которых необходимо определять возможность преобразования типов и устанавливать какое из преобразований является наилучшим.

Явные преобразования типов. В языке имеется несколько вариантов явного преобразования типов, каждый из которых обладает своими особенностями. Проверка семантики явных преобразований типов не будет рассматриваться в этом разделе, ниже будет описана только свойства и реализация неявных преобразований.

Преобразования типов, определяемые пользователем. В языке Си++ имеется возможность определять пользовательские преобразования типов, которые реализуются с помощью пользовательских функций-членов преобразования типов и конструкторов.

Зависимость от контекста. Кроме прочего преобразования типов в Си++ в некоторых случаях подчиняются правилам контроля доступа, например, существуют отдельные правила проверки доступности для преобразований указателей на классы и преобразований с участием пользовательских функций преобразования. Такие преобразования могут быть семантически верными в одной части программы или неверными в другой части программы в зависимости от контекста (функции в которой преобразование производится).

Зависимость от lvalue. Некоторые преобразования семантически корректны только в том случае, когда преобразуемое значение является lvalue, но некорректны, если преобразуемое значение есть rvalue. Это единственный дополнительный атрибут преобразуемого выражения, который учитывается при проверке преобразования типов, который отсутствует в предложенной выше реализации системы типов. Является ли преобразуемое выражение lvalue или

rvalue можно узнать только из соответствующего атрибута узла дерева выражения ([85], раздел 2.2). Этот атрибут выставляется в узле дерева во время проверки типа выражения.

Ранг преобразований. При проверке семантики вызова совместно используемой функции компилятор должен сравнить два возможных преобразования и выявить какое из двух преобразований типов фактического параметра к типам формальных параметров является «лучшим». Возможны случаи, когда из двух возможных преобразований ни одно не является «лучшим». Таким образом, на множестве всех возможных преобразований определен частичный порядок по отношению `best_match`. Соответствующее отношение описано в разделе 13.3.3 стандарта языка [1]. Установить какое из двух преобразований лучше можно на основании ранга преобразования или, если преобразования имеют одинаковый ранг с использованием более тонкой проверки.

Множество возможных преобразований. Для большинства случаев семантически корректных преобразований существует возможность осуществить преобразование многими способами. Ниже будет описано как из множества возможных преобразований выбирается единственное, по правилам которого преобразование должно производиться.

Формально можно определить предикат $Conv_c(Ts, Tt, Lv, f)$, где Ts тип преобразуемого выражения, Tt – тип, к которому преобразуется выражение, Lv – булев атрибут указывающий на то является ли преобразуемое выражение lvalue и f – контекст (функция, в пределах которой производится преобразование, либо 0 если преобразование производится вне тела любой функции). Предикат истинный если неявное преобразование семантически корректно, или ложный в противном случае. Также нам понадобится другой предикат: $Conv_n(Ts, Tt, Lv)$ определенный аналогично, но без учета контекста, то есть зависимость от контекста не рассматривается и любое возможное преобразование считается допустимым, даже если оно невозможно ни в одном из существующих в программе контекстов. По определению $Conv_c(Ts, Tt, Lv, f)$ влечет $Conv_n(Ts, Tt, Lv)$

Каждое неявное преобразование можно представить как тройку вида (e_s, c, e_t) или $(e, 0, 0)$. Где e_s, e_t, e может быть одним из стандартных преобразований определяемых ниже, c – пользовательское преобразование (функция-член преобразования либо конструктор). В том случае, когда пользовательское преобразование не используется, e является преобразованием от типа Ts к типу Tt, иначе если используется пользовательское преобразование типов e_t является

преобразованием к типу *Tt* от некоторого классового типа *C*, такого, что *c* является конструктором класса *C* или функцией преобразования, определенной в классе *C*, и e_s является преобразованием от типа *Ts* к типу единственного явного параметра конструктора, либо, в случае пользовательской функции преобразования, к типу *& cv C* где наличие квалификаторов *cv* полностью совпадает с соответствующими квалификаторами пользовательской функции преобразования.

Можно построить граф, *Gt* вершинами которого являются пары *(T, L)*, где *T* – тип, *L* – булево значение. Две вершины графа *(Ts, Ls)* и *(Tt, Lt)* соединены направленным ребром в том случае, когда $Conv_n(Ts,\ Tt,\ Ls)$. Такой граф не является транзитивно замкнутым по отношению достижимости вершин. Одна вершина может быть достижима из другой и, в то же время, может и не существовать ребра непосредственно их соединяющего. Примером таких ребер может быть любая пара преобразований с участием функций преобразования типов заданных пользователем. По определению графа любому его ребру можно приписать тройку или множество троек, определяющих преобразование. Семантика языка накладывает дополнительные ограничения на использование определенных пользователем функций преобразования, поэтому не любая возможная тройка (e_s, c, e_t) порождает ребро графа.

В приведенной ниже таблице сведена информация о различных преобразованиях типов описанная в разделах 4.1-4.12, 13.3 и 12.3.

Преобразование	Раздел стандарта	Категория	Ранг	Зависимость от контекста	Дополнительные сравнения
Совпадение типов		Совпадение	Полное совпадение	Нет	Нет
Lvalue в rvalue	4.1	Преобразование lvalue			
Массив к указателю	4.2				
Функция к указателю	4.3				
Преобразование квалификаторов	4.4	Преобразование квалификаторов			
Присоединение ссылки	13.3.3.1.4	Совпадение			
		Расширение или преобразование	Преобразование	Есть	Есть
Целочисленное расширение	4.5		Расширение	Нет	
Расширение с плавающей точкой	4.6				
Преобразование с плавающей точкой	4.8		Преобразование		
Целочисленное преобразование	4.7				
Преобразование между целочисленным и типом с плавающей точкой	4.9				
Преобразование указателей	4.10				

Преобразование указателей на класс				Есть	
Преобразование указателей на член класса	4.11				
Преобразование к булеву типу	4.12				
Преобразование с использованием пользовательской функции	12.3		Пользовательское преобразование		

Табл. 3. Сводная таблица различных неявных преобразований типов

Стандартные преобразования. Стандартное преобразование является цепочкой преобразований, определенных в разделе 4 стандарта [1]. В таблице эти преобразования не выделены цветом фона. Так как цепочки можно произвольно соединять такие преобразования порождают подграф графа *Gt* транзитивно замкнутый по отношению достижимости.

Каждую цепочку стандартного преобразования можно привести к каноническому виду, определенному в разделе 13.3.3.1.1 стандарта [1]. В каноническом виде любая цепочка выглядит как последовательность не более трех преобразований из различных категорий, приведенных в сводной таблице, в следующем порядке: преобразование lvalue, расширение или преобразование, преобразование квалификаторов. Использование этого свойства существенно упрощает поиск нужной цепочки преобразования.

Возможность использования пользовательского преобразования. Другой особенностью этого графа является то, что в если типы *Ts* и *Tt* не являются классовыми, типами ссылки на класс и их квалифицированными вариантами и верно $Conv_n(Ts, Tt, Ls)$, то преобразование между двумя этими типами является стандартным, то есть может быть выполнено без участия пользовательской функции преобразования. Такие преобразования также порождают в графе *Gt* замкнутый по отношению достижимости подграф. Таким образом, при реализации соответствующих предикатов можно легко проверить возможность использования пользовательских преобразований и отсечь большое число операций поиска и сравнения.

Предпочтительное преобразование. В некоторых случаях при использовании пользовательских преобразований типа может оказаться, что одному ребру приписано несколько возможных троек определяющих преобразование. В таких случаях предпочтительным всегда считается стандартное преобразование, если оно возможно. В остальных случаях преобразование возможно только при помощи

различных пользовательских функций преобразования. В таких случаях преобразование считается неоднозначным и семантически неверным. Можно привести примеры, в которых неоднозначности вызваны двумя различными пользовательскими преобразованиями типов или двумя конструкторами или пользовательским преобразованием типа и конструктором.

Семантически корректное преобразование. Некоторые виды преобразований, отдельно выделенные в приведенной выше таблице, зависимы от контекста в том смысле, что они могут быть корректны в одной точке программы, но некорректны в другой точке программы. Возможные варианты таких семантических ошибок:

- Использование неполно определенного класса.
- Преобразование указателей на недоступный базовый класс.
- Использование недоступной пользовательской функции преобразования.

Первый вид семантических проверок целиком зависит от наличия полного описания класса и легко реализуется. Второй и третий вид проверок реализуется как проверка доступности базового класса или члена класса, целиком определяется контекстом, в котором производится преобразование и подробно рассматривается в разделе 1.5 данной работы.

Ранжирование преобразований. При выборе наиболее подходящей функции из множества совместно используемых функций необходимо иметь возможность сравнивать различные неявные преобразования типов фактических параметров к типам формальных параметров функций. Такие сравнения определяются на тройках преобразований. Полностью правила сравнения преобразований приведены в разделе 13.3.3.2 стандарта [1]. Эти правила достаточно громоздки, здесь лишь необходимо отметить, что в простейших случаях такое сравнение можно произвести на основании ранга преобразования, приведенного в сводной таблице, и лучшими являются преобразования в следующем порядке рангов.

1. Полное совпадение.
2. Расширение.
3. Преобразование.
4. Пользовательское преобразование.

Если на основании ранга операции невозможно из двух преобразований невозможно выбрать лучшее, необходима более тонкая проверка, наличие дополнительных правил проверки отражено в четвертом столбце таблицы.

4.7 Реализация проверки типов для операций и выбора наилучшей совместно используемой функции

Взяв за основу описанные выше реализацию системы типов и свойства неявных преобразований типов, можно приступить к описанию реализации проверки типов. Ниже будут приведены описания основных функций, использованных для реализации проверки типов, и описаны основные идеи их реализации. Описываемые ниже процедуры проверки типов реализуют наиболее сложную часть проверки типов в языке Си++.

Функция `bool standard_conversion (Type source_type, Type target_type, bool lvalue)` возвращает булево значение – возможность преобразования значения типа `source_type` к типу `target_type` при условии, что преобразуемое значение является lvalue в зависимости от параметра `lvalue`. Функция реализует правила перечисленные в разделе 4 стандарта [1] и правила описанные в разделах 8.5.3 и 13.3.3.1.4 для случая преобразования к ссылочному типу. Благодаря эффективной реализации системы типов, особенно проверки типов на принадлежность определенным группам, эта функция выполняется очень быстро, что существенно влияет на скорость компиляции, так как такие проверки производятся очень часто. Функция не проверяет семантическую корректность преобразования для преобразований указателей на классы и указателей на члены классов, но тем не менее использует механизмы, описанные в первой главе, для проверки того, что классы находятся в отношении наследования.

Функция `bool direct_binding (Type source_type, Type targer_type, bool lvalue)` возвращает булево значение в зависимости от того, необходимо ли создание временного объекта при преобразовании значения типа `source_type` к ссылочному типу `target_type`. Если функция возвращает false это не означает, что создание временного объекта для инициализации необходимо, это лишь означает, что присоединение ссылки невозможно к преобразуемому объекту. Также как и предыдущая функция, данная функция не производит семантических проверок на доступность базового класса.

Функция `ICS find_ics (Type source_type, Type targer_type, bool lvalue)` возвращает структуру типа `ICS`, описывающую неявное преобразование для приведения значения типа `source_type` к типу `target_type`. Эта структура

является реализацией множества троек, описывающих неявное преобразование, описанных выше, при обсуждении свойств неявных преобразований. Эта функция использует функцию `standard_conversion` для проверки возможности выполнения стандартного преобразования, и если стандартное преобразование возможно, функция немедленно возвращает такое стандартное преобразование. В противном случае функция выполняет поиск пользовательских преобразований и проверку того, что пользовательское преобразование может быть дополнено двумя стандартными преобразованиями для того, чтобы сформировать полное преобразование типов. Если ни тип `source_type`, ни тип `target_type` не является классовым типом (включая варианты типов производных от классового по правилам образования квалифицированного или ссылочного типа) такое преобразование найти невозможно и возвращается специальное значение, сигнализирующее о том, что множество подходящих троек пусто.

Если `source_type` является классовым типом в классе и рекурсивно в его базовых классах производится поиск всех пользовательских функций преобразования, при этом учитываются правила скрытия имен для функций преобразования типа сформулированные в разделе 12.3.2 стандарта [1]. После того как все функции преобразования найдены, производится проверка на то, что они удовлетворяют требованиям на соответствие типу неявного параметра (в зависимости от квалификаторов типа `source_type`) и существование стандартного преобразования к типу `target_type`, что также реализовано с использованием функции `standard_conversion`. Так как поиск функций преобразования является рекурсивным по базовым классам и, кроме этого, производится проверка функций по правилам скрытия имен, для каждого класса такой поиск производится только один раз, а затем результаты сохраняются и в случае необходимости используются повторно. При этом используется тот факт, что все функции преобразования всегда определяются в теле класса и как только класс полностью определен добавление новых функций преобразования в этот класс невозможно.

Если тип `target_type` является классовым типом, то рассматриваются также все конструкторы этого класса, но не его базовых классов. Каждый конструктор проверяется на число явных неумалчиваемых параметров и с помощью функции `standard_conversion` проверяется возможность преобразования фактического параметра типа `source_type` к типу формального параметра конструктора.

Функция **`Type check_conversion(ICS &conv, Node *node, Object context)`** производит проверку семантики найденного ранее преобразования типов `conv` в контексте функции `context` применительно к узлу дерева выражения `node`. В первую очередь проверяется однозначность преобразования заданного `conv`. Как указывалось ранее, `conv` представляет собой реализацию множества троек преобразования. Если в множестве более одного элемента, то такое преобразование является неоднозначным. Если же в множестве нет ни одного преобразования, то преобразование невозможно. Для преобразований типов, таких как преобразования с использованием пользовательских преобразований, выполняется проверка доступности соответствующей функции. Для стандартных преобразований указателя или ссылки на производный класс к указателю или ссылке на базовый класс производится проверка на однозначность и доступность базового класса в контексте `context`. Оба вида проверок используют механизмы описанные в первой главе настоящй работы.

В случае выявления любой семантической ошибки выдается диагностическое сообщение, которое содержит информацию о позиции в программе узла node.

Функция возвращает значение типа `Type`, соответствующее типу, к которому производится преобразование.

Функция **`void insert_conversion (ICS &conv, Node *node, Node *statement, Object context)`** производит проверку семантики преобразования `conv` и, если проверка не выявила ошибок, вставляет в дерево специальный узел неявного преобразования, который содержит информацию, необходимую генератору кода для генерации кода преобразования, включая, если это необходимо, генерацию кода вызова пользовательской функции преобразования и создания временных объектов. Узел `statement` используется если для выполнения преобразования необходимо создание временного объекта классового типа с нетривиальным деструктором. В этом случае после выполнения вычисления выражения должен быть вызван деструктор для временного объекта. Генерация кода для таких выражений усложняется тем, что в некоторых случаях вычисляется только часть выражения, например из двух подвыражений условной операции вычисляется только одно, соответственно временные объекты могут реально и не создаваться и для них не должны вызываться деструкторы.

Функция **Ranc ranc_convertion (ICS &conv)** возвращает значение перечислимого типа – ранг преобразования `conv`. Затем значения, возвращаемые этой функцией, могут сравниваться для выявления преобразования имеющего лучший ранг. Эта функция не выполняет никаких детальных проверок и вычисляется эффективно.

Функция **int compare_convertions (ICS &conv1, ICS &conv2)** выполняет полное сравнение двух неявных преобразований по правилам сформулированным в разделе 13.3.3.2 стандарта [1]. Функция возвращает значения –1, если преобразование `conv1` “лучше” преобразования `conv2`; 1 - если преобразование `conv2` “лучше” преобразования `conv1` и 0 если ни одно из преобразований не лучше другого. Эта функция не выполняет семантических проверок и не вызывает никаких диагностических сообщений.

Функции **ObjectList candidate_functions (Object function) и ObjectList candidate_operators (Node *node)** возвращают списки пользовательских функций и определенных пользователем функций-операторов для вызова функции и узла оператора соответственно. В списки также включаются шаблоны функций и операторов если таковые найдены. Реализация поиска основана на механизмах поиска, подробно описанных в третьей главе.

Первая функция в качестве параметра получает дескриптор функции или шаблона функции, найденной механизмом поиска имен при построении дерева, и возвращает список всех совместно используемых функций, среди которых должен производиться выбор наилучшей на основании типов параметров. Правила построения такого списка сформулированы в разделе 13.3.1.1.1.

Вторая функция составляет список функций-операторов и функций преобразования, соответствующих узлу оператора node. Правила поиска таких функций сформулированы в разделах 13.3.1.1.2 и 13.3.1.2-13.3.1.6 стандарта [1]. В случае операции вызова должен производиться поиск как операторов вызова, так и операторов преобразования к типам указателя на функцию. Следующий пример корректной программы демонстрирует причины, по которым должен производиться поиск операторов преобразования:

```
typedef void (*PF)(int);
class A {
        operator PF();
};
void f()
{
        A a;
```

```
        a(1);
    }
```

В большинстве случаев поиск определенных пользователем операторов должен производиться в по общим правилам поиска имен вне контекста классов. В случае, когда единственным операндом унарной операции или левым операндом бинарной операции является объект классового типа, поиск также должен производиться в контексте класса, в том числе рекурсивно в его базовых классах. В отдельных частных случаях (операторы `=`, `->`, `[]`, `()`) пользовательские операторы обязаны быть членами класса, в этом случае поиск производится только в контексте классов, что позволяет избежать ненужных операций поиска имен.

Функции `ViableFunctionsList viable_functions (ObjectList &candidates, ParameterList ¶meters)` и `ViableFunctionsList viable_operators (ObjectList &candidates, ParameterList ¶meters)` осуществляют проверки и отсеивание заведомо неподходящих функций для функций кандидатов, переданных в качестве параметра `candidates`, на основании списка фактических параметров переданного в качестве параметра `parameters`. Правила для отсеивания заведомо неподходящих функций сформулированы в разделе 13.3.2. Эти проверки включают в себя проверку числа формальных и фактических параметров, а также проверку при помощи функции `find_ics` возможности преобразования фактических параметров к типам формальных параметров. На этом этапе не проводятся никакие семантические проверки преобразований кроме проверки на существование хотя бы одного возможного преобразования.

Для шаблонов функций эти функции генерируют настройки шаблонных функций, подходящих для конкретных типов параметров.

Элементами возвращаемого списка являются функции и объекты типа `ICS`, описывающие возможные преобразования типов параметров.

Функции `convert_parameters (ViableFunction function, ParameterList ¶meters, Object context)` и `convert_parameters (Type function_type, ParameterList ¶meters, Object context)` осуществляет проверку семантики преобразований фактических параметров, переданных ей в качестве параметра `parameters`, к типам формальных параметров функции, переданной в качестве параметра `function`, и вставляет в деревья параметров узлы преобразования. Проверка и вставка узлов

осуществляется при помощи функции insert_conversion. Первый вариант функции осуществляет обработку параметров для вызова известной функции с известными неявными преобразованиями параметров, второй вариант функции сначала осуществляет поиск преобразований для каждого фактического параметра при помощи функции find_ics, а затем проверку и вставку узла преобразования.

Обе функции используют общую вспомогательную функцию convert_parameter, осуществляющую основную часть их работы для одного параметра. Эта функция также реализует семантику передачи параметра по эллипсису и вставку в дерево ссылок для генерации параметров по-умолчанию.

Функция Type best_matching (Object function, Node *call_node) осуществляет поиск и проверку типов для операции вызова функции в том случае, когда в левой части операции вызова находится именованная функция.

На первом шаге проверки функция рекурсивно осуществляет проверку типов для каждого фактического параметра. При этом строится список ParameterList.

Затем строится список функций при помощи функции candidate_functions. Этот список не может быть пустым, так как одна функция уже передана в качестве параметра. Этот список сокращается при помощи функции viable_functions. Если полученный сокращенный список пуст генерируется диагностическое сообщение, содержащее список функций-кандидатов и проверка прекращается. Диагностическое сообщение позиционируется по узлу вызова call_node.

На следующем этапе при помощи вызова функции compare_ics сравниваются различные подходящие функции и делается попытка найти наилучшую функцию. Если такая функция не найдена генерируется диагностическое сообщение содержащее список подходящих функций.

На этом этапе известна единственная функция, которая должна вызываться. Для нее вызывается функция convert_parameters осуществляющая семантические проверки преобразований параметров.

Функция возвращает тип значения, возвращаемого выбранной наилучшей функцией.

Функция Type check_operator (Node *oper_node) осуществляет проверки, аналогичные описанной выше функции best_matching, для операторов. С

помощью функции candidate_operators составляется список возможных операторов. Основное отличие от функции best_matching заключается в том, что на этом этапе список кандидатов пополняется встроенными функциями-кандидатами по правилам сформулированным в разделе 13.6 стандарта [1]. Эти дополнительные функции-кандидаты представляют стандартные операторы, такие как, например, оператор сложения двух целых чисел.

Пополненный список проверяется функцией viable_operators по общим правилам и, затем, по общим же правилам из списка выбирается наиболее подходящий оператор. Если наиболее подходящим оператором оказался пользовательский оператор устанавливается поле узла оператора, сигнализирующее кодогенератору о том, что для данного узла должен генерироваться код вызова пользовательской функции.

Общие с функцией best_matching правила применяются для преобразования типов операндов, независимо от того, оказался ли лучшим пользовательский или встроенный вариант функции.

Функция Type check_call (Node *call_node) производит проверку типов для выражения вызова функции. Сначала проверяется тип значения стоящего в левой части операции вызова функции. Если в левой части находится узел дерева ссылающийся на функцию, для всего выражения вызова вызывается функция best_matching. Иначе, если в левой части находится произвольное выражение, имеющее тип указатель на функцию, для всего выражения вызова применяется второй вариант функции convert_parameters. Иначе для всего выражения вызывается функция check_operator.

Функция Type check_types (Node *node) производит проверку типов для произвольного узла выражения, включая операцию вызова функции. Эта функция проверяет типы для произвольной операции в дереве выражения, включая операции явного преобразования типов, проверки операций new и throw, которые не рассматривались выше. Для проверки типов операции вызова эта функция вызывает функцию check_call, а для проверки бинарных и унарных операций вызывается функция check_operator. Проверка типов для остальных видов операций, не рассмотренных выше, также реализована в отдельных функциях.

Эта функция прямо и косвенно вызывается рекурсивно для проверки типов подвыражений.

Взаимодействие функций проверки типов.

В завершении этого раздела приведем диаграмму взаимодействия описанных в этом разделе функций, реализующих проверку типов.

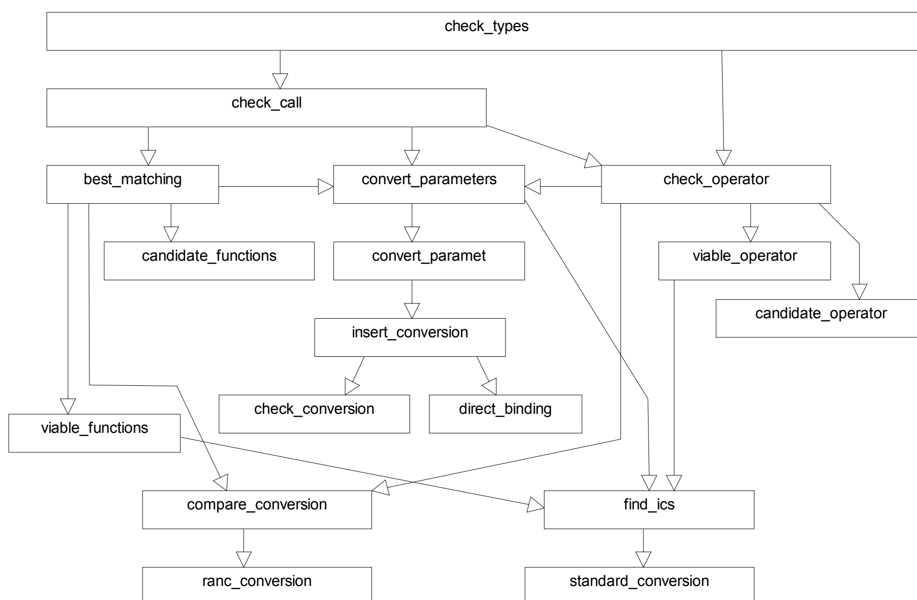


Рис. 4-7. Взаимодействие функций проверки типов.

4.8 Заключение. Выводы главы 4

В этой главе были рассмотрены методы реализации типов в компиляторе Си++. Подробно рассмотрен метод реализации основанный на хеш-таблице, а также обсуждались основные идеи реализации проверки типов.

Основные результаты четвертой главы могут быть сформулированы следующим образом:

1. Изучены особенности системы типов языка Си++ и ее отличия от систем типов в родственных языках программирования.
2. Сформулированы требования к реализации системы типов в компиляторе Си++.
3. Проанализированы недостатки традиционного способа реализации типов в виде цепочек при использовании в компиляторе Си++.
4. Предложен метод реализации системы типов в компиляторе на основе хеш-таблицы удовлетворяющая сформулированным требованиям.
5. Исследованы особенности проверки типов в языке Си++ и требования к реализации проверки типов в компиляторе.
6. Описана часть реализации проверки типов для операций, учитывающая возможность перегрузки операторов и использования операторов преобразования типов, определяемых пользователем.

Заключение

В работе получены следующие основные результаты:

1. Дано формальное определение графа наследования, на основе которого проведенео систематическое исследование семантических свойств механизма наследования классов в языке Си++. Построена и изучена формальная модель (граф подобъектов) строения объектов классов, учитывающая особенности языка. На основании формальной модели предложен способ эффективного вычисления числа подобъектов базового класса и дано формальное определение однозначности базового класса. Дано определение доминирующего класса, основанное на формальной модели, и предложен способ эффективного определения доминирующего класса. Предложен способ определения доступности базового класса и члена базового класса на основании графа наследования.
2. Разработаны методы и структуры данных компилятора, делающие возможным эффективную реализацию семантических проверок, связанных с отношениями между классами.
3. Разработаны алгоритмы генерации машинно-независимого низкоуровневого промежуточного кода и структур данных (таблиц) периода выполнения, реализующие семантику механизмов классов. В частноси, предложены следующие решнения: способ реализации множественного и виртуального наследования; метод реализации виртуального наследования, основанный на генерации таблиц смещений виртуальных базовых классов; алгоритмы генерации таких таблиц; метод генерации кода для доступа к членам классов и членам базовых классов; способ реализации преобразований указателей и ссылок на классы, учитывающий особенности множественного и виртуального наследования; метод реализации виртуальных функций, основанный на генерации таблиц. Дано определение доминирования для виртуальных функций и предложен способ его выявления. Описаны методы определения абстрактности классов и неявной виртуальности функций. Предложен метод реализации и преобразования указателей на члены классов. Описаны методы реализации операторов `typeid` и `dynamic_cast` с использованием объектов расширенного класса `type_info`. Предложен метод реализации динамического определения типов для исключительных ситуаций.

4. Исследована сложность алгоритмов генерации таблиц, используемых для реализации механизма классов, и даны оценки размера требуемых таблиц.
5. Разработаны структуры данных компилятора, используемые механизмами времени компиляции при генерации таблиц, реализующих механизмы виртуальных функций и виртуального наследования.
6. Исследованы особенности поиска имен в Си++ и отличия правил поиска имен в этом языке от правил поиска имен в других языках программирования. Сформулированы требования к реализации поиска имен в компиляторе.
7. Предложен эффективный способ проверки наличия определения имени в различных областях действия с использованием хеш-таблицы. Описана реализация механизма поиска имен, эффективно реализующая правила языка, использующая граф областей действия и дисплей областей действия. Формализованы условия, при которых объявленные в области действия имена становятся недоступными и могут быть удалены из хеш-таблицы. Предложен метод реализации конструкций `using` и `using namespace`. Описано взаимодействие механизма поиска имен с другими частями компилятора. Описано взаимодействие частей компилятора реализующих различные варианты поиска имен.
8. Исследованы особенности системы типов языка Си++ и ее отличия от систем типов в родственных языках программирования. Сформулированы требования к реализации системы типов в компиляторе Си++.
9. Проанализированы недостатки традиционного способа реализации типов в виде цепочек при использовании в компиляторе Си++. Предложен метод реализации системы типов в компиляторе на основе хеш-таблицы, удовлетворяющий сформулированным требованиям. Описана реализация типов как составная часть промежуточного представления ТТТ.
10. Исследованы особенности проверки типов в языке Си++ и требования к реализации проверки типов в компиляторе.
11. Описана часть реализации проверки типов для операций, учитывающая возможность перегрузки операторов и наличие операторов преобразования типов, определяемых пользователем. Описана часть реализации проверки типов при вызове совместно используемых функций.

Литература

1. ISO/IEC 14882:1998 *Programming languages -- C++*.
2. ISO/IEC DIS 9899 *Programming languages -- C*.
3. ISO/IEC 9899:1990 *Programming languages -- C*.
4. ISO/IEC 8652:1995 *Information Technology--Programming Languages—Ada.*
5. ISO/IEC 15291:1999, *Information Technology--Programming Languages--Ada Semantic Interface Specification (ASIS).*
6. ISO/IEC 18009:1999, *Conformity Assessment of an Ada Language Processor.*
7. ISO/IEC 13816:1997 - *Programming Language ISLISP.*
8. ISO/IEC TR 9547:1988 *Test methods for programming language processors - guidelines for their development and procedures for their approval*. 1993.
9. ISO/IEC TR 10034:1990 *Guidelines for the preparation of conformity clauses in programming language standards*. 1995.
10. NCITS technical committees**,** *J20 - Programming Language Smalltalk*, working papers, http://www.ncits.org/tc_home/j20sd4.htm.
11. Bancilhon F., Ferran G. *ODMG-93: the object database standard*. Data Engineering Bulletin, December 1994, vol. 17., N 4 .
12. R. G. Cattel *The Object Database standard: ODMG 2.0* The Morgan-Kaufmann series in Database Management systems, 1997.
13. D.E.Knuth *The Art of Computer Programming*, Second edition. Volume 3: *Sorting and Searching*. Addison-Wesley, 1998. Имеется русский перевод:
Д.Э. Кнут. *Искусство программирования,* Второе издание. Том 3. *Сортировка и поиск*. “Вильямс”, 2000.
14. A.V. Aho, R. Sethi, J.D. Ullman *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985. Имеется русский перевод:
А. Ахо, Р.Сети, Дж. Ульман. *Компиляторы: принципы, технологии, инструменты.* “Вильямс”, 2001.
15. В.А. Серебряков, М.П. Галочкин *Основы конструирования компиляторов*. Москва, УУРС, 2000.
16. S. Rybin, A. Strohmeier, E. Zueff *ASIS for GNAT: Goals, Problems and Implementation Strategy*, Ada-Europe'95 Conference, Frankfurt, Germany, October 1995.
17. S. Rybin, A. Strohmeier, E. Zueff *ASIS Implementation for the GNAT Compiler*, First Workshop on Free Software, Universidad Carlos III, Madrid, Spain, September 1995.
18. B.W. Kernighan, R.Pike *The Practice of Programming*. Addison-Wesley, 1999. Имеется русский перевод:
Б.Кернинган, Р. Пайк *Практика программирования*. СПб: Невский диалект, 2001.
19. B.W. Kernighan, R.Pike *The UNIX Programming Environment*. Prentice-Hall, 1984. Имеется русский перевод:
Б.Кернинган, Р. Пайк *UNIX – универсальная среда программирования*. Москва: Финансы и Статистика, 1992.

20. B.W. Kernighan, D.M. Ritchie *The C Programming Language*. Prentice-Hall, 1978. Имеется русский перевод:
Б.Кернинган, Д. Ритчи *Язык программирования Си*. Москва: Финансы и Статистика, 1992.

21. B. Stroustrup *Adding Classes to the C Language: An Exercise in Language Evolution* Software - Practice and Experience, pp 139-161. February 1983.

22. Ellis M. A., Stroustrup B., *The Annotated C++ Reference Manual* (ARM), Reading, MA: Addison-Wesley, 1990. Имеется русский перевод:
Эллис М., Строуструп Б. Справочное руководство по языку программирования C++ с комментариями: Пер. с англ.- М.: Мир, 1992.

23. B. Stroustrup *What is "Object-Oriented Programming"?* (1991 revised version), AT&T Bell Laboratories, Murray Hill, New Jersey 07974.
http://www.research.att.com/~bs/whatis.pdf

24. B. Stroustrup *Why C++ is not just an Object-Oriented Programming Language*, AT&T Bell Laboratories, Murray Hill, New Jersey 07974. Invited talk given at OOPSLA'95 in Austin Texas.
http://www.research.att.com/~bs/oopsla.pdf

25. B. Stroustrup *The C++ Programming Language*. The third edition. Addison-Wesley, 1997. Имеется русский перевод:
Б. Страуструп *Язык программирования Си++*, М; СПб: Невский диалект, Бином, 1999.

26. B. Stroustrup *The Design and Evolution of C++*. Addison-Wesley, 1994. Имеется русский перевод:
Б. Страуструп *Дизайн и эволюция языка C++*. М: ДМК, 2000.

27. G. Booch *Object-Oriented Design with Applications*, Benjamin Cummings, 1991. Имеется русский перевод:
Г.Буч, *Объектно-ориентированное проектирование с примерами применения* М., "Конкорд", 1992.

28. G. Booch *Object-Oriented Analysis and Design with Applications*, Second edition, Addison-Wesley, 1994. Имеется русский перевод:
Г.Буч *Объектно - ориентированный анализ и проектирование с примерами приложений на C++*, М: Бином, Невский диалект, 1998.

29. I. Pohl *Object-Oriented Programming Using C++*. Second edition. Имеется русский перевод:
А. Пол *Объектно-ориентированное программирование на Си++*. Второе издание М: Бином, 2001.

30. G. Booch, J. Rumbaugh, I. Jacobson *The Unified Modelling Language User Guide*. Addison-Wesley, 1999. Имеется русский перевод:
Г. Буч, Д. Рамбо, А. Джекобсон *UML: руководство пользователя*, ДМК, 2000.

31. G. Booch, J. Rumbaugh, I. Jacobson *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999.

32. D. C. Schmidt, S. Huston *C++ Network Programming: Mastering Complexity with ACE and Patterns*, Addison-Wesley Longman, 2002.

33. D. C. Schmidt, M. Stal, H. Rohert, F. Buschmann *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.

34. D.C. Schmidt *The ACE Programmer's guide* http://www.cs.wustl.edu/~schmidt/PDF/ACE-tutorial.pdf

35. C. J. Date, H. Darwen *Foundation for Object / Relational Databases: The Third Manifesto*. Second Edition. Addison-Wesley, 1998.

36. Л.А. Калиниченко *Стандарт систем управления объектными базами данных ODMG-93: краткий обзор и оценка состояния*, Москва. Открытые системы: Системы управления базами данных, №01, 1996. http://www.osp.ru/dbms/1996/01/46.htm

37. M. H. Austern *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999.

38. P. J. Plauger, A. Stepanov, M. Lee, D. Musser *The Standard Template Library*, Prentice-Hall, 2001.

39. N. Josuttis *The C++ Standard Library – A Tutorial and Reference*. Addison-Wesley, 1999.

40. L. Cardelli, P. Wegner *On understanding types, data abstraction, and polymorphism*. Computing Surveys, 17(4):471-522, 1985. http://research.microsoft.com/Users/luca/Papers/OnUnderstanding.A4.pdf

41. M. Abadi, L. Cardelli. *A Theory of Objects*. Springer, 1996.

42. J.G. Rossie, M.P. Friedman, M.Wand *Modelling Subobject-Based Inheritance.* ECOOP'96 Conference Proceedings, July 8-12, Linz, Austria. ftp://ftp.ccs.neu.edu/pub/people/wand/papers/ecoop-96.ps

43. M.M. Cristensen *Methods of Handling Exceptions in Object-Oriented Programming Languages*. M.Sc thesis, Odense University, Denmark, 1995. http://citeseer.nj.nec.com/christensen95method.html

44. A. Koenig, B. Stroustrup *Exceptions Handling in C++*. Journal of Object-Oriented Programming, July/August 1990.

45. D. Cameron, P. Faust, D.Lenkov, M.Mehta *A Portable Implementation of C++ Exception Handling.* Usenix C++ Conference Proceedings, Portland, 1992.

46. Budge, Kent G., James S. Peery, and Allen C. Robinson *High Performance Scientific Computing Using C++.* Usenix C++ Conference Proceedings, Portland, 1992.

47. Ю. В. Баскаков *Принципы построения тестовых комплексов для тестирования конформности компиляторов стандартам языков программирования*. Теоретические и прикладные проблемы информационных технологий: сборник трудов. Москва 2001.

48. А. С. Морозов *Модель организации и функции лаборатории многоплатформенного тестирования архитектурно-нейтральных программ*. Теоретические и прикладные проблемы информационных технологий: сборник трудов. Москва 2001.

49. Р.Ю. Рогов, Ю. В. Баскаков *Разностное сравнение стандартов языков программирования как основа построения многоязыковых систем тестирования конформности компиляторов*. Теоретические и прикладные проблемы информационных технологий: сборник трудов. Москва 2001.

50. И.В.Поттосин *Текущее состояние российских исследований и разработок в области трансляции* // Новосибирск, 1995, 32 с. - (Препр./РАН Сиб. отделение, ИСИ; №30). См. также И.В.Поттосин, Российские исследования

по языкам программирования и трансляции.
http://www.feedback.ru/yurix/articles/pottosin.htm

51. G. Ramalingam, H. Srivasan *A Member Lookup Algorithm for C++*, IBM T.J.Watson Research Center, ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas 1997. http://www.cs.rice.edu/~grosul/612f00/mem_lookup_alg_cpp.pdf

52. Graham S.L., Joy W.N. *Hashed Symbol Tables for Languages with Explicit Scope Control*, SIGPLAN Notices 1979, v.14, №8, pp 50-57.

53. Stephen C.Dewhurst *Flexible Symbol Table Structures for Compiling C++*, Software - Practice and Experience, Vol. 17(8), 503-512 (August 1987).

54. Richard M. Stallman *Using and Porting GNU CC*, Free Software Foundation, 1995. http://www.delorie.com/gnu/docs/gcc/gcc_toc.html

55. *The C++ Front End. Internal documentation (excerpt)*, *Version 2.40*, December 8, 1998, Edison Design Group, Inc. http://www.edg.com/cpp.html

56. Ian Joyner, *C++?? A Critique of C++*, 2nd Edition, c/- Unisys - ACUS, 115 Wicks Rd, North Ryde, Australia 2113, 1992.

57. Markku Sakkinen, *The darker side of C++ revisited*, Department of Computer Science and Information Systems, University of Jyväskylä, PL 35, SF-40351 Jyväskylä, Finland, 1993-1-13.

58. P. J. Moylan, *The case against C*, The ModulaTor, Vol. 3, No 6, pp 1-11, ModulaWare GmbH, 1993. Имеется русский перевод:
П. Мойлан. *Критика языка С* Технология программирования, Т.1, с.34-40, 1995.

59. D. L. Shang, *A Critical Comparison on Transframe, Java & C++*, Object Currents, 1996. http://www.visviva.com/transframe/tf_criti.htm

60. B. Kernighan, *Why Pascal is not my favorite programming language*. 1981. http://cm.bell-labs.com/cm/cs/cstr/100.ps.gz

61. J. Woehr, *Donald Knuth Interview,* Dr.Dobb"s Journal, №3, 1996.

62. Н. Вирт, *Долой пухлые программы*, Открытые системы, №6, 1996 http://www.osp.ru/os/1996/06/27.htm

63. C.S.Johnson, *A Tour Through the Portable C Compiler*. http://plan9.bell-labs.com/7thEdMan/vol2/porttour.bun.

64. *DWARF Debugging Information Format Specification, Version 2.0*, Tool Interface Standards (TIS), TIS Committee, May 1995. http://www.intel.com/vtune/tis.htm.

65. *Tool interface standard (TIS) Portable executable and Linkable Format Specification. Version 1.2*. TIS Committee May 1995. http://x86.ddj.com/ftp/manuals/tools/elf.pdf

66. J. Menapace, J. Kingdon, D. MacKenzie, *The "stabs" debugging information format*, Free Software Foundation, Inc., 1993. Contributed by Cygnus Support.

67. Д. Фомин, В. Черников, П. Виксне, М. Яфраков, П. Шевченко. *Новая DSP-архитектура NeuroMatrix и традиционный RISC – единое вычислительное ядро процессора NM6403*, Открытые системы, 5-6/1999.

68. П.Е.Виксне, Д.В.Фомин, В.М.Черников, НТЦ «Модуль», *Однокристальный цифровой нейропроцессор с переменной разрядностью операндов*, Известия Вузов, Приборостроение, 1996, т.39, №7, стр.13-21,

69. Зуев Е.А., Кротов А.Н. *Новые возможности Си++*. PC Magazine/ Russian Edition, Октябрь 1994.

70. Кротов А.Н. *Алгоритмы реализации механизма классов объектно-ориентированного языка программирования С++.* Дипломная работа. Кафедра математической логики и теории алгоритмов. Механико-математический факультет МГУ. Москва, 1995.

71. Зуев Е.А., Кротов А.Н., Сухомлин В.А. *Язык программирования Си++: этапы эволюции и современное состояние*. Тез. докл. Первой российской конференции "Индустрия программирования'96", Москва, 3-4 октября 1996 г. http://www.citforum.ru/programming/prg96/76.shtml.

72. Зуев Е.А., Кротов А.Н., Сухомлин В.А., Давыдов Д., Недиков Н.Н. *Система программирования тройного стандарта 3С++*. Тез. докл. Первой российской конференции "Индустрия программирования'96", 3-4 октября 1996 г., Москва. http://www.citforum.ru/programming/prg96/94.shtml.

73. Кротов А.Н., Зуев Е.А., *Подход к реализации исключительных ситуаций в компиляторе языка Си++.* “Компьютерные аспекты в научных исследованиях и учебном процессе”. Сборник по материалам научной конференции МГУ “Ломоносовские чтения - 96”. Москва 1996.

74. Зуев Е.А., Кротов А.Н. *Компилятор полного стандарта Си++*. Известия ВУЗов. Приборостроение, том 40, №3, 1997.

75. Кротов А.Н., Миронов А.Г. *Генерация кода в компиляторе Си++*. Известия ВУЗов. Приборостроение, том 40, №3, 1996.

76. Зуев Е.А., Кротов А.Н., Сухомлин В.А. *Система программирования тройного стандарта 3С++*. Тез. докл. IV Международной конференции "Развитие и применение открытых систем", Н.Новгород, 27-31 октября 1997. http://www.rapros97.nnov.ru/reports/15.html

77. Кротов А.Н. *Особенности правил поиска имен в языке программирования Си++*. Теоретические и прикладные проблемы информационных технологий: сборник трудов. Москва 2001.

78. Кротов А.Н. *Реализация поиска имен в компиляторе Си++*. Теоретические и прикладные проблемы информационных технологий: сборник трудов. Москва 2001.

79. Зуев Е.А. *Системное программное обеспечение цифрового нейронного процессора с переменной разрядностью операндов*. Известия ВУЗов. Приборостроение, том 39, №7, 1996.

80. Зуев Е.А. *Редкая профессия*. PC Magazine/Russian Edition. Спецвыпуск № 5(75), 1997.

81. Зуев Е.А. *Общий подход к созданию базового программного обеспечения нейропроцессора. Известия ВУЗов*. Приборостроение, том 40, №3, 1997.

82. Зуев Е.А., Котиков С.В., Челюбеев И.А. *Язык ассемблера нейропроцессора и его реализация в инструментальной среде*. Известия ВУЗов. Приборостроение, том 40, №3, 1997.

83. Зуев Е.А. *Реализация компилятора полного стандарта Си++ и виртуальной машины Си++*, Интернет-форум "Языки программирования и Интернет", ИнфоАрт, июль 1998, http://www.infoart.ru/

84. Зуев Е.А. *Русские "плюсы"*. Мир ПК, № 4, 1999.

85. Зуев Е.А. *Принципы и методы создания компилятора переднего плана Стандарта Си++.* Диссертация на соискание ученой степени кандидата физико-математических наук. Москва, 1999.

86. Зуев Е.А. *Унифицированное семантическое представление программ как ядро системы анализа и разработки.* Всероссийская конференция "Теоретические и прикладные проблемы информационных технологий", Москва, ВМиК МГУ, 2001. http://www.cs.inf.ethz.ch/~zueff/Presentations/Moscow%202001-6-1.pdf

87. Зуев Е.А. *Нестандартные аспекты применения языка XML.* Всероссийская конференция "Теоретические и прикладные проблемы информационных технологий", Москва, ВМиК МГУ, 2001. http://www.cs.inf.ethz.ch/~zueff/Presentations/Moscow%202001-6-2.pdf

88. P.W.Sebesta *Concepts of Programming Languages*. Fifth edition. Addison-Wesley, 1998. Имеется русский перевод:
Р.У. Себеста *Основные концепции языков программирования*. Пятое издание, Вильямс, 2001.

89. Gosling J.B., B.Joy, G. Steel *The Java Language Specification* Addison-Wesley, 1996.

90. Кротов А.Н. *Реализация механизмов времени выполнения языка C++ в компиляторе переднего плана*. Известия ВУЗов. Приборостроение, том 40, 1997, №3, С. 22-28