Реализация объектно-ориентированных языков программирования, гарантирующая межмодульную защиту

В.Ю. Волконский, В.Г. Тихонов, Е.А. Эльцин, П.Г. Матвеев

Введение

Целью межмодульной защиты является обеспечение безопасного взаимодействия модулей в рамках программной системы. Исполнение некорректного или ошибочного кода, случайно или преднамеренно внесенного в состав одного из модулей, не должно оказывать непредусмотренное воздействие на состояние других модулей и тем более на систему в целом.

Понятно, что невозможно обеспечить безупречную работу произвольно взятого модуля, другими словами, выявить и устранить все допущенные при его разработке ошибки. Можно лишь организовать межмодульное взаимодействие таким образом, чтобы исключить нарушения в работе модуля, вызванные внешними причинами, а именно, несанкционированным вмешательством других модулей. Для этого достаточно, чтобы все обращения к данному модулю осуществлялись только с помощью предписанных разработчиками модуля средств, то есть через интерфейс. Взаимодействие в обход интерфейса означает потенциальное нарушение внутренней логики модуля и его непредсказуемое поведение в дальнейшем.

В языках программирования высокого уровня правильное взаимодействие программных объектов, в том числе модулей, обеспечивается системой типов, механизмами областей действия и режимов доступа. Однако без надлежащих проверок во время исполнения программы эти механизмы не обеспечивают надежной защиты, например, в случае использования операций прямого доступа в память или передачи управления по динамически вычисляемым адресам. В данной статье мы описываем реализацию, которая базируется на аппаратной поддержке проверок в архитектуре Эльбрус-2000 [1,2,3].

В работе [4] была описана реализация межмодульной защиты для языка C [5], на основе которой можно построить реализации и для других процедурных языков программирования высокого уровня. Целью данной статьи является распространение принципов межмодульной защиты на языковые конструкции для поддержки объектно-ориентированного программирования языка C++ [6]. Отметим, что сейчас речь не идет о полной реализации межмодульной защиты для языка C++, так как некоторые существенные механизмы этого языка, например, исключительные ситуации, здесь не рассматриваются и являются темой отдельных работ.

В главе 1 кратко излагаются общие принципы, по которым строится межмодульная защита, и вводятся необходимые понятия. В главе 2 описывается каким образом межмодульная защита должна работать для объектов классов, а в главе 3 разбираются потенциально опасные операции над ними. Глава 4 содержит описание программно-аппаратной реализации принципов защищённого объектно-ориентированного программирования, сделанной в рамках проекта «Эльбрус-2000». Использование описанной в статье технологии для компиляции реальных программ и полученные при этом результаты обсуждаются в главах 5 и заключении.

1. Понятие межмодульной защиты

Для наглядности будем считать, что модуль представляет собой отдельную единицу компиляции с точки зрения языка С/С++.

- Все модули размещаются и взаимодействуют в едином виртуальном пространстве. Технологию взаимодействия программ из разных виртуальных пространств мы не рассматриваем.
- Все программные объекты модуля можно разделить на три группы: внутренние для модуля, экспортируемые в другие модули и импортируемые из других модулей. К этим объектам относятся переменные, функции, метки и прочие точки передачи управления.
- При вызове функции одного модуля из другого модуля действуют обычные правила передачи параметров и возврата значения, принятые в языках высокого уровня, включая передачу переменного числа параметров.
- Разрешается передавать в качестве параметров и возвращать из функции ссылки на внутренние объекты модуля.

Рассмотрим множество всех экспортируемых программных объектов модуля и множество тех его внутренних объектов, ссылки на которые каким-либо образом передаются в другие модули. Это множество назовем интерфейсом модуля. Цель межмодульной защиты состоит в том, чтобы гарантировать, что объекты, не входящие в интерфейс модуля, было бы невозможно прочитать или модифицировать из других модулей.

Для того чтобы модуль, имеющий доступ к интерфейсным объектам другого модуля, не мог через них получить доступ к его внутренним объектам, необходимо обеспечить контроль выполнения ряда условий.

- ***Контроль границ данных.*** Так как переменные часто располагаются в непрерывной области памяти, возможность выхода за границу участка, занимаемого одной переменной, означает возможность осуществить доступ к соседним переменным, что является нарушением защиты. В архитектуре Эльбрус-2000 контроль границ данных обеспечивается свойствами ссылок на данные.
- ***Контроль границ кода.*** Для защиты существенно отличать данные от кода. Использование кода другого модуля вместо данных означает возможность динамической модификации этого кода, например, с целью выдачи ссылок на внутренние объекты. Использование данных чужого модуля вместо кода означает возможность интерпретировать произвольные данные как команды машины, что приводит к непредсказуемому поведению модуля. В архитектуре Эльбрус-2000 контроль границ кода обеспечивается свойствами ссылок на код.
- ***Контроль соответствия данных и кода.*** При межмодульных вызовах необходимо синхронно с передачей управления переключать описание данных. В противном случае код одного модуля может быть исполнен в контексте данных другого модуля, что является нарушением защиты. В архитектуре Эльбрус-2000 передача управления и переключение описания данных объединены в одну атомарную операцию.
- ***Контроль чистоты памяти.*** В случае повторного использования участка памяти необходимо обеспечить, чтобы ссылки на объекты, записанные во время первого использования, не могли бы быть прочитаны при последующих его использованиях. Аппаратура Эльбрус-2000 осуществляет инициализацию памяти перед использованием.
- ***Контроль ссылок на уничтоженные объекты.*** Обращение по ссылке к объекту, время жизни которого уже завершилось, может привести к непредсказуемому поведению, а в случае повторного использования памяти - и к нарушению защиты. Аппаратура Эльбрус-2000 в комплексе с защищенной операционной системой синхронизирует время жизни объекта и всех ссылок на этот объект.

Ключевым в реализации защиты является понятие ссылки на объект. Ссылки поддерживают строго регламентированный набор аппаратных операций, во время которых осуществляются соответствующие проверки корректности. Ссылки защищаются тегами, поэтому их модификация в обход разрешенных операций невозможна.

- Для обращения к объектам различной природы используются различные виды ссылок. Вид ссылки определяет множество операций, которые можно применить к описываемому ссылкой объекту. Операции чтения или записи возможны только по ссылке на данные, а передача управления – только по ссылке на код.
- Создание ссылки является атомарной аппаратной операцией. В пределах модуля разрешается создавать ссылки только на те программные объекты, которые описаны в этом модуле. При создании ссылки осуществляется проверка корректности, соответствующая природе объекта. В частности, для кода проверяется, что на указанную точку действительно можно передавать управление.
- При любом обращении к объекту через ссылку осуществляется проверка корректности, соответствующая природе объекта. В частности, для данных осуществляется проверка выхода за границу массива.
- Все обращения к импортированным программным объектам, будь то чтение или запись значения переменной или вызов функции, осуществляются только через соответствующие ссылки.

Реализация защиты основана на избирательной выдаче ссылок. Каждый модуль, и только он, создает и предоставляет другим модулям ссылки на экспортируемые им объекты. Благодаря аппаратному контролю каждой операции с этими ссылками, через них невозможно получить доступ к внутренним объектам модуля.

Применение двух разновидностей ссылок, а именно, ссылок на ограниченные участки данных и ссылок на допустимые точки передачи управления, обеспечивает возможность реализации межмодульной защиты для интерфейсов, состоящих из функций и глобальных переменных, что и было сделано в работе [4].

2. Что мы хотим защищать в объектах классов

Во многих случаях поведение объектов класса мало отличается от поведения обычных переменных неклассового типа. Все виды контроля, описанные в главе 1, применяются и к объектам классов. Таким образом, проблемы, связанные с границами полных объектов, разделением данных и кода, ссылками на уничтоженные объекты, защитой собственно ссылок и т.д., считаются решенными и не рассматриваются. Далее речь пойдет только о свойствах, присущих исключительно объектам классов.

С точки зрения организации защиты объекты класса имеют ряд специфических особенностей. Для контроля над обращениями к членам-данным и функциям-членам класса помимо механизма областей действия используется еще и механизм прав доступа. Публичные члены-данные и функции-члены класса доступны везде в пределах своей области действия, приватные члены-данные и функции-члены класса доступны только в функциях-членах того же класса и в дружественных функциях. Нарушение этих правил, а именно несанкционированное изменение приватных членов-данных или вызов приватных функций-членов класса, может привести к серьезному нарушению в логике программной системы.

Все современные средства разработки обеспечивают контроль прав доступа на этапе компиляции программы. Очевидно, что такой контроль не является достаточным для надежной защиты. Некоторые языковые механизмы, такие как адресная арифметика и преобразование типов, позволяют нарушить права доступа уже на этапе исполнения скомпилированного кода. Наша цель состоит в том, чтобы исключить подобные нарушения.

Другая особенность объектов класса состоит в том, что эти объекты можно создавать и использовать в любом модуле, где имеется объявление класса. С точки зрения защиты это означает, что приватные члены-данные, обращение к которым надлежит контролировать, не локализованы в одном модуле, то есть построить их защиту только на основе уже имеющихся средств невозможно.

Тем не менее, мы постараемся максимально использовать те средства защиты, которые у нас уже есть. Посмотрим, из каких элементов состоит класс.

- Статические функции-члены класса, публичные и приватные.
- Статические члены-данные класса, публичные и приватные.

- Нестатические функции-члены класса, публичные и приватные.
- Нестатические члены-данные класса, публичные и приватные.

При создании объектов класса размножаются только нестатические данные класса, все остальные его элементы остаются в единственном экземпляре. Рассмотрим модуль, который состоит из описаний функций-членов и статических членов-данных некоторого класса. Будем говорить, что класс реализован в этом модуле. В целом, статические члены-данные классов ничем не отличаются от обычных глобальных переменных, а функции-члены классов – от обычных функций, поэтому для их защиты мы можем использовать обычные же механизмы, а именно, экспортировать ссылки только на публичные функции-члены и статические члены-данные класса.

Дробление программы на модули, каждый из которых реализует только один класс, является наиболее надежным, но и наименее эффективным, так как использует максимальное количество проверок времени исполнения. Кроме того, оно может оказаться неоправданно мелким в случае сильного зацепления классов между собой. Более целесообразным представляется собирать в одном модуле реализации сразу нескольких связанных классов и необходимые им сервисные функции и переменные. При этом можно ослабить требования защиты таким образом, что доступ к приватным членам объектов класса разрешается только в том модуле, которому принадлежит реализация этого класса, то есть функциям-членам этого класса и других классов того же модуля. Этот подход позволит разработчикам программной системы самим определять необходимый баланс между надежностью и эффективностью. Сильно зацепленные программные сущности, в том числе классы, собираются в одном модуле и взаимодействуют наиболее эффективно, программные сущности из разных модулей взаимодействуют наиболее надежно.

Итак, имеющиеся средства позволяют построить защиту приватных функций-членов и статических членов-данных класса. Осталось только решить проблему защиты для нестатических приватных членов-данных, грубо говоря, запретить обращение к ним вне рассматриваемого модуля. Для этого нам потребуются специальные ссылки на объекты класса, отличные от ссылок на объекты неклассового типа. Для выяснения свойств таких ссылок в главе 3 мы рассмотрим потенциально опасные с точки зрения нарушения прав доступа операции над объектами классов.

3. Операции над объектами классов

Для того чтобы гарантировать, что приватные нестатические члены-данные класса будут недоступны везде, кроме модуля, реализующего класс, необходимо обеспечить:

- уникальность ссылки на память объекта класса в момент его создания;
- контроль границ членов-данных объекта класса;
- контроль соответствия объекта класса и функции-члена класса;
- контроль преобразований типа объекта класса по дереву наследования.

3.1. Создание объекта класса

Согласно общим принципам, рассмотренным в главе 1, ссылка на объект формируется в том модуле, где этот объект находится. При этом для объектов классового типа сформированная ссылка должна описывать как публичные, так и приватные члены-данные. В случае, когда объект класса создается не в том модуле, где этот класс реализован, приватные члены-данные должны быть защищены от доступа как через вновь созданную ссылку на этот объект, так и через любые другие доступные модулю ссылки. Это означает, что объект класса нельзя располагать в доступных модулю областях памяти или стека, иначе защита будет нарушена. Типичным примером подобного нарушения является операция *placement new* в языке C++, которая позволяет размещать объект класса на заранее выделенной памяти. В случае доступа к этой памяти не через обращение к объекту класса изменить приватные члены-данные этого объекта класса не составляет труда (Рис. 1).

Таким образом, операция выделения памяти для объекта класса и создания ссылки соответствующего вида на эту память необходимо объединить в одно атомарное действие. Создание таких ссылок на память, выделенную каким-либо другим образом, необходимо запретить.

```
class A
{
public:
	// Размещается по смещению 0 от начала объекта.
	int foo;
private:
	// Размещается по смещению sizeof(int) от начала объекта
	int bar;
};

// Выделяем память достаточного размера и создаем на ней объект
// класса.
int mem[sizeof(A)/sizeof(int)];
A* ptr = new(mem) A;

// Обращение к приватному члену объекта класса вызывает ошибку
// компиляции.
ptr->bar = 0;

// Аналогично без ошибки компиляции.
mem[1] = 0;
```

Рис. 1. Нарушение защиты при помощи операции placement new

3.2. Обращение к членам-данным объекта класса

В рамках реализации механизма контроля прав доступа первой проблемой при обращении к членам-данным объекта класса является контроль того факта, что обращение к приватным членам-данным осуществляется только в том модуле, который реализует данный класс. Во-первых, это означает, что ссылка на объект класса должна различать области публичных и приватных членов-данных. Во-вторых, по ссылке на объект класса необходимо уметь вычислять, в каком модуле реализован класс, то есть некоторый условный аналог типовой информации.

Кроме того, как и в общем случае, необходим контроль границ членов-данных объекта класса, чтобы через ссылку на эти данные невозможно было бы получить доступ к смежной области памяти. Подобная ошибочная ситуация часто встречается в программах, интенсивно использующих адресную арифметику.

Обычно объект класса представляет собой непрерывную область памяти, в которой размещаются публичные и приватные члены-данные. Воспользовавшись этим фактом, можно через ссылку на публичный член класса получить доступ к приватному члену класса (Рис. 2).

```
class A
{
public:
	// Размещается по смещению 0 от начала объекта.
	int foo;
private:
	// Размещается по смещению sizeof(int) от начала объекта
	int bar;
};

// Создаем объект класса.
A obj;

// Обращение к приватному члену объекта класса вызывает ошибку
// компиляции.
obj.bar = 0;

// Аналогично без ошибки компиляции.
*(&obj.foo + 1) = 0;
```

Рис. 2. Нарушение защиты при обращении по смещению от публичного члена класса

Таким образом, необходимо контролировать границы каждого публичного члена класса, точнее, границы каждой непрерывной области памяти, в которой располагаются публичные члены-данные одного объекта класса. Это относится как к непосредственному обращению в память объекта, так и к формированию ссылок на нее, то есть при взятии адреса публичного члена класса полученная ссылка должна содержать соответствующие границы.

3.3. Вызов функции-члена класса

В случае, когда один модуль реализует несколько классов, может возникнуть следующая специфическая ситуация. Можно вызвать функцию-член одного класса для объекта другого класса, где оба эти класса реализованы в одном модуле. Если в любой функции в пределах модуля разрешить обращения к приватным членам-данным объектов любых классов, реализованных в модуле, то этот трюк при условии совпадения смещений позволит сделать непредусмотренное логикой программы изменение (Рис. 3).

```
class A
{
private:
	// Размещается по смещению 0 от начала объекта.
	int foo;

public:
	// Обращается к приватному члену объекта по смещению 0.
	void set_foo(int value) { foo = value; }
};

class B
{
private:
	// Размещается по смещению 0 от начала объекта.
	int bar;
};

// Создаем объект класса B.
B obj;

// Обращение к приватному члену объекта класса вызывает ошибку
// компиляции.
obj.bar = 0;

// Аналогично без ошибки компиляции.
(*(A*)(void*)&obj).set_foo(0);
```

Рис. 3. Нарушение защиты при подмене объекта this в функции-члене класса

Так как подобная подмена может быть осуществлена вне модуля, реализующего рассматриваемые классы, необходимо обеспечить контроль типа объекта класса при вызове его функции-члена. Иначе эту задачу можно представить как задачу обеспечения корректности параметра *this* при межмодульных вызовах.

Обратим внимание, что для решения поставленной задачи необходимо уметь различать объекты разных классов. Это означает, что по ссылке на объект класса необходимо уметь вычислять некоторую типовую информацию, более подробную, чем требовалось в предыдущем разделе.

3.4. Преобразование типа объекта класса по дереву наследования

Различают два типа преобразования по дереву наследования: преобразование к производному классу (*widening*) и преобразование к базовому классу (*narrowing*).

Корректность преобразования к производному классу можно проконтролировать только на этапе исполнения программы. Для этого в современных языках программирования существуют

специальные механизмы, например, *dynamic_cast* в языке C++. Работоспособность этих механизмов обычно обеспечивается программными средствами, в том числе ссылками на полный объект из каждого подобъекта и полноценной типовой информацией, то есть специальным модулем библиотеки поддержки языка. Перенос подобных механизмов на более низкий уровень ведет к сильному усложнению необходимых средств и не является целесообразным. Программная же реализация приведения к производному классу не является предметом рассмотрения в данной статье.

Преобразование к базовому классу является одной из наиболее интенсивно используемых операций над объектом класса. На низком уровне результатом этой операции является смещение адреса начала объекта класса и изменение смещений и размеров областей публичных и приватных членов-данных. Другими словами, на низком уровне преобразование задается некоторым набором смещений и размеров, который описывает изменение структуры объекта. Этот набор может быть индивидуальным для каждой пары производный класс/базовый класс. Очевидно, что применение несоответствующего набора может привести к попаданию приватных членов-данных в публичную область, то есть к нарушению защиты.

Таким образом, необходимо контролировать соответствие типа исходного объекта класса и набора чисел, описывающего преобразование. Это означает, что описание преобразований, во-первых, должно формироваться и защищаться от изменений так же строго, как ссылки на объекты классов, и, во-вторых, должны включать информацию о преобразуемых классах, то есть типовую информацию, аналогичную упомянутой в предыдущем разделе.

4. Программно-аппаратная реализация

Предыдущие главы описывали теоретические проблемы защищённой работы с классами, а также предлагали некоторые теоретические решения этих проблем. В данной главе описывается одно из аппаратно-программных решений такой работы, реализованное в рамках создания системы поддержки защищенного программирования для процессора "Эльбрус-2000". Здесь будет подробно рассмотрена проблема защищенного размещения данных объектов классов, проблемы представления и размещения реальных иерархий классов языка C++, а также преобразований, допустимых с точки зрения языка в рамках этой иерархии.

4.1. Представление объектов

В ходе рассмотрения категорий данных объекта, требующих защиты, были выделены три области: публичная область, публичная область только для чтения и, наконец, приватная область. Публичная область доступна по записи и чтению методами любого другого класса, публичная область только для чтения может изменяться только методами самого объекта, однако читать её могут все. Доступ к приватной области возможен только методам самого объекта. Полное описание подобных областей (дескриптор объекта) требует трёх дескрипторов, регламентирующих вышеописанные методы доступа. Полное вхождение всех дескрипторов в описание объекта сделало бы его слишком большим. Стоит, однако, заметить, что области объекта в большинстве случаев расположены компактно и имеют небольшой размер. Это позволяет ограничить их максимальный размер и расстояние между ними. В рассматриваемой реализации все области имеют одинаковый максимальный размер 4 Кбайта и максимальное расстояние между областями одного объекта также равно 4 Кбайтам. Для объектов, имеющих больший размер областей, в динамической памяти выделяется соответствующий участок памяти, и его дескриптор прописывается в необходимую область объекта.

Кроме описания областей объекта необходимо некоторым образом привязать сам объект к его методам. Поскольку только они могут иметь доступ к приватной области объекта и имеют право записывать в его публичную область только для чтения (в дальнейшем закрытые области). Роль такого связующего звена играет некоторый номер типа, который однозначно идентифицирует тип внутри приложения, так же помещающийся в его описание. Любая функция приложения, которая может установить во внешнем окружении эквивалентный номер типа, получит доступ к закрытым областям. Остальные же функции должны довольствоваться только открытыми областями. Таким образом, каждому классу приложения сопоставляется свой номер.

Каждому модулю присваивается свой диапазон доступных номеров типов, любая функция модуля может устанавливать в текущем контексте исполнения любой номер из этого и только этого диапазона, при попытке установки номера вне его возникает ошибка исполнения (здесь и далее это означает возбуждение аппаратного исключения, передающего управление операционной системе). Таким образом, каждый класс должен принадлежать какому-то модулю и только ему. Здесь принадлежность класса модулю определяется как нахождение всех методов класса в этом модуле. Если класс принадлежит нескольким модулям, то он должен иметь несколько номеров типов, что невозможно. Если класс не принадлежит ни к одному модулю, то ни один из существующих модулей не может иметь доступа к закрытым областям, такие классы нам не интересны, поскольку в них нечего защищать.

Номер класса является уникальным в рамках загруженного приложения. С другой стороны, модули могут быть переиспользованы в разных приложениях и, соответственно, иметь разные диапазоны номеров типов после их загрузки (иметь один и тот же диапазон номеров было бы слишком накладно). Как следствие, во время загрузки модуля необходимо настроить модуль на использование того или иного диапазона номеров типов. Для облегчения данной задачи некоторые из конструкций и объектов имеют возможность работы с локальным номером типа, из которого можно получить обычный (глобальный) номер типа путём прибавления начального значения диапазона типов, присвоенных модулю операционной системой. Использование локальных номеров избавляет от необходимости настройки модуля во время загрузки, тем самым, сокращая её время, и разрешает использование его образа сразу несколькими процессами. Очевидно, что использование локальных номеров возможно только для конструкций, которые не могут пересечь границы модуля, в противном случае необходимо его преобразование в глобальный номер. В нашем случае такими конструкциями являются методы классов, принадлежащих модулю, номер типа, с которым они работают, можно представить локально внутри модуля. На Рис. 4 показано отношение между локальными и глобальными номерами типов.

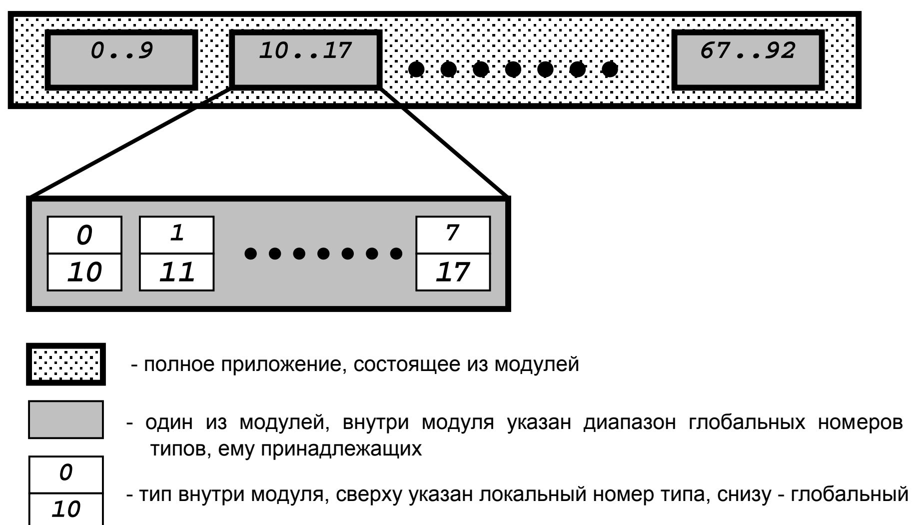


Рис. 4. Локальные и глобальные номера типов

Обозначим все необходимые для дескриптора объекта компоненты:

- *глобальный номер типа объекта* - необходим для определения связи между объектом и методами его класса;
- *дескриптор публичной области* - определяет область объекта, доступную как для чтения, так и для изменения любой функцией любого модуля;
- *дескриптор публичной области только для чтения* - определяет область объекта, доступную для чтения любой функцией любого модуля и для изменения только методами класса;
- *дескриптор приватной области* - определяет область объекта, доступную для чтения и записи только методами класса.

Наличие защищённых областей делает невозможным размещение самого объекта в областях, предоставляющих доступ только через дескриптор объекта. В частности, к таким областям относятся регистры и область переменных модуля, поскольку доступ к ним может быть осуществлён в обход дескриптора. Таким образом, объект может быть размещён только как автоматическая переменная, либо в динамической памяти, т.е. объект всегда размещается в памяти. Для доступа к каждой области, обозначенной дескриптором объекта, используются свои аппаратные операции, которые отличаются наличием разных проверок возможности доступа:

- *чтение и запись публичной области* - осуществляется проверка на превышение размеров области;
- *чтение публичной области только для чтения* - осуществляется проверка на превышение размеров области;
- *запись публичной области только для чтения* - кроме проверки на превышение размеров области, осуществляется проверка на совпадение номеров типов объекта и текущего метода;
- *чтение и запись приватной области* - осуществляется проверка на превышение размеров области, кроме этого проверяются на совпадение номера типов объекта и текущего метода.

При нарушении одного из вышеприведённых условий генерируется аппаратное исключение, и управление передаётся операционной системе.

4.2. Создание объектов

Перед использованием необходимо, прежде всего, создать сам объект. В предыдущей главе было обозначено, что объект может быть расположен на стеке или в динамической памяти. В рассматриваемой реализации для создания объекта на стеке используется специальная аппаратная операция, для создания же его в динамической памяти существует специальный системный вызов операционной системы. И для первого, и для второго случая в качестве параметра передаётся специальный шаблон[1], содержащий все необходимые данные для создания объекта – шаблон типа. Данный шаблон также защищён тегами от несанкционированного изменения пользовательским приложением. Очевидно, что подобный шаблон должен специфицировать размеры и местоположение всех областей объекта, а также глобальный номер типа создаваемого объекта. Используется глобальный номер, поскольку создание нового объекта возможно и вне пределов модуля, к которому принадлежит его тип. Кроме всего вышеперечисленного, шаблон типа также кодирует размер полного объекта, который реально может представлять несколько классов. Данное свойство обеспечивается языковыми механизмами наследования и включения.

В статье [4] описано аппаратно-программное решение, запрещающее переиспользование стековой памяти, которое делает возможным несанкционированный доступ к закрытым областям объекта. Для дескрипторов объектов было использовано точно такое же решение. В каждый дескриптор, полученный операцией создания объекта на стеке, прописывается уровень вложенности процедурного фрейма, в котором находится сам объект. Далее, при попытке сохранения дескриптора в глобальной памяти, либо во фрейме стека с меньшим уровнем вложенности, генерируется аппаратное исключение. Таким образом, дескриптор не может быть сохранён и переиспользован после выхода из функции, его породившей.

Размещение объектов в динамической памяти также основывается на методе, описанном в предыдущей статье. При выделении объекта выделяется память, которая ещё никогда не была каким-либо образом распределена и возвращена пользовательскому приложению. Далее, по исчерпанию возможной памяти, либо параллельно в процессе её выделения, производится запуск специальной программы сборки мусора, которая производит перенастройку указателей (ещё одно преимущество тегирования данных заключается в возможности однозначной идентификации указателей среди данных приложения) и компактировку использованной памяти.

[1] В C++ под словом шаблон подразумевается некоторая параметризованная конструкция языка. В нашем случае шаблон есть аппаратный тип данных, использующийся в операциях процессора и системных вызовах, если не указано обратное

Особую проблему представляет создание массивов объектов. Прежде всего опишем операции, которые могут быть применены к массивам объектов по стандарту языка C++ [6]:

- операции создания и удаления массивов;
- операции доступа к любому из элементов массива;
- операция получения указателя на любой из элементов массива;
- операция получения указателя за последний элемент массива (очевидно, данное требование продиктовано необходимостью эффективной реализации контейнеров библиотеки языка).

Для массива объектов избрано специальное программное представление, которое строится при помощи системного вызова защищённой операционной системы. Построение осуществляется по следующему алгоритму:

- строится обычный массив дескрипторов объектов всех элементов массива плюс последний мнимый элемент;
- в динамической памяти выделяется каждый элемент массива, включая мнимый, и их дескрипторы прописываются в уже выделенный массив дескрипторов по индексу элемента;
- при выделении элементов массива приватная часть объектов расширяется на размер дескриптора массива и в это место прописывается дескриптор массива на место в массиве дескрипторов объектов, где расположена ссылка на текущий элемент;
- пользовательскому приложению возвращается дескриптор первого объекта массива объектов.

Очевидно, что для алгоритма, приведённого выше, требуется два параметра: шаблон для элементов массива и количество элементов в массиве. Выше была особо отмечена роль последнего элемента массива, однако данная ситуация специфична для конкретного языка и в других может быть не нужна. Исходя из этих соображений, в системный вызов передаётся количество элементов уже с учётом мнимого элемента.

Описанный выше алгоритм вызывает процедуру выделения памяти под каждый элемент массива, однако, на самом деле, этого не нужно. Допустимо первоначально выделить область динамической памяти, достаточную для размещения объектов и всех структур его поддержки, а далее просто разметить её необходимым образом, тем самым сэкономив время на отдельных вызовах выделения объектов. Дескриптор обратного указателя строится следующим образом: базовый адрес дескриптора совпадает с оным базовым адресом массива дескрипторов объектов, размер равен размеру для этого же массива, а индекс есть смещение дескриптора соответствующего объекта. Такое построение дескриптора позволяет хранить не только указатель на массив дескрипторов объектов, но и место самого объекта в массиве. Все дескрипторы обратных указателей, размещенные внутри объектов, переводятся в состояние только для чтения. Это предотвращает несанкционированное изменение служебных структур. На Рис. 5 изображено схематичное представление структуры, порождаемой при создании массива объектов.

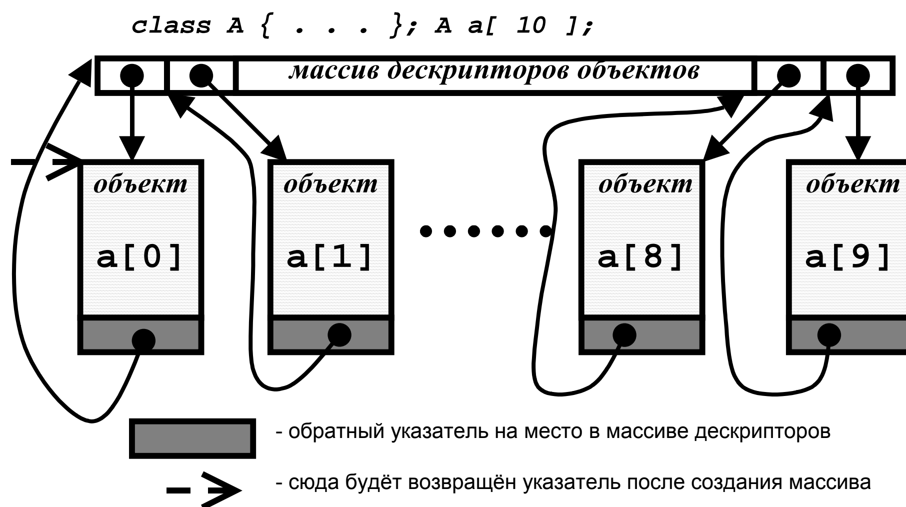


Рис 5. Структура массива объектов

Дальнейшее использование созданного подобным образом массива в операциях достаточно прозрачно:

- для доступа к любому элементу массива достаточно взять указатель на дескриптор объекта текущего элемента и передвинуть его в ту или иную сторону, далее взять значение по полученному адресу;
- тем же самым образом можно получить адрес любого элемента массива.

Вышеприведённая организация гарантирует также защиту от несанкционированного доступа. Для поддержки доступа к элементам массива извне класса компилятором генерируется специальный метод класса для чтения указателя из приватной области.

Необходимо отметить, что обратный указатель на массив дескрипторов всё же находится внутри приватной области объекта, которая доступна по записи из методов класса объекта, следовательно, она может быть ими изменена. Однако альтернативой предложенной реализации может быть только полный перевод всей работы с массивами на операционную систему, что повлечёт за собой сильное снижение эффективности.

4.3. Иерархия классов объекта

В объектно-ориентированных языках классы связаны отношениями включения и наследования. Отношение включения подразумевает под собой содержание одного класса другим. Наследование есть более сложное отношение, которое позволяет объекту представлять несколько сущностей одновременно. В принципе, с точки зрения защиты классов, отношения включения и наследования эквивалентны, в дальнейшем такие отношения будут рассматриваться как *подклассирование*. Если класс включается в другой класс или наследуется другим классом, то такой класс называется *подклассом*[2]. C++ предлагает три вида подклассов: приватный (private), защищённый (protected) и публичный (public). Для включения такие подклассы получаются путём включения этих классов в приватную, защищённую или публичную область соответственно. Наследование же формирует такие подклассы путём приватного, защищённого или публичного наследования.

При рассмотрении объекта в реализации выделено три важных области для защиты: публичная, публичная только для чтения и приватная. При построении отношений между классами объекта имеет смысл продолжить подобное разделение за исключением, может быть, публичной области только для чтения, поскольку ценность её в данном контексте весьма сомнительна.

В самом деле, в C++ для подклассов публичное отношение только для чтения возможно только для публичных константных членов-объектов. Однако, такое преобразование весьма сложно, поскольку оно подразумевает не только изменение областей, но и формирование константного указателя. Архитектура «Эльбрус-2000» не предполагает такой операции. Таким образом, в поле нашего рассмотрения остаются только два вида отношений: публичное и приватное.

В языках объектно-ориентированного программирования объект представляет собой конгломерат классов, получающийся путём применения отношений наследования и включения. Операции этих языков программирования предоставляют не только доступ к областям классов, но и вводят операции выделения подклассов из классов. Таким образом, для эффективной реализации необходимо иметь операции по приведению объекта из одного класса в другой, не нарушающие защиту. В данном месте защита понимается как невозможность несанкционированного доступа к областям одного класса методами другого класса, даже если два класса содержатся в одном объекте.

Поскольку объект класса состоит из трёх уже упоминавшихся областей, приведение объекта из одного класса в другой в нашем случае является перемещением границ и размеров областей объекта. Архитектура «Эльбрус-2000» поддерживает такое преобразование на уровне аппаратуры: вводится операция приведения дескриптора объекта одного класса в дескриптор объекта другого класса с применением специального шаблона. Шаблон содержит смещение и размер областей нового дескриптора объекта, описывающего уже другой класс, представляющийся объектом. Смещения даны относительно одноименных областей старого типа дескриптора объекта. Таким образом, шаблон полностью кодирует приведение объекта из одного класса в дру-

---

[2] В литературе по объектно-ориентированному подходу подклассом обычно называется класс – наследник другого класса. В нашем случае всё наоборот: подклассом называется часть целого класса.

гой. Существуют две потенциальные опасности применения операции с подобным шаблоном. Пользователь может сам сформировать шаблон и получить доступ к закрытым областям объекта. От этого шаблон защищает уже описанный механизм тегирования, теги могут быть созданы только операционной системой, при изменении шаблона пользователем теги будут изменены, что сделает невозможным применение шаблона в операции приведения типа объекта. Другой опасностью является применение шаблона приведения к не «своему» дескриптору, т.е. к дескриптору объекта другого класса, нежели предполагаемый исходный класс. Защиту от подобной операции выполняют глобальные номера исходного и целевого типов, которые присутствуют в шаблоне. Во время выполнения операции приведения типа осуществляется проверка номеров исходных типов дескриптора и шаблона; при несовпадении возникает ошибка исполнения. После выполнения этой операции новому дескриптору будет присвоен целевой номер типа из шаблона. Таким образом, приведение типа жестко зафиксировано видом шаблона: пользователь не может применить «чужой» шаблон к дескриптору объекта и не может сам сформировать шаблон. Операционная система формирует все шаблоны и может проверить их корректность.

Подведём итог содержанию шаблона приведения типа:

- *перемещение и размер публичной области* – необходимо для указания нового местоположения публичной области;
- *перемещение и размер публичной области только для чтения* – необходимо для указания нового местоположения публичной области только для чтения;
- *перемещение и размер приватной области* – необходимо для указания нового местоположения приватной области;
- *глобальный номер исходного типа* – необходим для верификации аргументов операции приведения типа, перед приведением номера типов в дескрипторе объекта и номер исходного типа сравниваются, и, при несовпадении, генерируется аппаратное исключение;
- *глобальный номер целевого типа* – устанавливается после выполнения приведения объекта к новому классу.

В некоторых языках программирования возможно приведение типа объекта как в сторону подкласса, так и в обратную сторону. Очевидно, что, руководствуясь вышеприведённой операцией приведения, возможна реализация приведения от подкласса к классу. Такая ситуация может нарушить защиту памяти других объектов, поскольку трансформируемый объект может не представлять класс, в который осуществляет трансформирование. Например, при создании объекта использовался шаблон класса А, далее от класса А был пронаследован класс Б. В результате в приложении может появиться шаблон преобразования класса А в Б. Однако ранее созданный объект представляет только класс А. Таким образом, после преобразования дескриптор объекта будет описывать область памяти, не принадлежащую полному объекту, что есть нарушение защиты.

Одним из решений описанной выше проблемы является запрет на использование шаблонов приведения от подклассов к классам. Это решение усложняет реализацию некоторых операций объектно-ориентированных языков программирования, однако подобное преобразование статистически малозначимо и может быть либо опущено при реализации, либо реализовано неэффективным образом. Одной из реализаций, позволяющей производить приведение от подкласса к классу, является размещение дескриптора полного объекта (most-derived) в публичной области только для чтения каждого класса объекта. Для выполнения операции «расширения» (widening) этот дескриптор надо привести к необходимому классу, который в свою очередь есть подкласс класса полного объекта, либо сам класс полного объекта.

Другим возможным решением является сохранение базового адреса объекта в дескрипторе объекта в процессе приведения. Однако в архитектуре «Эльбрус - 2000» нет смещения публичной области (из-за нехватки места в дескрипторе), вместо этого предполагается, что публичная область начинается сразу с базового адреса дескриптора, а операция приведения типа добавляет смещение к самому базовому адресу, причём данное смещение не может быть меньше нуля. Таким образом, приведение публичной области от подкласса к классу через аппаратный шаблон приведения невозможно.

Альтернативная реализация. *В принципе, можно обойтись без шаблонов приведения типа, реализуя отношения наследования и включения через указатели на базовые и включаемые классы, размещённые в соответствующих областях объекта. Однако, данная схема достаточно неэффективна. Создание таких объектов требует введения специальных процедур.*

Кроме этого, эффективность приведения от одного класса к другому линейно зависит от глубины отношения. Ещё одним доводом против подобной реализации является высокая трудоемкость создания подобных объектов на локальном стеке. В архитектуре «Эльбрус-2000» объекты, созданные во вложенных процедурах, не могут быть переданы процедурам, которые их вызвали, что предполагает необходимость непосредственной подстановки соответствующего кода в места создания объектов. В принципе, можно отказаться от создания локальных объектов в пользу объектов в глобальной памяти, однако подобное решение может оказаться слишком ресурсоёмким, либо потребует нетривиальных решений со стороны операционной системы (например, создания специального стека объектов).

4.4. Размещение областей классов

В разделе 4.2 и 4.3 описаны способы создания объектов и возможности по преобразованию классов, представляемых объектом. Для возможности правильного применения подобных операций классы объекта должны быть размещены в памяти подобающим образом. Главные требования, налагаемые на размещение областей архитектурой «Эльбрус - 2000», приведены далее.

- Одноимённые области классов объекта должны располагаться друг за другом. В самом деле, шаблон приведения типов позволяет преобразовывать друг в друга только одноимённые области, что исключает их перемешивание.
- Одноименные области классов, связанные отношением подклассов, должны находиться на одинаковом расстоянии друг от друга вне зависимости от объектов, в которых они находятся. Иерархия классов может быть представлена в объектах, имеющих разный полный тип, однако, будучи преобразованными к одинаковым подклассам, они могут быть обработаны одинаковым образом одним и тем же кодом.
- Выше была приведена схема создания массивов объектов. С одной стороны, схема предполагает вставку дополнительного указателя в приватную область. С другой стороны, смещения между одноименными областями классов, связанных отношением подклассов, должны оставаться неизменными. Далее, в массивах могут участвовать только полные, неприведённые объекты. И, наконец, последнее утверждение: поля класса должны располагаться по одинаковым смещениям внутри области, вне зависимости от метода его получения. Из всех этих утверждений следует, что обратный указатель на массив дескрипторов должен находится в конце приватной области класса, а сама эта область должна находиться в конце всех приватных областей полного объекта. Из этого, а также предыдущего утверждения, следует, что приватные области классов, связанных отношениями подклассов, должны быть расположены по направлению от наименее к наиболее зависимому в сторону увеличения адресов памяти.
- Подкласс может быть как публичным, так и приватным. В последнем случае подкласс может быть выделен и использован только самим классом. Его публичная область не должна пересекаться с публичными областями публичных подклассов. В свою очередь, публичные подклассы могут иметь свои приватные подклассы, публичные области которых также не должны быть доступны извне. Архитектура «Эльбрус - 2000» не поддерживает «прерывистых» областей в дескрипторе объекта. Следствием всего этого является невозможность объединения публичных областей классов в одну протяжённую публичную область. То же можно сказать для публичных областей только для чтения.
- Приватные области не имеют приведённой выше проблемы, поскольку они должны быть доступны только своему классу и, как следствие, не должны объединяться.
- Публичная область класса должна располагаться ближе к началу объекта по отношению к публичной области подкласса. Данное утверждение возникает из следующего. Приведение публичной области есть перемещение базового адреса дескриптора. Приведение с отрицательным смещением запрещено. Приведение от класса к подклассу есть статистически более значимая операция, нежели обратная.

Исходя из всех этих требований, можно предложить следующую схему размещения областей классов объекта. Каждый класс имеет три свои области. Подкласс класса располагает свои области на фиксированном удалении от одноимённых областей класса. Классы, не связанные между собой отношениями подклассирования, располагаются в объемлющем классе как угод-

но. Публичные области классов расположены, начиная от самого внешнего класса. Приватные области расположены в обратном порядке.

Остаётся последняя проблема: защита приватных подклассов. Приватные подклассы должны быть доступны только непосредственным классам и никому другому. Для предложенного размещения такая защита может быть обеспечена только операционной системой. Она должна предоставлять шаблоны приведения классов в приватные подклассы только модулю, содержащему класс. В этом случае доступ к приватному классу из другого модуля будет невозможен. Хочется подчеркнуть, что такая защита действует только на уровне модулей, но не классов. Другой класс данного модуля всё же может получить доступ к приватному подклассу «чужого» класса.

На Рис. 6 приведён пример размещения областей классов объекта при отношении подкласcирования. Рис. 7 изображает приведение класса объекта к его подклассу.

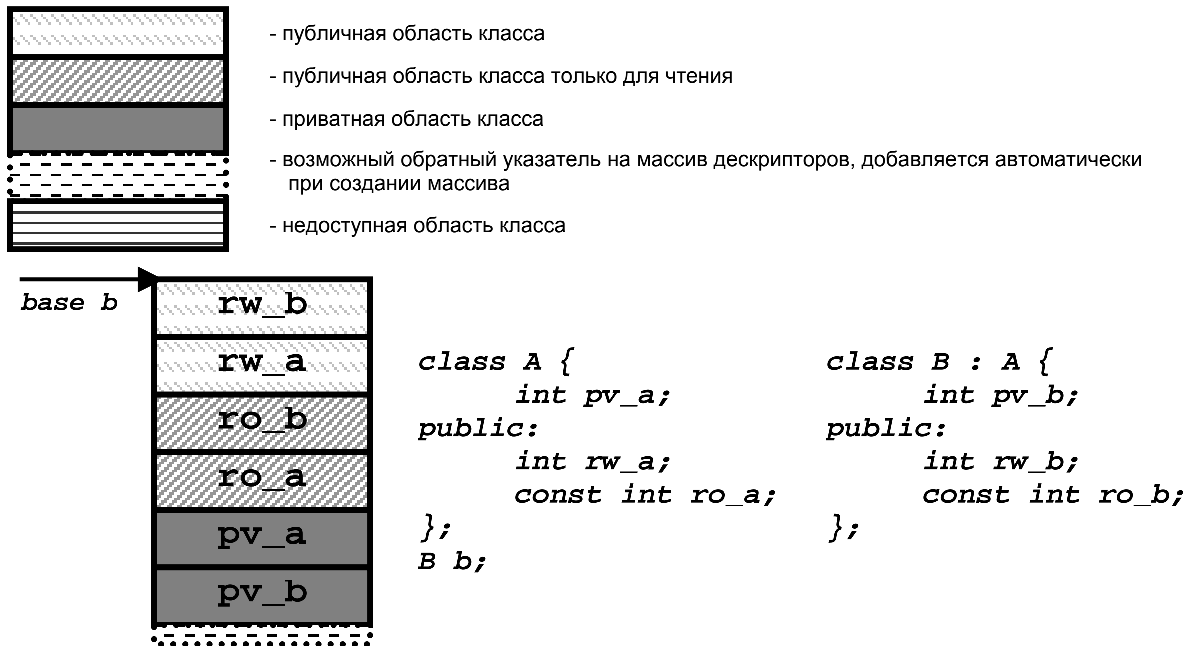


Рис. 6. Размещение классов объекта

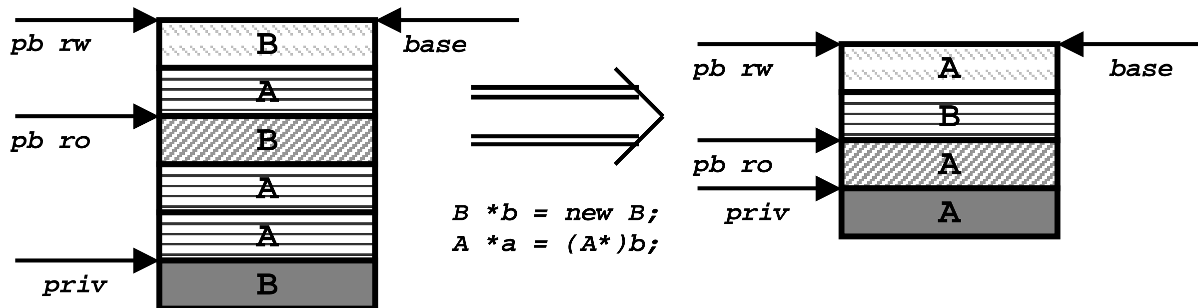


Рис. 7. Приведение к подклассу

4.5. Порождение и верификация шаблонов

При работе пользовательского приложения для создания и преобразования объектов используются специальные шаблоны. Они защищены тегами, и, как следствие, могут быть сформированы только операционной системой, причём необходимо гарантировать отсутствие потенциальной возможности получения несанкционированного доступа к данным объектов. Статическая проверка отсутствия такой возможности сразу после загрузки позволяет увеличить эффективность работы приложения (очевидно, что введение специальных действий на этапе его исполнения замедлит его).

Альтернативная реализация. *В главе 4.3 был кратко рассмотрен альтернативный подход к отображению иерархии классов в объекте. В принципе, он позволяет вообще отказаться от*

выполнения верификации шаблонов, если использовать схему размещения объектов через указатели. В этом случае все отношения между объектами будут представлены через них, и шаблоны приведения будут вообще не нужны. Как следствие, приложение не будет содержать операций, потенциально нарушающих защиту. Однако, как уже говорилось, подобная схема весьма неэффективна.

Прежде всего, перечислим все шаблоны, которые должны порождаться операционной системой:

- *шаблоны создания объекта* – требуются для операций создания объекта как на стеке, так и в динамической памяти;
- *шаблоны приведения объектов* – требуются для выделения подклассов у классов объектов.

Других шаблонов порождаться не должно. Как уже говорилось, все шаблоны защищены специальными тегами от модификации пользователем.

Наиболее очевидной, и на первой взгляд простой, является схема полного формирования образов шаблонов на этапе компиляции приложения. Далее, на фазе загрузки, операционная система должна расставить теги на готовые эти образы. Однако, кроме загрузки, необходимо доказать отсутствие потенциальной возможности нарушения защиты пользовательским приложением, исходя только из той информации, которая предоставлена самими шаблонами. Очевидно, что такая задача весьма нетривиальна по своей сути. При разработке детального алгоритма подобной верификации была выявлена его высокая сложность, из-за необходимости восстановления неявных взаимосвязей между классами. Кроме этого, использование подобной схемы невозможно при использовании схемы размещения областей классов, предложенной выше, поскольку нет возможности отличить приватный подкласс от публичного, тем самым приложение может нарушить защиту, продекларированную исходной программой.

В ходе исследования была предложена несколько более сложная на первый взгляд, но более эффективная схема формирования шаблонов. Вместо реальных шаблонов модуль содержит некоторую информацию о классах, которые ему принадлежат, а также об отношениях этих классов к другим классам, как внутри модуля, так и вне его. После загрузки *всех* модулей операционная система обрабатывает данную информацию и строит все необходимые шаблоны. Таким образом, сложная верификация шаблонов исчезает, остаётся лишь проблема верификации самой информации, что представляется намного менее сложной проблемой.

Рассмотрим представление классов, принадлежащих модулю. Прежде всего, необходимо знать размеры всех трёх областей класса. Кроме размеров, необходимо также предоставлять информацию о выравнивании каждой из областей. Класс может содержать внутри себя другие классы, являющиеся его подклассами. Для описания таких взаимосвязей задаются количество подклассов и массив, их описывающий. Элементами этого массива также являются структуры, содержащие, в свою очередь, ссылку на класс подкласса, а также маску, описывающую отношение данного подкласса к классу, например, является ли он приватным подклассом, является ли базовым и включаемым и т.д. Всё вышеописанное должно кодироваться в файловом представлении модуля и загружаться в память вместе с ним. Ссылки на классы изнутри самих классов можно кодировать при помощи механизма перемещений (relocations).

После загрузки всех модулей в памяти может быть построена полная иерархия классов данного приложения, соответствующая той, что была задана самим языком программирования. При построении такой иерархии необходима проверка на возможные циклические отношения между классами приложения. Например, ссылка на класс в описании подкласса ссылается на сам класс этого подкласса. Возникновение подобных отношений означает попытку намеренного или случайного предоставления ошибочной информации, которая должна пресекаться. К счастью эти нарушения легко выявляются (например, путём пометки уже обработанных классов при построении иерархии). Кроме построения иерархии классов, каждому модулю назначается диапазон номеров типов, ему принадлежащих, а также каждому классу модуля присваивается свой глобальный номер. Построенная иерархия используется для создания аппаратных шаблонов на этапе загрузки программы. Кроме этого, она может быть использована для реализации некоторых операций системной поддержки, например, для ещё одной реализации операции приведения от подкласса к классу. Пример одной из иерархий дан на Рис. 8. Описание необходимых элементов структур, участвующих в иерархии дано на Рис. 9.

Создание *шаблона создания объекта* требует ссылки на класс внутри построенной иерархии. По этой ссылке операционная система обрабатывает сам класс и его подклассы, не только

непосредственные, размещая области объекта. После этого формируется и сбрасывается сам аппаратный шаблон.

Модуль 1 **Модуль 2**

Исходные тексты классов приложения

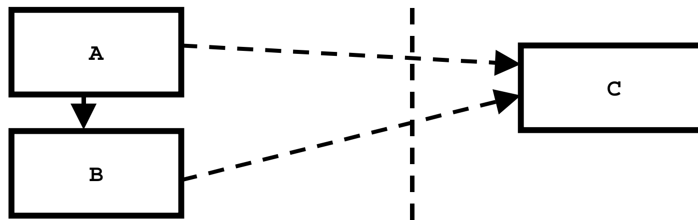
Модуль 1:

```
class A : B        class B
{                  {
private:           private:
      C c;               C c;
};                 };
A a;
```

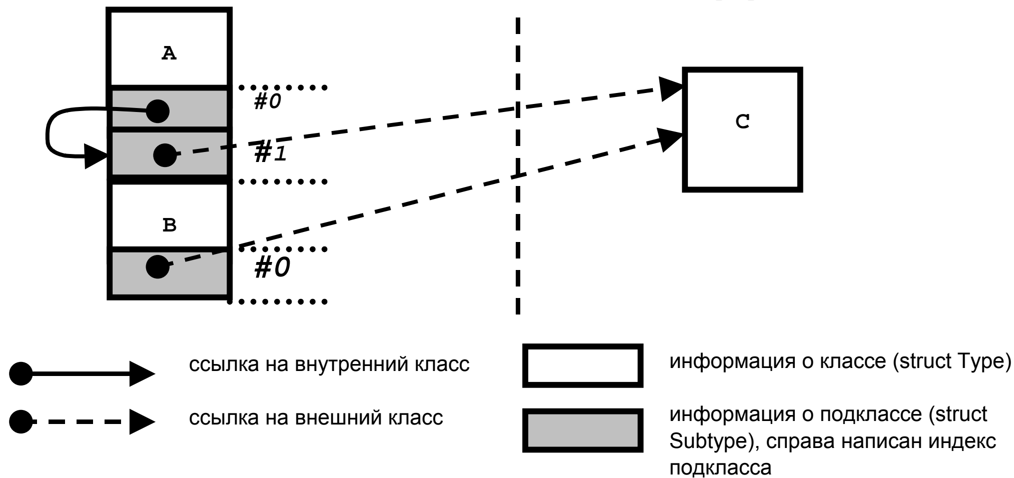
Модуль 2:

```
class C
{
      int pv;
};
```

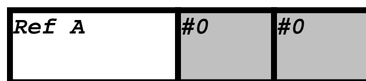
Размещение классов по модулям и их зависимости



Информация о типах, принадлежащих модулям и их взаимосвязям (иерархия)



Информация для построения шаблона приведения в выражении: ((B&)a).c



Информация для построения шаблона приведения в: a.c

Рис. 8. Размещение информации о классах в модулях (иерархия)

Создание *шаблона приведения объекта,* кроме ссылки на класс, который является исходным классом шаблона, требует указание пути приведения внутри иерархии. Путь задаётся массивом индексов, выделяемых подклассов на каждом шаге приведения. Например, для приведения исходного класса к одному из его подклассов, необязательно непосредственному, требуется задание номера этого подкласса внутри класса, далее номера подкласса внутри подкласса и т.д. В процессе прохода по пути приведения вычисляются необходимые смещения для подкласса, а также вычисляется глобальный номер целевого класса. На основе этих данных может быть построен шаблон приведения. На Рис. 10 описана структура с необходимыми элементами для построения шаблона.

```
// Описание подкласса класса.
struct Subtype
{
	// Ссылка на тип подкласса. Конечно же, прямая ссылка
	// на класс невозможна в файловом представлении модуля
	// она кодируется при помощи аппарата перемещений
	const Type *type;
	// Ненулевое значение, если подкласс является приватным.
	unsigned is_private;
};

// Описание самого класса.
struct Type
{
	// Размер публичной области класса.
	unsigned rw_size;
	// Размер публичной области класса только для чтения.
	unsigned ro_size;
	// Размер приватной области класса.
	unsigned pv_size;
	// Выравнивание публичной области.
	unsigned rw_alignment;
	// Выравнивание публичной области только для чтения.
	unsigned ro_alignment;
	// Выравнивание приватной области.
	unsigned pv_alignment;
	// Количество подклассов класса.
	unsigned subtypes_num;
	// Массив подклассов.
	Subtype subtypes[ subtypes_num ];
};
```

Рис. 9. Описание класса и подкласса иерархии

```
// Описание информации для генерации шаблона приведения.
struct Cast
{
	// Исходный тип приведения.
	Type *source_type;
	// Длина пути приведения.
	unsigned subtype_indices_num;
	// Путь приведения.
	unsigned subtype_indices[ subtype_indices_num ];
};
```

Рис. 10. Описание информации для построения шаблона приведения

В процессе построения шаблона необходима верификация предоставляемой модулем информации. Первой из проблем, которые могут возникнуть, является указание значения индекса большего, нежели количество подклассов в текущем классе на пути приведения. Например, класс имеет всего 2-а подкласса, однако индекс на текущем шаге приведения равен 3-м. Данная ситуация говорит об умышленной или случайной несогласованности предоставляемой информации. Вторая проблема более сложна. В ходе прохождения по пути приведения мы можем выйти на приватный подкласс. Доступ к приватным подклассам класса доступен только из самого класса и недоступен извне. В терминах пути приведения это выражается следующими правилами.

- Путь приведения должен содержать не более одного приватного подкласса. Классу может быть доступен его приватный подкласс, но приватный подкласс подкласса ему не доступен.
- Если путь приведения содержит приватный подкласс, то этот подкласс должен быть первым в пути приведения. Здесь можно применить те же рассуждения, что и в предыдущем пункте. Если приватный подкласс не находится в начале пути, то это означает,

что кому-то требуется приватный подкласс не своего подкласса. Такой доступ запрещён языками объектно-ориентированного программирования.

- И, наконец, последнее правило. Шаблон приведения, построенный для пути, имеющего приватный подкласс, может присутствовать только в модуле, которому принадлежит исходный класс. В противном случае класс из одного модуля может получить доступ к приватным данным класса из другого модуля.

Если нарушено одно из этих условий, то информация, предоставленная для создания *шаблона приведения,* является неправильной, что не позволяет надлежащим образом защитить загружаемую программу.

Все вышеприведённые рассуждения предполагают раннее связывание модулей: на этапе загрузки приложения в памяти размещаются все модули и далее производятся разрешения всех ссылок. Однако данное ограничение вполне преодолимо. Например, при переведении всего использования шаблонов на функциональный интерфейс с поздним разрешением ссылок на функции, как это сделано, например, для разделяемых модулей в системе Linux [12] или классов виртуальной машины Java [9].

6. Компиляция существующих программ

В случае, когда программная система изначально проектируется с учетом принципов защиты, разработчики каждого модуля предоставляют список реализованных в нем классов. Тогда при компиляции отдельных файлов не возникает никаких дополнительных проблем, связанных с принадлежностью классов, так как всегда известно из каких файлов составлен модуль, и какие классы ему принадлежат.

Однако в большинстве существующих программных текстов принципы защиты не принимаются во внимание. Реализация класса может быть размазана по нескольким единицам компиляции, и одна единица компиляции может содержать части реализации разных классов. Это обстоятельство затрудняет разделение большой программы на отдельные защищенные модули.

Наилучшим решением проблемы является анализ существующего программного текста, возможно, некоторая его переработка, и создание описания состава модулей из отдельных файлов и принадлежности к ним классов. Конечно, при этом необходимо учитывать внутреннюю логику системы, поэтому наиболее эффективные результаты достигаются при выполнении подобного анализа разработчиками системы анализа вручную. При этом необходимо сохранять баланс между защищенностью и эффективностью. Наиболее оптимальным с точки зрения конечного результата является такое разделение программы на модули, которое обеспечивает максимальную эффективность при достаточном уровне защиты, то есть минимум необходимых проверок времени исполнения. К сожалению, это весьма трудоемкий процесс.

Для отладки и тестирования реализации защиты нам пришлось обработать большое количество объемных исходных текстов на языке C++. Естественно, процесс разбиения программы на отдельные защищенные модули пришлось автоматизировать. Отметим, что нашей целью было получение разбиения, оптимального для тестирования свойств защиты, а не с точки зрения баланса надежности и эффективности. В нашем случае наиболее целесообразным является разбиение программы на как можно большее число модулей, насколько это позволяет зацепление отдельных классов между собой.

На первом этапе определяется принадлежность классов отдельным единицам компиляции. Очевидно, что класс принадлежит единице компиляции, если в ней описана хотя бы одна его функция-член или хотя бы одна его статическая переменная.

Некоторую проблему для определения принадлежности представляют классы без функций-членов и без статических членов-данных, однако, здесь помогает следующее рассуждение. Доступ к приватным членам-данным такого класса разрешен только друзьям этого класса. Если в единице компиляции есть описание хотя бы одного друга класса, то класс принадлежит этой единице компиляции. Если же у класса нет друзей, то выполнить обращение к его приватным членам-данным вообще невозможно. Такой класс может принадлежать любому модулю.

Таким образом, все множество единиц компиляции разбивается на непересекающиеся подмножества из единиц компиляции, реализующих хотя бы один общий класс. На втором этапе единицы компиляции из отдельного подмножества объединяются в модули. В лучшем случае,

когда в каждой единице компиляции оказываются части реализации только одного класса, получается столько модулей, сколько классов в программе. В худшем случае, когда каждая единица компиляции содержит части реализации разных классов, зацепленные между собой через использование общих переменных или функций из этой единицы компиляции, разбиение может привести к организации одного большого модуля, однако без переработки программы решить эту проблему невозможно.

Заключение

Практические результаты

Основным практическим результатом описанной работы является реализация транслятора с языка C++, построенная на основе приведённых выше решений как аппаратных, так и программных. Стоит, однако, отметить, что данная статья не затрагивает всех вопросов, возникших при реализации языка C++. Но, вообще говоря, реализован полный язык. За пределами статьи остались вопросы реализации исключений, поддержки виртуальных механизмов, динамической типовой информации (RTTI), особенности реализации шаблонов и многое другое. Вследствие всего этого, практические результаты, описанные далее, необходимо рассматривать как результаты решения более обширного круга задач.

В процессе работы над компилятором был осуществлён запуск тестового пакета cppvs [7]. Данный пакет представляет собой обширный набор (до 3500) небольших исходных файлов (каждый не более 100 строк), проверяющих полноту реализации языка C++. Пакет прошёл практически полностью, за исключением тестов с прямым или косвенным нарушением защиты или использованием неинициализированных переменных. Такой результат позволяет утверждать, что, в основных чертах, реализация объектно-ориентированных языков на основе предложенного аппаратно - программного решения возможна.

Кроме разработки самого компилятора, была осуществлена раскрутка стандартной библиотеки C++ на основе исходных текстов свободно распространяемой библиотеки STLport [8]. Тесты, поставляемые вместе с самой библиотекой, были запущены практически в полном объёме. Таким образом, было создано полное окружение, необходимое для создания реальных приложений на языке C++. Полученная система программирования была проверена на объемных реальных задачах, в том числе на некоторых C++ - тестах-кандидатах пакета spec-2004, без изменения их исходных текстов.

При разработке компилятора для проверки его работоспособности был разработан прототип защищённой операционной системы. Он не реализует всех функций реальной операционной системы, однако специфика защиты в нём отработана почти полностью. В частности, проверены алгоритмы системных вызовов для поддержки защиты, сборщика мусора (с компактировкой памяти), а также загрузки и верификации приложения. Все запуски тестов и реальных приложений проводились на этом прототипе.

Нужно отметить, что в ходе работы был доработан не только сам компилятор. При помощи него были найдены некоторые ошибки как в тестовых пакетах, так и в реализации библиотеки. Это указывает на жизнеспособность и крайнюю полезность предложенной реализации; необходимость дальнейшего продолжения исследований.

Альтернативные решения

На данный момент известно две успешные разработки, затрагивающие проблемы, описанные в статье. Первой из них является виртуальная машина Java, как она специфицирована фирмой Sun [9], вторая - платформа Microsoft .NET IL (Intermediate Language) [11]. Принципиально эти разработки очень похожи друг на друга.

В основе каждой из этих разработок находится специальная архитектура. Чаще всего эти архитектуры реализуются программно, однако существуют и аппаратные реализации, например, система picoJava [10]. Одна из особенностей этих архитектур заключается в том, что операции обращения к членам классов в них представляются отдельными командами, а не расписываются через доступ по адресу и смещению. На этапе компиляции для каждого класса строится описание всех его полей и методов, по которому команды доступа выполняют все необходимые

проверки, включая проверку режима доступа. Похожим образом выполняется приведение типа для объектов классов.

С нашей точки зрения общим недостатком этих решений является использование специальных, ‘безопасных’ языков программирования, из которых исключена адресная арифметика и другие, трудно контролируемые конструкции. Для виртуальной машины Java используется только язык Java. Для платформы Microsoft .NET IL используются несколько языков программирования, в том числе C# и даже C++, но только в редуцированном (*managed*) виде. Дополнительно оставлена возможность вызова так называемых *native* методов без средств защиты для поддержки функций, трудно реализуемых на языках с упомянутыми ограничениями. Мы же стремимся поддержать традиционные языки программирования в максимально полном объеме, без ограничений для защиты.

Развитие

В данной статье мы построили межмодульную защиту для механизма классов в объектно-ориентированных языках программирования на примере языка C++. Темой дальнейших работ является защищенная реализация других механизмов современных языков программирования. В частности, одна из ближайших статей будет посвящена поддержке исключительных ситуаций.

Литература

1. Б.А.Бабаян. Защищенные информационные системы. – www.elbrus.ru
2. Ф.А.Груздов, Ю.Х.Сахин. Архитектурная поддержка типизации данных. – ‘Информационные технологии и вычислительные системы’. – Москва, ИВВС РАН, 1999
3. Ф.А.Груздов. Архитектурная поддержка защищенных вычислений при реализации объектно-ориентированных языков. – Москва, 2000
4. В.В.Волконский, В.Г.Тихонов, Е.А.Эльцин Реализация языков программирования, гарантирующая межмодульную защиту. – Высокопроизводительные вычислительные системы и микропроцессоры. Сборник научных трудов. Выпуск 2. стр.3-20. – 1999
5. International Standard ISO/IEC 9899 Programming languages – C. – 1990
6. International Standard ISO/IEC 14882 Programming languages – C++. – 1998
7. Ю.В.Баскаков. Принципы построения тестовых комплектов для тестирования конформности компиляторов стандартам языков программирования. – в сб. трудов под ред. В.А.Сухомлина ‘Теоретические и прикладные проблемы ИТ’ – Москва, ВМиК МГУ, 2001
8. STLport. – www.stlport.com
9. Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification. – Addison-Wesley, 1997
10. picoJava II Programmer’s Reference Manual. – Sun Microsystems, Inc., March, 1999
11. Serge Lidin. . NET IL Assembler. – Microsoft Press, 2002
12. John R. Levine. Linkers and Loaders - Morgan Kaufmann, January, 2000