Дмитрий Якушев

«Философия» программирования на языке C++

издание второе, дополненное и исправленное

УДК 004.43
ББК 32.973.202
Я49

Якушев Д. М.
Я49 «Философия» программирования на языке C++. / Д. М. Якушев. — 2-е изд. — М.: Бук-пресс, 2006. — 320 с.
ISBN 5-9643-0028-6

Автором языка C++ является Бьерн Страуструп, сотрудник известной фирмы AT&T. C++ (а точнее, его предшественник, C with classes) был создан под влиянием языка Simula (надо сказать, что этот язык программирования появился еще в 1967 году). Собственно, к тому моменту, когда появился C++, C уже заработал себе популярность; профессиональные программисты уважают его за возможность использовать преимущества конкретной архитектуры, создавая при этом программы на языке относительно высокого уровня.

В настоящее время C++ — один из самых популярных (если не самый популярный) языков программирования. Именно C++ позволяет написать программу с использованием объектно-ориентированных подходов (а программы, которые этого требуют, обычно очень большие) и при этом достаточно «быструю».

Эта книга познакомит читателя с «философией» и основами программирования на языке C++. В книге приводится множество примеров, скомпилированных и проверенных автором.

УДК 004.43
ББК 32.973.202

ISBN 5-9643-0028-6

Содержание

Часть 1. Введение в язык программирования C++

Часть 2. Турбо C++

Часть 3. От теории к практике

Научно-популярное издание

Якушев Дмитрий Михайлович

«ФИЛОСОФИЯ» ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Главный редактор *Б. К. Леонтьев*
Шеф-редактор *А. Г. Бенеташвили*
Оригинал-макет *И. В. Царик*
Художник *О. К. Алехин*
Художественный редактор *М. Л. Мишин*
Технический редактор *К. В. Шапиро*
Корректоры *Л. С. Зимина, К. В. Толкачева*

Подписано в печать 07.05.2006. Формат 60x90/16.
Гарнитура «Ньютон». Бумага офсетная. Печать офсетная.
Печ. л. 20. Тираж 3000.

Часть 1. Введение в язык программирования C++

Глава 1. C++ — язык программирования общего назначения

Автором языка C++ является Бьерн Страуструп, сотрудник известной фирмы AT&T. C++ (а точнее, его предшественник, C with classes) был создан под влиянием Simula (надо сказать, что этот язык программирования появился еще в 1967 году). Собственно, к тому моменту (когда появлялся C++), C уже заработал себе популярность; обычно, его очень уважают за возможность использовать возможности конкретной архитектуры, при этом все еще используя язык относительно высокого уровня. Правда, обычно, именно за это его и не любят. Конечно же, при этом терялись (или извращались) некоторые положения построения программ; например, в C фактически отсутствует возможность создавать модульные программы. Нет, их конечно же создают, но при помощи некоторых «костылей» в виде использования директив препроцессора — такой модульности, как, например в Modula-2 или Ada, в C нет. Кстати сказать, этого нет до сих пор и в C++.

С самого начала подчеркивалось то, что C++ — развитие языка Си, возможно, некоторый его диалект. Об этом говорит тот факт, что первым компилятором (существующим до сих пор) являлся **cfront**, который занимался тем, что переводил исходный текст на C++ в исходный текст на Си. Это мнение бытует до сих пор и, надо сказать, до сих пор оно является небезосновательным; тем не менее, C и C++ — разные языки программирования.

Основное отличие C++, когда он только появлялся, была явная поддержка объектно-ориентированного подхода к программированию. Надо понимать, что программировать с использованием ООП и ООА можно где угодно, даже если инструментарий явно его не поддерживает; в качестве примера можно взять библиотеку пользовательского интер-

фейса GTK+, которая написана на «чистом» C и использованием принципов объектно-ориентированного «дизайна». Введение в язык программирования средств для поддержки ООП означает то, что на стадии компиляции (а не на стадии выполнения программы) будет производится проверка совместимости типов, наличия методов и т.п. В принципе, это достаточно удобно.

Опять же, точно так же как и C не является в чистом виде языком программирования высокого уровня (из-за того, что позволяет выполнить слишком много трюков), C++, строго говоря, не является объектно-ориентированным языком программирования. Мешают этому такие его особенности, как наличие виртуальных функций (потому что при разговоре о полиморфизме никто никогда не оговаривает того, что некоторые методы будут участвовать в нем, а некоторые — нет), присутствие до сих пор функции **main()** и т.п. Кроме того, в C++ нет таких сущностей, как, например, метаклассы (хотя они, наверное, не так уж сильно и нужны) и интерфейсы (вместо них используется множественное наследование). Тем не менее, C++ на текущий момент один из самых популярных (если не самый популярный) язык программирования. Почему? Да потому что все эти «уродства» позволяют в итоге написать программу с использованием объектно-ориентированных подходов (а программы, которые этого требуют, обычно очень большие) и при этом достаточно «быструю». Этому способствует именно наличие виртуальных функций (т.е., то, что программист сам разделяет еще во время проектирования, где ему понадобится полиморфизм, а где будет достаточно объединить некоторые функции в группу по некоторому признаку), обязательное наличие оптимизатора и прочее. Все это позволяет при грамотном использовании все-таки написать работающую программу. Правда, достаточно часто встречаются примеры неграмотного использования, в результате чего выбор C++ для реализации проекта превращается в пытку для программистов и руководства.

На текущий момент на язык программирования C++ существует стандарт, в который включена, помимо прочего, стандартная библиотека шаблонов. О шаблонах разговор особый, формально они тоже являются «костылями» для того, чтобы сделать, так сказать, полиморфизм на стадии компиляции (и в этом качестве очень полезны), а вот библиотека — бесспорно правильное со всех сторон новшество. Наличие, наконец-то, стандартных способов переносимо (имеется в виду, с компилятора на компилятор) отсортировать список (кроме написания всех соответствующих подпрограмм и структур данных самостоятельно) очень сильно облегчило жизнь программиста. Правда, до сих пор очень много людей плохо себе представляют как устроена STL и как ей пользоваться (кроме **std::cin** и **std::cout**).

Важной вехой в развитии программирования явилось создание и широкое распространение языка C++. Этот язык, сохранив средства ставшего общепризнанным стандартом для написания системных и прикладных программ языка C (процедурно-ориентированный язык), ввел в практику программирования возможности нового технологического подхода к разработке программного обеспечения, получившего название «объектно-ориентированное программирование».

Язык программирования **C++** — это **C***, расширенный введением классов, **inline**-функций, перегруженных операций, перегруженных имен функций, константных типов, ссылок, операций управления свободной памятью, проверки параметров функций.

Внедрение в практику написания программ объектно-ориентированной парадигмы дает развитие новых областей информатики, значительное повышение уровня технологичности создаваемых программных средств, сокращение затрат на разработку и сопровождение программ, их повторное использование, вовлечение в процесс расширения интеллектуальных возможностей ЭВМ.

Объектный подход информационного моделирования предметных областей все более успешно применяется в качестве основы для структуризации их информационных отражений и, в частности, баз знаний.

C++ является языком программирования общего назначения. Именно этот язык хорошо известен своей эффективностью, экономичностью, и переносимостью.

Указанные преимущества **C++** обеспечивают хорошее качество разработки почти любого вида программного продукта.

Использование **C++** в качестве инструментального языка позволяет получать быстрые и компактные программы. Во многих случаях программы, написанные на **C++**, сравнимы по скорости с программами, написанными на языке ассемблера.

Перечислим некоторые существенные особенности языка **C++**:

- **C++** обеспечивает полный набор операторов структурного программирования;
- **C++** предлагает необычно большой набор операций;
- Многие операции **C++** соответствуют машинным командам и поэтому допускают прямую трансляцию в машинный код;

- Разнообразие операций позволяет выбирать их различные наборы для минимизации результирующего кода;
- **C++** поддерживает указатели на переменные и функции;
- Указатель на объект программы соответствует машинному адресу этого объекта;
- Посредством разумного использования указателей можно создавать эффективно выполняемые программы, т.к. указатели позволяют ссылаться на объекты тем же самым путем, как это делает ЭВМ;
- **C++** поддерживает арифметику указателей, и тем самым позволяет осуществлять непосредственный доступ и манипуляции с адресами памяти.

Глава 2. Лексика

Есть шесть классов лексем: идентификаторы, ключевые слова, константы, строки, операторы и прочие разделители. Символы пробела, табуляции и новой строки, а также комментарии (собирательно — «белые места»), как описано ниже, игнорируются, за исключением тех случаев, когда они служат разделителями лексем.

Некое пустое место необходимо для разделения идентификаторов, ключевых слов и констант, которые в противном случае окажутся соприкасающимися. Если входной поток разобран на лексемы до данного символа, принимается, что следующая лексема содержит наиболее длинную строку символов из тех, что могут составить лексему.

Комментарии

Символы /* задают начало комментария, заканчивающегося символами */. Комментарии не могут быть вложенными.

Символы // начинают комментарий, который заканчивается в конце строки, на которой они появились.

Идентификаторы (имена)

Идентификатор — последовательность букв и цифр произвольной длины; первый символ обязан быть буквой; подчерк '_' считается за букву; буквы в верхнем и нижнем регистрах являются различными.

Ключевые слова

Следующие идентификаторы зарезервированы для использования в качестве ключевых слов и не могут использоваться иным образом:

- asm
- auto
- break
- case
- char
- class
- const
- continue
- default
- delete
- do
- double
- else
- enum
- extern
- float
- for
- friend
- goto
- if
- inline
- int
- long
- new
- operator
- overload

- public
- register
- return
- short
- sizeof
- static
- struct
- switch
- this
- typedef
- union
- unsigned
- virtual
- void
- while

Идентификаторы **signed** и **volatile** зарезервированы для применения в будущем.

Константы

Есть несколько видов констант. Ниже приводится краткая сводка аппаратных характеристик, которые влияют на их размеры.

Целые константы

Целая константа, состоящая из последовательности цифр, считается восьмеричной, если она начинается с 0 (цифры ноль), и десятичной в противном случае. Цифры 8 и 9 не являются восьмеричными цифрами.

Последовательность цифр, которой предшествует **0x** или **0X**, воспринимается как шестнадцатеричное целое.

В шестнадцатеричные цифры входят буквы от **a** или **A** до **f** или **F**, имеющие значения от 10 до 15.

Десятичная константа, значение которой превышает наибольшее машинное целое со знаком, считается длинной (**long**); восьмеричная и шестнадцатеричная константа, значение которой превышает наиболь-

шее машинное целое со знаком, считается **long**; в остальных случаях целые константы считаются **int**.

Явно заданные длинные константы

Десятичная, восьмеричная или шестнадцатеричная константа, за которой непосредственно стоит **l** (латинская буква «эль») или **L**, считается длинной константой.

Символьные константы

Символьная константа состоит из символа, заключенного в одиночные кавычки (апострофы), как, например, **'x'**. Значением символьной константы является численное значение символа в машинном наборе символов (алфавите).

Символьные константы считаются данными типа **int**. Некоторые неграфические символы, одиночная кавычка **'** и обратная косая \, могут быть представлены в соответствие со следующим списком **escape**-последовательностей:

- символ новой строки NL(LF) \n
- горизонтальная табуляция NT \t
- вертикальная табуляция VT \v
- возврат на шаг BS \b
- возврат каретки CR \r
- перевод формата FF \f
- обратная косая \ \\
- одиночная кавычка (апостроф) ' \'
- набор битов 0ddd \ddd
- набор битов 0xddd \xddd

Escape-последовательность **\ddd** состоит из обратной косой, за которой следуют 1, 2 или 3 восьмеричных цифры, задающие значение требуемого символа. Специальным случаем такой конструкции является **\0** (не следует ни одной цифры), задающая пустой символ **NULL**.

Escape-последовательность **\xddd** состоит из обратной косой, за которой следуют 1, 2 или 3 шестнадцатеричных цифры, задающие значение требуемого символа. Если следующий за обратной косой символ не является одним из перечисленных, то обратная косая игнорируется.

Константы с плавающей точкой

Константа с плавающей точкой состоит из целой части, десятичной точки, мантиссы, **e** или **E** и целого показателя степени (возможно, но не обязательно, со знаком). Целая часть и мантисса обе состоят из последовательности цифр. Целая часть или мантисса (но не обе сразу) может быть опущена; или десятичная точка, или **e (E)** вместе с целым показателем степени (но не обе части одновременно) может быть опущена. Константа с плавающей точкой имеет тип **double**.

Перечислимые константы

Имена, описанные как перечислители, являются константами типа **int**.

Описанные константы

Объект любого типа может быть определен как имеющий постоянное значение во всей области видимости его имени. В случае указателей для достижения этого используется декларатор ***const**; для объектов, не являющихся указателями, используется описатель **const**.

Строки

Строка есть последовательность символов, заключенная в двойные кавычки: **«...»**. Строка имеет тип **массив символов** и класс памяти **static**, она инициализируется заданными символами.

Компилятор располагает в конце каждой строки нулевой (пустой) байт **\0** с тем, чтобы сканирующая строку программа могла найти ее конец.

В строке перед символом двойной кавычки **"** обязательно должен стоять \; кроме того, могут использоваться те же **escape**-последовательности, что были описаны для символьных констант.

И, наконец, символ новой строки может появляться только сразу после \; тогда оба, — \ и символ новой строки, — игнорируются.

Глава 3. Синтаксис

Запись синтаксиса

По синтаксическим правилам записи синтаксические категории выделяются курсивом, а литеральные слова и символы шрифтом постоянной ширины.

Альтернативные категории записываются на разных строках. Необязательный терминальный или нетерминальный символ обозначается нижним индексом «opt», так что **{выражение opt}** указывает на необязательность выражения в фигурных скобках.

Имена и типы

Имя обозначает (денотирует) объект, функцию, тип, значение или метку. Имя вводится в программе описанием. Имя может использоваться только внутри области текста программы, называемой его областью видимости. Имя имеет тип, определяющий его использование.

Объект — это область памяти. Объект имеет класс памяти, определяющий его время жизни. Смысл значения, обнаруженного в объекте, определяется типом имени, использованного для доступа к нему.

Глава 4. Область видимости

Есть четыре вида областей видимости: локальная, файл, программа и класс.

- **Локальная**

Имя, описанное в блоке, локально в этом блоке и может использоваться только в нем после места описания и в охватываемых блоках.

Исключение составляют метки, которые могут использоваться в любом месте функции, в которой они описаны. Имена формальных параметров функции рассматриваются так, как если бы они были описаны в самом внешнем блоке этой функции.

- **Файл**

Имя, описанное вне любого блока или класса, может использоваться в файле, где оно описано, после места описания.

- **Класс**

Имя члена класса локально для его класса и может использоваться только в функции члене этого класса, после примененной к объекту его класса операции, или после примененной к указателю на объект его класса операции **->**.

На статические члены класса и функции члены можно также ссылаться с помощью операции **::** там, где имя их класса находится в области видимости.

Класс, описанный внутри класса, не считается членом, и его имя принадлежит охватывающей области видимости.

Имя может быть скрыто посредством явного описания того же имени в блоке или классе. Имя в блоке или классе может быть скрыто только именем, описанным в охватываемом блоке или классе.

Скрытое нелокальное имя также может использоваться, когда его область видимости указана операцией **::**.

Имя класса, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит **class**, **struct** или **union**. Имя перечисления **enum**, скрытое именем, которое не является именем типа, все равно может использоваться, если перед ним стоит **enum**.

Глава 5. Определения

Описание является определением, за исключением тех случаев, когда оно описывает функции, не задавая тела функции, когда оно содержит спецификатор **extern** (1) и в нем нет инициализатора или тела функции, или когда оно является описанием класса.

Глава 6. Компоновка

Имя в файловой области видимости, не описанное явно как **static**, является общим для каждого файла многофайловой программы. Таковым же является имя функции. О таких именах говорится, что они внешние.

Каждое описание внешнего имени в программе относится к тому же объекту, функции, классу, перечислению или значению перечислителя.

Типы, специфицированные во всех описаниях внешнего имени должны быть идентичны. Может быть больше одного определения типа, перечисления, **inline**-функции или несоставного **const**, при условии, что определения идентичны, появляются в разных файлах и все инициализаторы являются константными выражениями.

Во всех остальных случаях должно быть ровно одно определение для внешнего имени в программе.

Реализация может потребовать, чтобы составное **const**, использованное там, где не встречено никакого определения **const**, должно быть явно описано **extern** и иметь в программе ровно одно определение. Это же ограничение может налагаться на **inline**-функции.

Глава 7. Классы памяти

Есть два описываемых класса памяти:

- **автоматический**
- **статический**.

Автоматические объекты локальны для каждого вызова блока и сбрасываются по выходе из него.

Статические объекты существуют и сохраняют свое значение в течение выполнения всей программы.

Некоторые объекты не связаны с именами и их времена жизни явно управляются операторами **new** и **delete**.

Глава 8. Основные типы

Объекты, описанные как символы (**char**), достаточны для хранения любого элемента машинного набора символов, и если принадлежащий этому набору символ хранится в символьной переменной, то ее значение равно целому коду этого символа.

В настоящий момент имеются целые трех размеров, описываемые как **short int**, **int** и **long int**. Более длинные целые (**long int**) предоставляют не меньше памяти, чем более короткие целые (**short int**), но при реализации или длинные, или короткие, или и те и другие могут стать эквивалентными обычным целым.

«Обычные» целые имеют естественный размер, задаваемый архитектурой центральной машины; остальные размеры делаются такими, чтобы они отвечали специальным потребностям.

Каждое перечисление является набором именованных констант. Свойства **enum** идентичны свойствам **int**. Целые без знака, описываемые как **unsigned**, подчиняются правилам арифметики по модулю **2n**, где **n** — число бит в их представлении.

Числа с плавающей точкой одинарной (**float**) и двойной (**double**) точности в некоторых машинных реализациях могут быть синонимами.

Поскольку объекты перечисленных выше типов вполне можно интерпретировать как числа, мы будем говорить о них как об арифметических типах.

Типы **char**, **int** всех размеров и **enum** будут собирательно называться целыми типами. Типы **float** и **double** будут собирательно называться плавающими типами.

Тип данных **void** (пустой) определяет пустое множество значений. Значение (несуществующее) объекта **void** нельзя использовать никаким образом, не могут применяться ни явное, ни неявное преобразования.

Поскольку пустое выражение обозначает несуществующее значение, такое выражение такое выражение может использоваться только как оператор выражение или как левый операнд в выражении с запятой. Выражение может явно преобразовываться к типу **void**.

Глава 9. Производные типы

Кроме основных арифметических типов концептуально существует бесконечно много производных типов, сконструированных из основных типов следующим образом:

- массивы объектов данного типа;
- функции, получающие аргументы данного типа и возвращающие объекты данного типа;
- указатели на объекты данного типа;
- ссылки на объекты данного типа;
- константы, являющиеся значениями данного типа;
- классы, содержащие последовательность объектов различных типов, множество функций для работы с этими объектами и набор ограничений на доступ к этим объектам и функциям;

- структуры, являющиеся классами без ограничений доступа;
- объединения, являющиеся структурами, которые могут в разное время содержать объекты разных типов.

В целом эти способы конструирования объектов могут применяться рекурсивно.

Объект типа **void*** (указатель на **void**) можно использовать для указания на объекты неизвестного типа.

Глава 10. Объекты и LVALUE (адреса)

Объект есть область памяти; **lvalue** (адрес) есть выражение, ссылающееся на объект. Очевидный пример адресного выражения — имя объекта.

Есть операции, дающие адресные выражения: например, если **E** — выражение типа указатель, то ***E** — адресное выражение, ссылающееся на объект, на который указывает **E**.

Термин «**lvalue**» происходит из выражения присваивания **E1=E2**, в котором левый операнд **E1** должен быть адресным (**value**) выражением.

Ниже при обсуждении каждого оператора указывается, требует ли он адресные операнды и возвращает ли он адресное значение.

Глава 11. Символы и целые

Символ или короткое целое могут использоваться, если может использоваться целое. Во всех случаях значение преобразуется к целому.

Преобразование короткого целого к длинному всегда включает в себя знаковое расширение; целые являются величинами со знаком. Содержат символы знаковый разряд или нет, является машинно-зависимым. Более явный тип **unsigned char** ограничивает изменение значения от 0 до машинно-зависимого максимума.

В машинах, где символы рассматриваются как имеющие знак (знаковые), символы множества кода **ASCII** являются положительными.

Однако, символьная константа, заданная восьмеричной **esc**-последовательностью подвергается знаковому расширению и может стать отрицательным числом; так например, **'\377'** имеет значение **-1**.

Когда длинное целое преобразуется в короткое или в **char**, оно урезается влево; избыточные биты просто теряются.

Float и double

Для выражений **float** могут выполняться действия арифметики с плавающей точкой одинарной точности. Преобразования между числами одинарной и двойной точности выполняются настолько математически корректно, насколько позволяет аппаратура.

Плавающие и целые

Преобразования плавающих значений в интегральный тип имеет склонность быть машинно-зависимым. В частности, направление усечения отрицательных чисел различается от машины к машине. Если предоставляемого пространства для значения не хватает, то результат не определен.

Преобразование интегрального значения в плавающий тип выполняются хорошо. При нехватке в аппаратной реализации требуемых бит, возникает некоторая потеря точности.

Указатели и целые

Выражение целого типа можно прибавить к указателю или вычесть из него; в таком случае первый преобразуется, как указывается при обсуждении операции сложения.

Можно производить вычитание над двумя указателями на объекты одного типа; в этом случае результат преобразуется к типу **int** или **long** в зависимости от машины.

Unsigned

Всегда при сочетании целого без знака и обычного целого обычное целое преобразуется к типу **unsigned** и результат имеет тип **unsigned**.

Значением является наименьшее целое без знака, равное целому со знаком (**mod 2****(размер слова)) (т.е. по модулю **2****(размер слова)). В дополнительном двоичном представлении это преобразование является пустым, и никаких реальных изменений в двоичном представлении не происходит.

При преобразовании целого без знака в длинное значение результата численно совпадает со значением целого без знака. Таким образом, преобразование сводится к дополнению нулями слева.

Глава 12. Преобразования

Арифметические преобразования

Большое количество операций вызывают преобразования и дают тип результата одинаковым образом. Этот стереотип будет называться **«обычным арифметическим преобразованием»**.

Во-первых, любые операнды типа **char**, **unsigned char** или **short** преобразуются к типу **int**.

Далее, если один из операндов имеет тип **double**, то другой преобразуется к типу **double** и тот же тип имеет результат.

Иначе, если один из операндов имеет тип **unsigned long**, то другой преобразуется к типу **unsigned long** и таков же тип результата.

Иначе, если один из операндов имеет тип **long**, то другой преобразуется к типу **long** и таков же тип результата.

Иначе, если один из операндов имеет тип **unsigned**, то другой преобразуется к типу **unsigned** и таков же тип результата.

Иначе оба операнда должны иметь тип **int** и таков же тип результата.

Преобразования указателей

Везде, где указатели присваиваются, инициализируются, сравниваются и т.д. могут выполняться следующие преобразования.

- Константа 0 может преобразовываться в указатель, и гарантируется, что это значение породит указатель, отличный от указателя на любой объект.
- Указатель любого типа может преобразовываться в **void***.
- Указатель на класс может преобразовываться в указатель на открытый базовый класс этого класса.
- Имя вектора может преобразовываться в указатель на его первый элемент.

- Идентификатор, описанный как **«функция, возвращающая ...»**, всегда, когда он не используется в позиции имени функции в вызове, преобразуется в **«указатель на функцию, возвращающую ...»**.

Преобразования ссылок

Везде, где инициализируются ссылки, может выполняться следующее преобразование.

Ссылка на класс может преобразовываться в ссылку на открытый базовый класс этого класса.

Глава 13. Выражения и операции

Приоритет операций в выражениях такой же, как и порядок главных подразделов в этом разделе, наибольший приоритет у первого. Внутри каждого подраздела операции имеют одинаковый приоритет.

В каждом подразделе для рассматриваемых в нем операций определяется их левая или правая ассоциативность (порядок обработки операндов). Приоритет и ассоциативность всех операций собран вместе в описании грамматики. В остальных случаях порядок вычисления выражения не определен. Точнее, компилятор волен вычислять подвыражения в том порядке, который он считает более эффективным, даже если подвыражения вызывают побочные эффекты.

Порядок возникновения побочных эффектов не определен. Выражения, включающие в себя коммутативные и ассоциативные операции (*, +, &, |, ^), могут быть реорганизованы произвольным образом, даже при наличии скобок; для задания определенного порядка вычисления выражения необходимо использовать явную временную переменную.

Обработка переполнения и контроль деления при вычислении выражения машинно-зависимы. В большинстве существующих реализаций **C++** переполнение целого игнорируется; обработка деления на 0 и всех исключительных ситуаций с числами с плавающей точкой различаются от машины к машине и обычно могут регулироваться библиотечными функциями.

Кроме стандартного значения, операции могут быть перегружены, то есть, могут быть заданы их значения для случая их применения к типам, определяемым пользователем.

Основные выражения

Идентификатор есть первичное выражение, причем соответственно описанное. **Имя_функции_операции** есть идентификатор со специальным значением.

Операция **::**, за которой следует идентификатор из файловой области видимости, есть то же, что и идентификатор.

Это позволяет ссылаться на объект даже в том случае, когда его идентификатор скрыт.

Typedef-имя, за которым следует **::**, после чего следует идентификатор, является первичным выражением. **Typedef**-имя должно обозначать класс, и идентификатор должен обозначать член этого класса. Его тип специфицируется описанием идентификатора.

Typedef-имя может быть скрыто именем, которое не является именем типа. В этом случае **typedef**-имя все равно может быть найдено и его можно использовать.

Константа является первичным выражением. Ее тип должен быть **int**, **long** или **double** в зависимости от ее формы.

Строка является первичным выражением. Ее тип — **«массив символов»**. Обычно он сразу же преобразуется в указатель на ее первый символ.

Ключевое слово **this** является локальной переменной в теле функции члена. Оно является указателем на объект, для которого функция была вызвана.

Выражение, заключенное в круглые скобки, является первичным выражением, чей тип и значение те же, что и у не заключенного в скобки выражения. Наличие скобок не влияет на то, является выражение **lvalue** или нет.

Первичное выражение, за которым следует выражение в квадратных скобках, является первичным выражением. Интуитивный смысл — индекс. Обычно первичное выражение имеет тип **«указатель на ...»**, индексирующее выражение имеет тип **int** и тип результата есть **«...»**.

Выражение **E1[E2]** идентично (по определению) выражению ***((E1)+(E2))**.

Вызов функции является первичным выражением, за которым следуют скобки, содержащие список (возможно, пустой) разделенных запятыми выражений, составляющих фактические параметры для функции. Первичное выражение должно иметь тип **«функция, возвращающая ...»** или **«указатель на функцию, возвращающую ...»**, и результат вызова функции имеет тип **«...»**.

Каждый формальный параметр инициализируется фактическим параметром. Выполняются стандартные и определяемые пользователем преобразования. Функция может изменять значения своих формальных параметров, но эти изменения не могут повлиять на значения фактических параметров за исключением случая, когда формальный параметр имеет ссылочный тип.

Функция может быть описана как получающая меньше или больше параметров, чем специфицировано в описании функции. Каждый фактический параметр типа **float**, для которого нет формального параметра, преобразуются к типу **double**; и, как обычно, имена массивов преобразуются к указателям. Порядок вычисления параметров не определен языком; имейте в виду различия между компиляторами. Допустимы рекурсивные вызовы любых функций.

Первичное выражение, после которого стоит точка, за которой следует идентификатор (или идентификатор, уточненный **typedef**-именем с помощью операции **::**) является выражением. Первое выражение должно быть объектом класса, а идентификатор должен именовать член этого класса.

Значением является именованный член объекта, и оно является адресным, если первое выражение является адресным.

Следует отметить, что **«классовые объекты»** могут быть **структурами** или **объединениями**.

Первичное выражение, после которого стоит стрелка (**->**), за которой следует идентификатор (или идентификатор, уточненный **typedef**-именем с помощью операции **::**) является выражением.

Первое выражение должно быть указателем на объект класса, а идентификатор должен именовать член этого класса. Значение является адресом, ссылающимся на именованный член класса, на который указывает указательное выражение.

Так, выражение **E1->MOS** есть то же, что и **(*E1).MOS**. Если первичное выражение дает значение типа **«указатель на ...»**, значением выражения был объект, обозначаемый ссылкой. Ссылку можно считать именем объекта.

Унарные операции

Унарная операция ***** означает косвенное обращение: выражение должно быть указателем и результатом будет **lvalue**, ссылающееся на объ-

ект, на который указывает выражение. Если выражение имеет тип «**указатель на ...**», то тип результата есть «**...**».

Результатом унарной операции **&** является указатель на объект, на который ссылается операнд. Операнд должен быть **lvalue**. Если выражение имеет тип «**...**», то тип результата есть «**указатель на ...**».

Результатом унарной операции + является значение ее операнда после выполнения обычных арифметических преобразований. Операнд должен быть арифметического типа.

Результатом унарной операции является отрицательное значение ее операнда. Операнд должен иметь целый тип. Выполняются обычные арифметические преобразования. Отрицательное значение беззнаковой величины вычислятся посредством вычитания ее значения из **2n**, где **n** — число битов в целом типа **int**.

Результатом операции логического отрицания **!** является 1, если значение операнда 0, и 0, если значение операнда не 0. Результат имеет тип **int**. Применима к любому арифметическому типу или к указателям.

Операция ~ дает дополнение значения операнда до единицы. Выполняются обычные арифметические преобразования. Операнд должен иметь интегральный тип.

Увеличение и Уменьшение

Операнд префиксного ++ получает приращение. Операнд должен быть адресным. Значением является новое значение операнда, но оно не адресное. Выражение **++x** эквивалентно **x+=1**. По поводу данных о преобразованиях смотрите обсуждение операций сложения и присваивания.

Операнд префиксного **--** уменьшается аналогично действию префиксной операции ++.

Значение, получаемое при использовании постфиксного ++, есть значение операнда. Операнд должен быть адресным.

После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Значение, получаемое при использовании постфиксной **--**, есть значение операнда. Операнд должен быть адресным. После того, как результат отмечен, объект увеличивается так же, как и в префиксной операции ++. Тип результата тот же, что и тип операнда.

Sizeof

Операция **sizeof** дает размер операнда в байтах. (Байт не определяется языком иначе, чем через значение **sizeof**.

Однако, во всех существующих реализациях байт есть пространство, необходимое для хранения **char**).

При применении к массиву результатом является полное количество байтов в массиве. Размер определяется из описаний объектов, входящих в выражение. Семантически это выражение является беззнаковой константой и может быть использовано в любом месте, где требуется константа.

Операцию **sizeof** можно также применять к заключенному в скобки имени типа. В этом случае она дает размер, в байтах, объекта указанного типа.

Явное Преобразование Типа

Простое_имя_типа, возможно, заключенное в скобки, за которым идет заключенное в скобки выражение (или **список_выражений**, если тип является классом с соответствующим образом описанным конструктором) влечет преобразование значения выражения в названный тип.

Чтобы записать преобразование в тип, не имеющий простого имени, **имя_типа** должно быть заключено в скобки. Если имя типа заключено в скобки, выражение заключать в скобки необязательно. Такая запись называется приведением к типу.

Указатель может быть явно преобразован к любому из интегральных типов, достаточно по величине для его хранения. То, какой из **int** и **long** требуется, является машинно-зависимым. Отображающая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине.

Объект интегрального типа может быть явно преобразован в указатель. Отображающая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машиннозависимой.

Указатель на один тип может быть явно преобразован в указатель на другой тип. Использование полученного в результате указателя может привести к исключительной ситуации адресации, если исходный указатель не указывает на объект, соответствующим образом выровненный в памяти.

Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений. Различные машины могут различаться по числу бит в указателях и требованиям к выравниванию объектов.

Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Объект может преобразовываться в объект класса только если был описан соответствующий конструктор или операция преобразования.

Объект может явно преобразовываться в ссылочный тип **&X**, если указатель на этот объект может явно преобразовываться в **X***.

Свободная Память

Операция **new** создает объект типа **имя_типа**, к которому он применен. Время жизни объекта, созданного с помощью **new**, не ограничено областью видимости, в которой он создан. Операция **new** возвращает указатель на созданный ей объект.

Когда объект является массивом, возвращается указатель на его первый элемент. Например, и **new int** и **new int[10]** возвращают **int***.

Для объектов некоторых классов надо предоставлять инициализатор. Операция **new** для получения памяти вызывает функцию:

```
void* operator new (long);
```

Параметр задает требуемое число байтов. Память будет инициализирована. Если **operator new()** не может найти требуемое количество памяти, то она возвращает ноль.

Операция **delete** уничтожает объект, созданный операцией **new**. Ее результат является **void**. Операнд **delete** должен быть указателем, возвращенным **new**. Результат применения **delete** к указателю, который не был получен с помощью операции **new**. Однако уничтожение с помощью **delete** указателя со значением ноль безвредно.

Чтобы освободить указанную память, операция **delete** вызывает функцию

```
void operator delete (void*);
```

В форме

```
delete [ выражение ] выражение
```

второй параметр указывает на вектор, а первое выражение задает число элементов этого вектора. Задание числа элементов является избыточным за исключением случаев уничтожения векторов некоторых классов.

Мультипликативные операции

Мультипликативные операции *, / и % группируют слева направо. Выполняются обычные арифметические преобразования.

Синтаксис:

```
выражение * выражение
выражение / выражение
выражение % выражение
```

Бинарная операция * определяет умножение. Операция * ассоциативна и выражения с несколькими умножениями на одном уровне могут быть реорганизованы компилятором.

Бинарная операция / определяет деление. При делении положительных целых округление осуществляется в сторону 0, но если какой-либо из операндов отрицателен, то форма округления является машинно-зависимой.

На всех машинах, охватываемых данным руководством, остаток имеет тот же знак, что и делимое. Всегда истинно, что **(a/b)*b + a%b** равно **a** (если **b** не 0).

Бинарная операция % дает остаток от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды не должны быть числами с плавающей точкой.

Аддитивные операции

Аддитивные операции + и — группируют слева направо. Выполняются обычные арифметические преобразования.

Каждая операция имеет некоторые дополнительные возможности, связанные с типами.

Синтаксис:

```
выражение + выражение
выражение - выражение
```

Результатом операции + является сумма операндов. Можно суммировать указатель на объект массива и значение целого типа. Последнее во всех случаях преобразуется к смещению адреса с помощью умножения его на длину объекта, на который указывает указатель.

Результатом является указатель того же типа, что и исходный указатель, указывающий на другой объект того же массива и соответствующим образом смещенный от первоначального объекта. Так, если **P** есть указатель на объект массива, то выражение **P+1** есть указатель на следу-

ющий объект массива. Никакие другие комбинации типов для указателей не допустимы.

Операция + ассоциативна и выражение с несколькими умножениями на одном уровне может быть реорганизовано компилятором.

Результатом операции — является разность операндов. Выполняются обычные арифметические преобразования. Кроме того, значение любого целого типа может вычитаться из указателя, в этом случае применяются те же преобразования, что и к сложению.

Если вычитаются указатели на объекты одного типа, то результат преобразуется (посредством деления на длину объекта) к целому, представляющему собой число объектов, разделяющих объекты, указанные указателями.

В зависимости от машины результирующее целое может быть или типа **int**, или типа **long**.

Вообще говоря, это преобразование будет давать неопределенный результат кроме тех случаев, когда указатели указывают на объекты одного массива, поскольку указатели, даже на объекты одинакового типа, не обязательно различаются на величину, кратную длине объекта.

Операции сдвига

Операции сдвига << и >> группируют слева направо. Обе выполняют одно обычное арифметическое преобразование над своими операндами, каждый из которых должен быть целым. В этом случае правый операнд преобразуется к типу **int**; тип результата совпадает с типом левого операнда. Результат не определен, если правый операнд отрицателен или больше или равен длине объекта в битах.

Синтаксис:

```
выражение << выражение
выражение >> выражение
```

Значением **E1 << E2** является **E1** (рассматриваемое как битовое представление), сдвинутое влево на **E2** битов; освободившиеся биты заполняются нулями. Значением **E1 >> E2** является **E1**, сдвинутое вправо на **E2** битовых позиций.

Гарантируется, что сдвиг вправо является **логическим** (заполнение нулями), если **E1** является **unsigned**; в противном случае он может быть **арифметическим** (заполнение копией знакового бита).

Операции отношения

Операции отношения (сравнения) группируют слева направо, но этот факт не очень-то полезен: **a < b < c** не означает то, чем кажется.

Синтаксис:

```
выражение < выражение
выражение > выражение
выражение <= выражение
выражение >= выражение
```

Операции < (меньше чем), > (больше чем), <= и >= все дают **0**, если заданное соотношение ложно, и **1**, если оно истинно. Тип результата **int**.

Выполняются обычные арифметические преобразования. Могут сравниваться два указателя; результат зависит от относительного положения объектов, на которые указывают указатели, в адресном пространстве. Сравнение указателей переносимо только если указатели указывают на объекты одного массива.

Операции равенства

Синтаксис:

```
выражение == выражение
выражение != выражение
```

Операции **==** и **!=** в точности аналогичны операциям сравнения за исключением их низкого приоритета. (Так, а **< b == c < d** есть **1** всегда, когда **a < b** и **c < d** имеют одинаковое истинностное значение).

Указатель может сравниваться с **0**.

Операция побитовое И

Синтаксис:

```
выражение & выражение
```

Операция **&** ассоциативна, и выражения, содержащие **&**, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция **И** операндов. Операция применяется только к целым операндам.

Операция побитовое исключающее ИЛИ

Синтаксис:

```
выражение ^ выражение
```

Операция ^ ассоциативна, и выражения, содержащие ^, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция исключающее **ИЛИ** операндов. Операция применяется только к целым операндам.

Операция побитовое включающее ИЛИ

Синтаксис:

```
выражение | выражение
```

Операция | ассоциативна, и выражения, содержащие |, могут реорганизовываться. Выполняются обычные арифметические преобразования; результатом является побитовая функция включающее **ИЛИ** операндов. Операция применяется только к целым операндам.

Операция логическое И

Синтаксис:

```
выражение && выражение
```

Операция **&&** группирует слева направо. Она возвращает **1**, если оба операнда ненулевые, и **0** в противном случае. В противоположность операции **&** операция **&&** гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд есть **0**.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип **int**.

Операция логическое ИЛИ

Синтаксис:

```
выражение || выражение
```

Операция || группирует слева направо. Она возвращает **1**, если хотя бы один из ее операндов ненулевой, и **0** в противном случае. В противоположность операции | операция || гарантирует вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд не есть **0**.

Операнды не обязаны иметь один и тот же тип, но каждый из них должен иметь один из основных типов или быть указателем. Результат всегда имеет тип **int**.

Условная операция

Синтаксис:

```
выражение ? выражение : выражение
```

Условная операция группирует слева направо. Вычисляется первое выражение, и если оно не **0**, то результатом является значение второго выражения, в противном случае значение третьего выражения.

Если это возможно, то выполняются обычные арифметические преобразования для приведения второго и третьего выражения к общему типу.

Если это возможно, то выполняются преобразования указателей для приведения второго и третьего выражения к общему типу. Вычисляется только одно из второго и третьего выражений.

Операции присваивания

Есть много операций присваивания, все группируют слева направо. Все в качестве левого операнда требуют **lvalue**, и тип выражения присваивания тот же, что и у его левого операнда. Это **lvalue** не может ссылаться на константу (имя массива, имя функции или **const**).

Значением является значение, хранящееся в левом операнде после выполнения присваивания.

Синтаксис:

```
выражение операция_присваивания выражение
```

Где **операция_присваивания** — одна из:

- =
- +=
- -=
- *=
- /=
- %=
- >>=
- <<=
- &=
- ~=

- |=

В простом присваивании с = значение выражения замещает собой значение объекта, на который ссылается операнд в левой части. Если оба операнда имеют арифметический тип, то при подготовке к присваиванию правый операнд преобразуется к типу левого.

Если аргумент в левой части имеет указательный тип, аргумент в правой части должен быть того же типа или типа, который может быть преобразован к нему.

Оба операнда могут быть объектами одного класса. Могут присваиваться объекты некоторых производных классов.

Присваивание объекту типа **«указатель на ...»** выполнит присваивание объекту, денотируемому ссылкой.

Выполнение выражения вида **E1 op= E2** можно представить себе как эквивалентное **E1 = E1 op (E2)**; но **E1** вычисляется только один раз.

В **+=** и **-=** левый операнд может быть указателем, и в этом случае (интегральный) правый операнд преобразуется так, как объяснялось выше; все правые операнды и не являющиеся указателями левые должны иметь арифметический тип.

Операция запятая

Синтаксис:

```
выражение , выражение
```

Пара выражений, разделенных запятой, вычисляется слева направо, значение левого выражения теряется. Тип и значение результата являются типом и значением правого операнда. Эта операция группирует слева направо.

В контексте, где запятая имеет специальное значение, как например в списке фактических параметров функции и в списке инициализаторов, операция запятая, как она описана в этом разделе, может появляться только в скобках.

Например:

```
f (a,(t=3,t+2),c)
```

имеет три параметра, вторым из которых является значение 5.

Перегруженные операции

Большинство операций может быть перегружено, то есть, описано так, чтобы они получали в качестве операндов объекты классов. Изменить приоритет операций невозможно. Невозможно изменить смысл операций при применении их к неклассовым объектам.

Предопределенный смысл операций **=** и **&** (унарной) при применении их к объектам классов может быть изменен.

Эквивалентность операций, применяемых к основным типам (например, **++a** эквивалентно **a+=1**), не обязательно выполняется для операций, применяемых к классовым типам. Некоторые операции, например, присваивание, в случае применения к основным типам требуют, чтобы операнд был **lvalue**; это не требуется для операций, описанных для классовых типов.

Унарные операции

Унарная операция, префиксная или постфиксная, может быть определена или с помощью функции члена, не получающей параметров, или с помощью функции друга, получающей один параметр, но не двумя способами одновременно.

Так, для любой унарной операции **@**, **x@** и **@x** могут интерпретироваться как **x.операция@()** или **операция@(x)**. При перегрузке операций ++ и -- невозможно различить префиксное и постфиксное использование.

Бинарные операции

Бинарная операция может быть определена или с помощью функции члена, получающей один параметр, или с помощью функции друга, получающей два параметра, но не двумя способами одновременно. Так, для любой бинарной операции **@**, **x@y** может быть проинтерпретировано как **x.операция@(y)** или **операция@(x,y)**.

Особые операции

Вызов функции:

```
первичное_выражение (список_выражений opt)
```

и индексирование

```
первичное_выражение [выражение]
```

считаются бинарными операциями. Именами определяющей функции являются соответственно **operator()** и **operator[]**. Обращение **x(arg)** интерпретируется как **x.operator()(arg)** для классового объекта **x**. Индексирование **x[y]** интерпретируется как **x.operator[](y)**.

Глава 14. Описания

Описания используются для определения интерпретации, даваемой каждому идентификатору; они не обязательно резервируют память, связанную с идентификатором. Описания имеют вид:

```
спецификаторы_описания opt список_описателей opt;
описание_имени
asm_описание
```

Описатели в **списке_описателей** содержат идентификаторы, подлежащие описанию. **Спецификаторы_описания** могут быть опущены только в определениях внешних функций или в описаниях внешних функций. Список описателей может быть пустым только при описании класса или перечисления, то есть, когда **спецификаторы_описания** — это **class_спецификатор** или **enum_спецификатор**.

Список должен быть внутренне непротиворечив в описываемом ниже смысле.

Глава 15. Спецификаторы класса памяти

Спецификаторы **«класса памяти»** (**sc**-спецификатор) это:

- auto
- static
- extern
- register

Описания, использующие спецификаторы **auto**, **static** и **register** также служат определениями тем, что они вызывают резервирование соответствующего объема памяти. Если описание extern не является определением, то где-то еще должно быть определение для данных идентификаторов.

Описание **register** лучше всего представить как описание **auto** (автоматический) с подсказкой компилятору, что описанные переменные усиленно используются. Подсказка может быть проигнорирована. К ним не может применяться операция получения адреса **&**.

Спецификаторы **auto** или **register** могут применяться только к именам, описанным в блоке, или к формальным параметрам. Внутри блока не может быть описаний ни статических функций, ни статических формальных параметров.

В описании может быть задан максимум один **sc_спецификатор**. Если в описании отсутствует **sc_спецификатор**, то класс памяти принимается автоматическим внутри функции и статическим вне. Исключение: функции не могут быть автоматическими.

Спецификаторы **static** и **extern** могут использоваться только для имен объектов и функций.

Некоторые спецификаторы могут использоваться только в описаниях функций:

- overload
- inline
- virtual

Спецификатор перегрузки **overload** делает возможным использование одного имени для обозначения нескольких функций.

Спецификатор **inline** является только подсказкой компилятору, не влияет на смысл программы и может быть проигнорирован.

Он используется, чтобы указать на то, что при вызове функции **inline** — подстановка тела функции предпочтительнее обычной реализации вызова функции. Функция, определенная внутри описания класса, является **inline** по умолчанию. Спецификатор **virtual** может использоваться только в описаниях членов класса.

Спецификатор **friend** используется для отмены правил сокрытия имени для членов класса и может использоваться только внутри описаний классов. С помощью спецификатора **typedef** вводится имя для типа.

Спецификаторы Типа

Спецификаторами типов (**спецификатор_типа**) являются:

```
спецификатор_типа:
простое_имя_типа
class_спецификатор
enum-спецификатор сложный_спецификатор_типа
const
```

Слово **const** можно добавлять к любому допустимому **спецификатору_типа**. В остальных случаях в описании может быть дано не более од-

ного **спецификатора_типа**. Объект типа **const** не является **lvalue**. Если в описании опущен спецификатор типа, он принимается **int**.

Простое_имя_типа — это:

- char
- short
- int
- long
- unsigned
- float
- double
- const
- void

Слова **long**, **short** и **unsigned** можно рассматривать как прилагательные. Они могут применяться к типу **int**; **unsigned** может также применяться к типам **char**, **short** и **long**.

Сложный_спецификатор_типа — это:

- ключ typedef-имя
- ключ идентификатор

Ключ:

- class
- struct
- union
- enum

Сложный спецификатор типа можно использовать для ссылки на имя класса или перечисления там, где имя может быть скрыто локальным именем.

Например:

```
class x { ... };
void f(int x)
{
  class x a;
// ...
```


Если имя класса или перечисления ранее описано не было, **сложный_спецификатор_типа** работает как **описание_имени**.

Глава 16. Описатели

Список_описателей, появляющийся в описании, есть разделенная запятыми последовательность описателей, каждый из которых может иметь инициализатор.

```
список_описателей:
иниц_описатель
иниц_описатель , список_описателей
```

Где **иниц_описатель**:

```
описатель инициализатор opt
```

Спецификатор в описании указывает тип и класс памяти объектов, к которым относятся описатели.

Описатели имеют синтаксис:

```
описатель:
оп_имя
( описатель )
* const opt описатель
& const opt описатель
описатель
( список_описаний_параметров )
описатель
[ константное_выражение opt ]
```

Где **оп-имя**:

```
простое_оп_имя
typedef-имя :: простое_оп_имя
```

А **простое_оп_имя** — это:

```
идентификатор
typedef-имя
~ typedef-имя
имя_функции_операции
имя_функции_преобразования
```

Группировка та же, что и в выражениях.

Смысл описателей

Каждый описатель считается утверждением того, что если в выражении возникает конструкция, имеющая ту же форму, что и описатель, то она дает объект указанного типа и класса памяти. Каждый описатель содержит ровно одно **оп_имя**; оно определяет описываемый идентификатор. За исключением описаний некоторых специальных функций, **оп_имя** будет простым идентификатором.

Если в качестве описателя возникает ничем не снабженный идентификатор, то он имеет тип, указанный спецификатором, возглавляющим описание. Описатель в скобках эквивалентен описателю без скобок, но связку сложных описателей скобки могут изменять.

Теперь представим себе описание:

```
T D1
```

где **T** — спецификатор типа (как **int** и т.д.), а **D1 —** описатель. Допустим, что это описание заставляет идентификатор иметь тип **«... T»**, где **«...»** пусто, если идентификатор **D1** есть просто обычный идентификатор (так что тип **x** в **«int x»** есть просто **int**). Тогда, если **D1** имеет вид

```
*D
```

то тип содержащегося идентификатора есть **«... указатель на T»**

Если **D1** имеет вид

```
* const D
```

то тип содержащегося идентификатора есть **«... константный указатель на T»**, то есть, того же типа, что и ***D**, но не **lvalue**.

Если **D1** имеет вид

```
&D
```

или

```
& const D
```

то тип содержащегося идентификатора есть **«... ссылка на T»**. Поскольку ссылка по определению не может быть **lvalue**, использование **const** излишне. Невозможно иметь ссылку на **void (void&)**.

Если **D1** имеет вид

```
D (список_описаний_параметров)
```

то содержащийся идентификатор имеет тип **«... функция, принимающая параметр типа список_описаний_параметров и возвращающая T»**.

```
список_описаний_параметров:
список_описаний_парам opt ... opt
список_описаний_парам:
```



```
список_описаний_парам , описание_параметра
описание_параметра
описание_параметра:
спецификаторы_описания описатель
спецификаторы_описания описатель = выражение
спецификаторы_описания абстракт_описатель
спецификаторы_описания абстракт_описатель = выражение
```

Если **список_описаний_параметров** заканчивается многоточием, то о числе параметров известно лишь, что оно равно или больше числа специфицированных типов параметров; если он пуст, то функция не получает ни одного параметра.

Все описания для функции должны согласовываться и в типе возвращаемого значения, а также в числе и типе параметров.

Список_описаний_параметров используется для проверки и преобразования фактических параметров и для контроля присваивания указателю на функцию. Если в описании параметра специфицировано выражение, то это выражение используется как параметр по умолчанию.

Параметры по умолчанию будут использоваться в вызовах, где опущены стоящие в хвосте параметры. Параметр по умолчанию не может переопределяться более поздними описаниями. Однако, описание может добавлять параметры по умолчанию, не заданные в предыдущих описаниях.

Идентификатор может по желанию быть задан как имя параметра. Если он присутствует в описании функции, его использовать нельзя, поскольку он сразу выходит из области видимости. Если он присутствует в определении функции, то он именует формальный параметр.

Если **D1** имеет вид:

```
D[константное_выражение]
```

или

```
D[]
```

то тип содержащегося идентификатора есть **«... массив объектов типа T»**. В первом случае **константное_выражение** есть выражение, значение которого может быть определено во время компиляции, и тип которого **int**.

Если подряд идут несколько спецификаций **«массив из»**, то создается многомерный массив; константное выражение, определяющее границы массива, может быть опущено только для первого члена последовательности.

Этот пропуск полезен, когда массив является внешним, и настоящее определение, которое резервирует память, находится в другом месте.

Первое константное выражение может также быть опущено, когда за описателем следует инициализация. В этом случае используется размер, вычисленный исходя из числа начальных элементов.

Массив может быть построен из одного из основных типов, из указателей, из структуры или объединения или из другого массива (для получения многомерного массива).

Не все возможности, которые позволяет приведенный выше синтаксис, допустимы. Ограничения следующие: функция не может возвращать массив или функцию, хотя она может возвращать указатели на эти объекты; не существует массивов функций, хотя могут быть массивы указателей на функции.

Примеры

Описание:

```
int i;
int *ip;
int f ();
int *fip ();
int (*pfi) ();
```

описывает целое **i**, указатель **ip** на целое, функцию **f**, возвращающую целое, функцию **fip**, возвращающую указатель на целое, и указатель **pfi** на функцию, возвращающую целое. Особенно полезно сравнить последние две. Цепочка ***fip()** есть ***(fip())**, как предполагается в описании, и та же конструкция требуется в выражении, вызов функции **fip**, и затем косвенное использование результата через (указатель) для получения целого.

В описателе **(*pfi)()** внешние скобки необходимы, поскольку они также входят в выражение, для указания того, что функция получается косвенно через указатель на функцию, которая затем вызывается; это возвращает целое.

Функции **f** и **fip** описаны как не получающие параметров, и **fip** как указывающая на функцию, не получающую параметров.

Описание:

```
const a = 10, *pc = &a, *const cpc = pc;
int b, *const cp = &b;
```

описывает

- **a**: целую константу
- **pc**: указатель на целую константу
- **cpc**: константный указатель на целую константу
- **b**: целое
- **cp**: константный указатель на целое.

Значения **a**, **cpc** и **cp** не могут быть изменены после инициализации. Значение **pc** может быть изменено, как и объект, указываемый **cp**.

Примеры недопустимых выражений:

```
a = 1;
a++;
*pc = 2;
cp = &a;
cpc++;
```

Примеры допустимых выражений:

```
b = a;
*cp = a;
pc++;
pc = cpc;
```

Описание:

```
fseek (FILE*,long,int);
```

описывает функцию, получающую три параметра специальных типов. Поскольку тип возвращаемого значения не определен, принимается, что он **int**.

Описание:

```
point (int = 0,int = 0);
```

описывает функцию, которая может быть вызвана без параметров, с одним или двумя параметрами типа **int**.

Например:

```
point (1,2);
point (1) /* имеет смысл point (1,0); */
point ()    /* имеет смысл point (0,0); */
```

Описание:

```
printf (char* ... );
```

описывает функцию, которая может быть вызываться с различными числом и типами параметров.

Например:

```
printf ("hello, world");
printf ("a=%d b=%d",a,b);
printf ("string=%s",st);
```

Однако, она всегда должна иметь своим первым параметром **char***.

В качестве другого примера,

```
float fa[17], *afp[17];
```

описывает массив чисел с плавающей точкой и массив указателей на числа с плавающей точкой.

И, наконец,

```
static int x3d[3][5][7];
```

описывает массив целых, размером 3x6x7.

Совсем подробно:

- **x3d** является массивом из трех элементов;
- каждый из элементов является массивом из пяти элементов;
- каждый из последних элементов является массивом из семи целых.

Появление каждого из выражений **x3d**, **x3d[i]**, **x3d[i][j]**, **x3d[i][j][k]** может быть приемлемо.

Первые три имеют тип «**массив**», последний имеет тип **int**.

Глава 17. Описания классов

Класс специфицирует тип. Его имя становится **typedef**-имя, которое может быть использовано даже внутри самого спецификатора класса. Объекты класса состоят из последовательности членов.

Синтаксис:

```
заголовок_класса  { список_членов opt }
заголовок_класса { список_членов opt public :
список_членов opt }
```

Где **заголовок_класса**:

```
arper идентификатор opt
```


Где **идентификатор opt**:

```
public opt typedef-имя
```

А **arper** может иметь вид:

- class
- struct
- union

Структура является классом, все члены которого общие. Объединение является классом, содержащим в каждый момент только один член. Список членов может описывать члены вида: данные, функция, класс, определение типа, перечисление и поле.

Список членов может также содержать описания, регулирующие видимость имен членов.

Синтаксис:

```
описание_члена
список_членов opt
Описание_члена:
спецификаторы_описания opt описатель_члена;
Описатель_члена:
описатель
идентификатор opt :
константное_выражение
```

Члены, являющиеся классовыми объектами, должны быть объектами предварительно полностью описанных классов. В частности, класс **cl** не может содержать объект класса **cl**, но он может содержать указатель на объект класса **cl**. Имена объектов в различных классах не конфликтуют между собой и с обычными переменными.

Вот простой пример описания структуры:

```
struct tnode
  {
 char tword[20];
 int count;
 tnode *left;
 tnode *right;
  };
```

содержащей массив из 20 символов, целое и два указателя на такие же структуры.

Если было дано такое описание, то описание

```
tnode s, *sp
```

описывает **s** как структуру данного сорта и **sp** как указатель на структуру данного сорта.

При наличии этих описаний выражение

```
sp->count
```

ссылается на поле **count** структуры, на которую указывает **sp**;

```
s.left
```

ссылается на указатель левого поддерева структуры **s**;

```
s.right->tword[0]
```

ссылается на первый символ члена **tword** правого поддерева структуры **s**.

Статические члены

Член-данные класса может быть **static**; члены-функции не могут. Члены не могут быть **auto**, **register** или **extern**. Есть единственная копия статического члена, совместно используемая всеми членами класса в программе. На статический член **mem** класса **cl** можно ссылаться **cl:mem**, то есть без ссылки на объект. Он существует, даже если не было создано ни одного объекта класса **cl**.

Функции-члены

Функция, описанная как член, (без спецификатора **friend**) называется функцией членом и вызывается с помощью синтаксиса члена класса.

Например:

```
struct tnode
{
char tword[20];
int count;
tnode *left;
tnode *right;
void set (char* w,tnode* l,tnode* r);
};
tnode n1, n2;
n1.set ("asdf",&n2,0);
n2.set ("ghjk",0,0);
```

Определение функции члена рассматривается как находящееся в области видимости ее класса. Это значит, что она может непосредственно использовать имена ее класса. Если определение функции члена находится вне описания класса, то имя функции члена должно быть уточнено именем класса с помощью записи


```
typedef-имя . простое_оп_имя
```

Например:

```
void tnode.set (char* w,tnode* l,tnode* r)
{
count = strlen (w);
if (sizeof (tword) <= count) error ("tnode string too long");
strcpy (tword,w);
left = l;
right = r;
}
```

Имя функции **tnode.set** определяет то, что множество функций является членом класса **tnode**. Это позволяет использовать имена членов **word**, **count**, **left** и **right**.

В функции члене имя члена ссылается на объект, для которого была вызвана функция. Так, в вызове **n1.set(...) tword** ссылается на **n1.tword**, а в вызове **n2.set(...)** он ссылается на **n2.tword**. В этом примере предполагается, что функции **strlen**, **error** и **strcpy** описаны где-то в другом месте как внешние функции.

В члене функции ключевое слово **this** указывает на объект, для которого вызвана функция. Типом **this** в функции, которая является членом класса **cl**, является **cl***.

Если **mem** — член класса **cl**, то **mem** и **this->mem** — синонимы в функции члене класса **cl** (если **mem** не был использован в качестве имени локальной переменной в промежуточной области видимости).

Функция член может быть определена в описании класса. Помещение определения функции члена в описание класса является кратким видом записи описания ее в описании класса и затем определения ее как **inline** сразу после описания класса.

Например:

```
int b;
struct x
  {
  int f () { return b; }
  int f () { return b; }
  int b;
  };
```

означает

```
int b;
struct x
  {
  int f ();
  int b;
  };
inline x.f () { return b; }
```

Для функций членов не нужно использование спецификатора **overload**: если имя описывается как означающее несколько имен в классе, то оно перегружено.

Применение операции получения адреса к функциям членам допустимо. Тип параметра результирующей функции указатель на есть **(...)**, то есть, неизвестен. Любое использование его является зависимым от реализации, поскольку способ инициализации указателя для вызова функции члена не определен.

Производные классы

В конструкции

```
arper идентификатор:public opt typedef-имя
```

typedef-имя должно означать ранее описанный класс, называемый базовым классом для класса, подлежащего описанию. Говорится, что последний выводится из предшествующего.

На члены базового класса можно ссылаться, как если бы они были членами производного класса, за исключением тех случаев, когда имя базового члена было переопределено в производном классе; в этом случае для ссылки на скрытое имя может использоваться такая запись:

```
typedef-имя :: идентификатор
```

Например:

```
struct base
  {
  int a;
  int b;
  };
struct derived : public base
  {
  int b;
  int c;
  };
derived d;
```



```
d.a = 1;
d.base::b = 2;
d.b = 3;
d.c = 4;
```

осуществляет присваивание четырем членам **d**.

Производный тип сам может использоваться как базовый.

Виртуальные функции

Если базовый класс **base** содержит (виртуальную) **virtual** функцию **vf**, а производный класс **derived** также содержит функцию **vf**, то вызов **vf** для объекта класса **derived** вызывает **derived::vf**.

Например:

```
struct base
  {
  virtual void vf ();
  void f ();
  };
struct derived : public base
  {
  void vf ();
  void f ();
  };
derived d;
base* bp = &d;
bp->vf ();
bp->f ();
```

Вызовы вызывают, соответственно, **derived::vf** и **base::f** для объекта класса **derived**, именованного **d**. Так что интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызвана, в то время как интерпретация вызова невиртуальной функции зависит только от типа указателя, обозначающего объект.

Из этого следует, что тип объектов классов с виртуальными функциями и объектов классов, выведенных из таких классов, могут быть определены во время выполнения.

Если производный класс имеет член с тем же именем, что и у виртуальной функции в базовом классе, то оба члена должны иметь одинаковый тип. Виртуальная функция не может быть другом (**friend**).

Функция **f** в классе, выведенном из класса, который имеет виртуальную функцию **f**, сама рассматривается как виртуальная. Виртуальная

функция в базовом классе должна быть определена. Виртуальная функция, которая была определена в базовом классе, не нуждается в определении в производном классе.

В этом случае функция, определенная для базового класса, используется во всех вызовах.

Конструкторы

Член функция с именем, совпадающим с именем ее класса, называется конструктором. Конструктор не имеет типа возвращаемого значения; он используется для конструирования значений с типом его класса. С помощью конструктора можно создавать новые объекты его типа, используя синтаксис:

```
typedef-имя ( список_параметров opt )
```

Например:

```
complex zz = complex (1,2.3);
cprint (complex (7.8,1.2));
```

Объекты, созданные таким образом, не имеют имени (если конструктор не использован как инициализатор, как это было с **zz** выше), и их время жизни ограничено областью видимости, в которой они созданы. Они не могут рассматриваться как константы их типа.

Если класс имеет конструктор, то он вызывается для каждого объекта этого класса перед тем, как этот объект будет как-либо использован.

Конструктор может быть **overload**, но не **virtual** или **friend**. Если класс имеет базовый класс с конструктором, то конструктор для базового класса вызывается до вызова конструктора для производного класса.

Конструкторы для объектов членов, если таковые есть, выполняются после конструктора базового класса и до конструктора объекта, содержащего их.

Объяснение того, как могут быть специфицированы параметры для базового класса, а того, как конструкторы могут использоваться для управления свободной памятью.

Преобразования

Конструктор, получающий один параметр, определяет преобразование из типа своего параметра в тип своего класса. Такие преобразования неявно применяются дополнительно к обычным арифметическим преобразованиям.

Поэтому присваивание объекту из класса **X** допустимо, если или присваиваемое значение является **X**, или если **X** имеет конструктор, который получает присваиваемое значение как свой единственный параметр.

Аналогично конструкторы используются для преобразования параметров функции и инициализаторов.

Например:

```
class X { ... X (int); };
f (X arg)
  {
  X a = 1;          /* a = X (1) */
  a = 2;   /* a = X (2) */
  f (3);   /* f (X (3)) */
  }
```

Если для класса **X** не найден ни один конструктор, принимающий присваиваемый тип, то не делается никаких попыток отыскать конструктор для преобразования присваиваемого типа в тип, который мог бы быть приемлем для конструкторов класса **X**.

Например:

```
class X { ... X (int); };
class X { ... Y (X); };
Y a = 1; /* недопустимо: Y (X (1)) не пробуется */
```

Деструкторы

Функция член класса **cl** с именем **~cl** называется деструктором. Деструктор не возвращает никакого значения и не получает никаких параметров; он используется для уничтожения значений типа **cl** непосредственно перед уничтожением содержащего их объекта.

Деструктор не может быть **overload**, **virtual** или **friend**.

Деструктор для базового класса выполняется после деструктора производного от него класса. Как деструкторы используются для управления свободной памятью.

Видимость имен членов

Члены класса, описанные с ключевым словом **class**, являются закрытыми, это значит, что их имена могут использоваться только функциями членами и друзьями, пока они не появятся после метки **public:**. В этом случае они являются общими.

Общий член может использоваться любой функцией. Структура является классом, все члены которого общие.

Если перед именем базового класса в описании производного класса стоит ключевое слово **public**, то общие члены базового класса являются общими для производного класса; если нет, то они являются закрытыми.

Общий член **mem** закрытого базового класса **base** может быть описан как общий для производного класса с помощью описания вида:

```
typedef-имя . идентификатор;
```

в котором **typedef**-имя означает базовый класс, а идентификатор есть имя члена базового класса. Такое описание может появляться в общей части производного класса.

Рассмотрим:

```
class base
  {
  int a;
  public:
  int b,c;
  int bf ();
  };
class derived : base
  {
  int d;
  public:
  base.c;
  int e;
  int df ();
  };
int ef (derived&);
```

Внешняя функция **ef** может использовать только имена **c**, **e** и **df**. Являясь членом **derived**, функция **df** может использовать имена **b**, **c**, **bf**, **d**, **e** и **df**, но не **a**. Являясь членом **base**, функция **bf** может использовать члены **a**, **b**, **c** и **bf**.

Друзья (friends)

Другом класса является функция **не-член**, которая может использовать имена закрытых членов. Следующий пример иллюстрирует различия между членами и друзьями:

```
class private
  {
  int a;
  friend void friend_set (private*,int);
  public:
  void member_set (int);
  };
void friend_set (private* p,int i)
{ p->a=i; }
void private.member_set (int i)
{ a = i; }
private obj;
friend_set (&obj,10);
obj.member_set (10);
```


Если описание **friend** относится к перегруженному имени или операции, то другом становится только функция с описанными типами параметров. Все функции класса **cl1** могут быть сделаны друзьями класса **cl2** с помощью одного описания

```
class cl2
  {
  friend cl1;
  . . .
  };
```

Функция-операция

Большинство операций могут быть перегружены с тем, чтобы они могли получать в качестве операндов объекты класса.

```
имя_функции_операции: operator op
```

Где **op**:

- +
- -
- *
- /
- %
- ^
- &
- |
- ~
- !

- =
- <
- >
- +=
- -=
- *=
- /=
- %=
- ^=
- &=
- |=
- <<
- >>
- <<=
- >>=
- ==
- !=
- <=
- >=
- &&
- ||
- ++
- --
- ()
- []

Последние две операции — это вызов функции и индексирование. Функция операция может или быть функцией членом, или получать по меньшей мере один параметр класса.

Структуры

Структура есть класс, все члены которого общие. Это значит, что:

```
struct s { ... };
```

эквивалентно

```
class s { public: ... };
```

Структура может иметь функции члены (включая конструкторы и деструкторы).

Объединения

Объединение можно считать структурой, все объекты члены которой начинаются со смещения 0, и размер которой достаточен для содержания любого из ее объектов членов.

В каждый момент времени в объединении может храниться не больше одного из объектов членов.

Объединение может иметь функции члены (включая конструкторы и деструкторы).

Поля бит

Описатель члена вида:

```
идентификатор opt: константное_выражение
```

определяет поле; его длина отделяется от имени поля двоеточием. Поля упаковываются в машинные целые; они не являются альтернативой слов. Поле, не влезающее в оставшееся в целом место, помещается в следующее слово. Поле не может быть шире слова.

На некоторых машинах они размещаются справа налево, а на некоторых слева направо. Неименованные поля полезны при заполнении для согласования внешне предписанных размещений (форматов).

В особых случаях неименованные поля длины 0 задают выравнивание следующего поля по границе слова. Не требуется аппаратной поддержки любых полей, кроме целых. Более того, даже целые поля могут рассматриваться как **unsigned**.

По этим причинам рекомендуется описывать поля как **unsigned**. К полям не может применяться операция получения адреса **&**, поэтому нет указателей на поля. Поля не могут быть членами объединения.

Вложенные классы

Класс может быть описан внутри другого класса. В этом случае область видимости имен внутреннего класса его и общих имен ограничивается охватывающим классом. За исключением этого ограничения допустимо, чтобы внутренний класс уже был описан вне охватывающего класса.

Описание одного класса внутри другого не влияет на правила доступа к закрытым членам и не помещает функции члены внутреннего класса в область видимости охватывающего класса.

Например:

```
int x;
class enclose /* охватывающий */
  {
  int x;
  class inner
 {
 int y;
 f () { x=1 }
 ...
 };
  g (inner*);
  ...
  };
int inner; /* вложенный */
enclose.g (inner* p) { ... }
```

В этом примере **x** в **f** ссылается на **x**, описанный перед классом **enclose**. Поскольку у является закрытым членом **inner**, **g** не может его использовать. Поскольку **g** является членом **enclose**, имена, использованные в **g**, считаются находящимися в области видимости класса **enclose**.

Поэтому **inner** в описании параметров **g** относится к охваченному типу **inner**, а не к **int**.

Глава 18. Инициализация

Описание может задавать начальное значение описываемого идентификатора. Инициализатору предшествует **=**, и он состоит из выражения или списка значений, заключенного в фигурные скобки.

Синтаксис:

```
= expression
= { список_инициализаторов }
= { список_инициализаторов ,}
( список_выражений )
список_инициализаторов:
выражение список_инициализаторов,
список_инициализаторов
{ список_инициализаторов }
```

Все выражения в инициализаторе статической или внешней переменной должны быть константными выражениями или выражениями, которые сводятся к адресам ранее описанных переменных, возможно со смещением на константное выражение.

Автоматические и регистровые переменные могут инициализироваться любыми выражениями, включающими константы, ранее описанные переменные и функции.

Гарантируется, что неинициализированные статические и внешние переменные получают в качестве начального значения **«пустое значение»**. Когда инициализатор применяется к скаляру (указатель или объект арифметического типа), он состоит из одного выражения, возможно, заключенного в фигурные скобки.

Начальное значение объекта находится из выражения; выполняются те же преобразования, что и при присваивании.

Заметьте, что поскольку **()** не является инициализатором, то **«X a();»** является не описанием объекта класса **X**, а описанием функции, не получающей значений и возвращающей **X**.

Список инициализаторов

Когда описанная переменная является составной (класс или массив), то инициализатор может состоять из заключенного в фигурные скобки, разделенного запятыми списка инициализаторов для членов составного объекта, в порядке возрастания индекса или по порядку членов.

Если массив содержит составные подобъекты, то это правило рекурсивно применяется к членам составного подобъекта. Если инициализаторов в списке меньше, чем членов в составном подобъекте, то составной подобъект дополняется нулями.

Фигурные скобки могут опускаться следующим образом. Если инициализатор начинается с левой фигурной скобки, то следующий за ней список инициализаторов инициализирует члены составного объек-

та; наличие числа инициализаторов, большего, чем число членов, считается ошибочным.

Если, однако, инициализатор не начинается с левой фигурной скобки, то из списка берутся только элементы, достаточные для сопоставления членам составного объекта, частью которого является текущий составной объект.

Например,

```
int x[] = { 1, 3, 5 };
```

описывает и инициализирует **x** как одномерный массив, имеющий три члена, поскольку размер не был указан и дано три инициализатора.

```
float y[4][3] =
  {
  { 1, 3, 5 },
  { 2, 4, 6 },
  { 3, 5, 7 }
  };
```

является полностью снабженной квадратными скобками инициализацией: 1,3 и 5 инициализируют первый ряд массива **y[0]**, а именно, **y[0][2]**.

Аналогично, следующие две строки инициализируют **y[1]** и **y[2]**. Инициализатор заканчивается раньше, поэтому **y[3]** инициализируется значением 0. В точности тот же эффект может быть достигнут с помощью

```
float y[4][3] = { 1, 3, 5, 2, 4, 6, 3, 5, 7 };
```

Инициализатор для **y** начинается с левой фигурной скобки, но не начинается с нее инициализатор для **y[0]**, поэтому используется три значения из списка. Аналогично, следующие три успешно используются для **y[1]** и следующие три для **y[2]**.

```
float y[4][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
```

инициализирует первый столбец **y** (рассматриваемого как двумерный массив) и оставляет остальные элементы нулями.

Классовые объекты

Объект с закрытыми членами не может быть инициализирован с помощью простого присваивания, как это описывалось выше; это же относится к объекту объединение. Если класс имеет конструктор, не получающий значений, то этот конструктор используется для объектов, которые явно не инициализированы.

Параметры для конструктора могут также быть представлены в виде заключенного в круглые скобки списка.

Например:

```
struct complex
  {
  float re;
  float im;
  complex (float r,float i)        { re=r; im=i; }
  complex (float r) { re=r; im=0; }
  };
complex zz (1,2.3);
complex* zp = new complex (1,2.3);
```

Инициализация может быть также выполнена с помощью явного присваивания; преобразования производятся.

Например:

```
complex zz1 = complex (1,2.3);
complex zz2 = complex (123);
complex zz3 = 123;
complex zz4 = zz3;
```

Если конструктор ссылается на объект своего собственного класса, то он будет вызываться при инициализации объекта другим объектом этого класса, но не при инициализации объекта конструктором.

Объект класса, имеющего конструкторы, может быть членом составного объекта только если он сам не имеет конструктора или если его конструкторы не имеют параметров. В последнем случае конструктор вызывается при создании составного объекта.

Если член составного объекта является членом класса с деструкторами, то этот деструктор вызывается при уничтожении составного объекта.

Ссылки

Когда переменная описана как **T&**, что есть **«ссылка на тип T»**, она может быть инициализирована или указателем на тип **T**, или объектом типа **T**. В последнем случае будет неявно применена операция взятия адреса **&**.

Например:

```
int i;
int& r1 = i;
int& r2 = &i;
```

И **r1** и **r2** будут указывать на **i**.

Обработка инициализации ссылки очень сильно зависит от того, что ей присваивается. Ссылка неявно переадресуется при ее использовании.

Например:

```
r1 = r2;
```

означает копирование целого, на которое указывает **r2**, в целое, на которое указывает **r1**.

Ссылка должна быть инициализирована. Таким образом, ссылку можно считать именем объекта.

Чтобы получить указатель **pp**, обозначающий тот объект, что и ссылка **rr**, можно написать **pp=&rr**. Это будет проинтерпретировано как:

```
pp=&*rr
```

Если инициализатор для ссылки на тип **T** не является адресным выражением, то будет создан и инициализирован с помощью правил инициализации объект типа **T**. Тогда значением ссылки станет адрес объекта. Время жизни объекта, созданного таким способом, будет в той области видимости, в которой он создан.

Например:

```
double& rr = 1;
```

допустимо, и **rr** будет указывать на объект типа **double**, в котором хранится значение 1.0.

Ссылки особенно полезны в качестве типов параметров.

Массивы символов

Последняя сокращенная запись позволяет инициализировать строкой массив данных типа **char**. В этом случае последовательные символы строки инициализируют члены массива.

Например:

```
char msg[] = "Syntax error on line %d\n";
```

демонстрирует массив символов, члены которого инициализированы строкой.

Имена типов

Иногда (для неявного задания преобразования типов и в качестве параметра **sizeof** или **new**) нужно использовать имя типа данных. Это выполняется при помощи **«имени типа»**, которое по сути является описанием для объекта этого типа, в котором опущено имя объекта.

Синтаксис:

```
спецификатор_типа абстрактный_описатель
Абстрактный_описатель:
пустой
*
абстрактный_описатель      абстрактный_описатель ( список_описателей_параметров)
абстрактный_описатель
[ константное_выражение opt ]
( абстрактный_описатель )
```

Является возможным идентифицировать положение в **абстрактном_описателе**, где должен был бы появляться идентификатор в случае, если бы конструкция была описателем в описании. Тогда именованный тип является тем же, что и тип предполагаемого идентификатора.

Например:

```
int
int *
int *[3]
int *()
int (*)()
```

именует, соответственно, типы **«целое»**, **«указатель на целое»**, **«указатель на массив из трех целых»**, **«функция, возвращающая указатель на функцию, возвращающую целое»** и **«указатель на целое»**.

Простое имя типа есть имя типа, состоящее из одного идентификатора или ключевого слова.

Простое_имя_типа:

- typedef-имя
- char
- short
- int
- long
- unsigned
- float
- double

Они используются в альтернативном синтаксисе для преобразования типов.

Например:

```
(double) a
```

может быть также записано как

```
double (a)
```

Определение типа typedef

Описания, содержащие **спецификатор_описания typedef**, определяют идентификаторы, которые позднее могут использоваться так, как если бы они были ключевыми словами типа, именующее основные или производные типы. Синтаксис:

```
typedef-имя:
идентификатор
```

Внутри области видимости описания, содержащего **typedef**, каждый идентификатор, возникающий как часть какого-либо описателя, становится в этом месте синтаксически эквивалентным ключевому слову типа, которое именует тип, ассоциированный с идентификатором. Имя класса или перечисления также является **typedef**-именем.

Например, после

```
typedef int MILES, *KLICKSP;
struct complex { double re, im; };
```

каждая из конструкций

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

является допустимым описанием; **distance** имеет тип **int**, **metricp** имеет тип «**указатель на int**».

typedef не вводит новых типов, но только синонимы для типов, которые могли бы быть определены другим путем. Так в приведенном выше примере **distance** рассматривается как имеющая в точности тот же тип, что и любой другой **int** объект.

Но описание класса вводит новый тип.

Например:

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

описывает три переменных трех различных типов.

Описание вида:

```
arper      идентификатор ;
enum идентификатор ;
```

определяет то, что идентификатор является именем некоторого (возможно, еще не определенного) класса или перечисления. Такие описания позволяют описывать классы, ссылающихся друг на друга.

Например:

```
class vector;
class matrix
  {
  ...
  friend matrix operator* (matrix&,vector&);
  };
class vector
  {
  ...
  friend matrix operator*  (matrix&,vector&);
  };
```

Глава 19. Перегруженные имена функций

В тех случаях, когда для одного имени определено несколько (различных) описаний функций, это имя называется перегруженным.

При использовании этого имени правильная функция выбирается с помощью сравнения типов фактических параметров с типами параметров в описаниях функций. К перегруженным именам неприменима операция получения адреса **&**.

Из обычных арифметических преобразований для вызова перегруженной функции выполняются только **char->short->int**, **int->double**, **int->long** и **float->double**.

Для того, чтобы перегрузить имя функции **не-члена** описание **overload** должно предшествовать любому описанию функции.

Например:

```
overload abs;
int abs (int);
double abs (double);
```

Когда вызывается перегруженное имя, по порядку производится сканирование списка функций для нахождения той, которая может быть вызвана.

Например, **abs(12)** вызывает **abs(int)**, а **abs(12.0)** будет вызывать **abs(double)**. Если бы был зарезервирован порядок вызова, то оба обращения вызвали бы **abs(double)**.

Если в случае вызова перегруженного имени с помощью вышеуказанного метода не найдено ни одной функции, и если функция получает параметр типа класса, то конструкторы классов параметров (в этом случае существует единственный набор преобразований, делающий вызов допустимым) применяются неявным образом.

Например:

```
class X { ... X (int); };
class Y { ... Y (int); };
class Z { ... Z (char*); };
overload int f (X), f (Y);
overload int g (X), g (Y);
f (1); /* неверно: неоднозначность f(X(1)) или f(Y(1)) */
g (1); /* g(X(1)) */
g ("asdf"); /* g(Z("asdf")) */
```

Все имена функций операций являются автоматически перегруженными.

Глава 20. Описание перечисления

Перечисления являются **int** с именованными константами.

```
enum_спецификатор:
enum идентификатор opt { enum_список }
enum_список:
перечислитель
enum_список, перечислитель
Перечислитель:
идентификатор
идентификатор = константное_выражение
```

Идентификаторы в **enum**-списке описаны как константы и могут появляться во всех местах, где требуются константы.

Если не появляется ни одного перечислителя **c =**, то значения всех соответствующих констант начинаются с 0 и возрастают на 1 по мере чтения описания слева направо.

Перечислитель **c =** дает ассоциированному с ним идентификатору указанное значение; последующие идентификаторы продолжают прогрессию от присвоенного значения.

Имена перечислителей должны быть отличными от имен обычных переменных. Значения перечислителей не обязательно должны быть различными.

Роль идентификатора в спецификаторе перечисления **enum_спецификатор** полностью аналогична роли имени класса; он именует определенный нумератор.

Например:

```
enum color { chartreuse, burgundy, claret=20, winedark };
  ...
  color *cp, col;
  ...
  col = claret;
  cp = &col;
  ...
  if (*cp == burgundy) ...
```

делает **color** именем типа, описывающего различные цвета, и затем описывает cp как указатель на объект этого типа. Возможные значения извлекаются из множества **{ 0, 1, 20, 21 }**.

Глава 21. Описание Asm

Описание **Asm** имеет вид:

```
asm (строка);
```

Смысл описания **asm** не определен. Обычно оно используется для передачи информации ассемблеру через компилятор.

Глава 22. Операторы

Операторы выполняются последовательно во всех случаях кроме особо оговоренных.

Оператор выражение

Большинство операторов является операторами выражение, которые имеют вид

```
выражение ;
```

Обычно операторы выражение являются присваиваниями и вызовами функций.

Составной оператор, или блок

Составной оператор (называемый также **«блок»**, что эквивалентно) дает возможность использовать несколько операторов в том месте, где предполагается использование одного:

```
Составной_оператор:
{ список_описаний opt список_операторов opt }
список_описаний:
описание
описание список_описаний
Список_операторов:
оператор
оператор список_операторов
```

Если какой-либо из идентификаторов в **списке_описаний** был ранее описан, то внешнее описание выталкивается на время выполнения блока, и снова входит в силу по его окончании.

Каждая инициализация **auto** или **register** переменных производится всякий раз при входе в голову блока. В блок делать передачу; в этом случае инициализации не выполняются. Инициализации переменных, имеющих класс памяти **static** осуществляются только один раз в начале выполнения программы.

Условный оператор

Есть два вида условных операторов:

```
if ( выражение ) оператор
if ( выражение ) оператор else оператор
```


В обоих случаях вычисляется выражение, и если оно не ноль, то выполняется первый подоператор. Во втором случае второй подоператор выполняется, если выражение есть 0. Как обычно, неоднозначность **«else»** разрешается посредством того, что **else** связывается с последним встреченным **if**, не имеющим **else**.

Оператор while

Оператор **while** имеет вид:

```
while ( выражение ) оператор
```

Выполнение подоператора повторяется, пока значение выражения остается ненулевым. Проверка выполняется перед каждым выполнением оператора.

Оператор do

Оператор **do** имеет вид:

```
do оператор while (выражение);
```

Выполнение подоператора повторяется до тех пор, пока значение выражения не станет нулем. Проверка выполняется после каждого выполнения оператора.

Оператор for

Оператор **for** имеет вид:

```
for ( выражение_1 opt ; выражение_2 opt ; выражение_3 opt )
оператор
```

Этот оператор эквивалентен следующему:

```
выражение_1;
while (выражение_2)
{ оператор выражение_3; }
```

- первое выражение задает инициализацию цикла;
- второе выражение задает осуществляемую перед каждой итерацией проверку, по которой производится выход из цикла, если выражение становится нулем;
- третье выражение часто задает приращение, выполняемое после каждой итерации.

Каждое или все выражения могут быть опущены. Отсутствие **выражения_2** делает подразумеваемое **while**-предложение эквивалентным **while(1)**; остальные опущенные выражения просто пропускаются в описанном выше расширении.

Оператор switch

Оператор **switch** вызывает передачу управления на один из нескольких операторов в зависимости от значения выражения. Он имеет вид

```
switch ( выражение ) оператор
```

Выражение должно быть целого типа или типа указателя. Любой оператор внутри оператора может быть помечен одним или более префиксом **case** следующим образом:

```
case константное_выражение :
```

где **константное_выражение** должно иметь тот же тип что и выражение-переключатель; производятся обычные арифметические преобразования. В одном операторе **switch** никакие две константы, помеченные **case**, не могут иметь одинаковое значение.

Может также быть не более чем один префикс оператора вида

```
default :
```

Когда выполнен оператор **switch**, проведено вычисление его выражения и сравнение его с каждой **case** константой.

Если одна из констант равна значению выражения, то управление передается на выражение, следующее за подошедшим префиксом **case**.

Если никакая **case** константа не соответствует выражению, и есть префикс **default**, то управление передается на выражение, которому он предшествует.

Если нет соответствующих вариантов **case** и **default** отсутствует, то никакой из операторов в операторе **switch** не выполняется.

Префиксы **case** и **default** сами по себе не изменяют поток управления, который после задержки идет дальше, перескакивая через эти префиксы.

Обычно зависящий от **switch** оператор является составным. В голове этого оператора могут стоять описания, но инициализации автоматических и регистровых переменных являются безрезультатными.

Оператор break

Оператор

```
break ;
```

прекращает выполнение ближайшего охватывающего **while**, **do**, **for** или **switch** оператора; управление передается на оператор, следующий за законченным.

Оператор continue

Оператор

```
continue ;
```

вызывает передачу управления на управляющую продолжением цикла часть наименьшего охватывающего оператора **while**, **do** или **for**; то есть на конец петли цикла. Точнее, в каждом из операторов

```
while (...)
do
for (...)
{
  {
 {
 ...
 ...
 ...
 contin:;
 contin:;
 contin:;
 }
  }
}
while (...);
```

continue эквивалентно **goto contin** (За **contin:** следует пустой оператор).

Оператор return

Возврат из функции в вызывающую программу осуществляется с помощью оператора **return**, имеющего один из двух видов:

```
return ;
return выражение ;
```

Первый может использоваться только в функциях, не возвращающих значения, т.е. в функциях с типом возвращаемого значения **void**.

Вторая форма может использоваться только в функциях, не возвращающих значение; вызывающей функцию программе возвращается значение выражения.

Если необходимо, то выражение преобразуется, как это делается при присваивании, к типу функции, в которой оно возникло.

Обход конца функции эквивалентен возврату **return** без возвращаемого значения.

Оператор goto

Можно осуществлять безусловную передачу управления с помощью оператора

```
goto идентификатор ;
```

Идентификатор должен быть меткой, расположенной в текущей функции.

Помеченные операторы

Перед любым оператором может стоять префикс метка, имеющий вид

```
идентификатор :
```

который служит для описания идентификатора как метки. Метка используется только как объект для **goto**.

Областью видимости метки является текущая функция, исключая любой подблок, в котором был переописан такой же идентификатор.

Пустой оператор

Пустой оператор имеет вид

```
;
```

Пустой оператор используется для помещения метки непосредственно перед **}** составного оператора или того, чтобы снабдить такие операторы, как **while**, пустым телом.

Оператор delete

Оператор **delete** имеет вид

```
delete выражение ;
```

Результатом выражения должен быть указатель. Объект, на который он указывает, уничтожается. Это значит, что после оператора уничтожения **delete** нельзя гарантировать, что объект имеет определенное значение.

Эффект от применения **delete** к указателю, не полученному из операции **new**, не определен. Однако, уничтожение указателя с нулевым значением безопасно.

Оператор asm

Оператор **asm** имеет вид

```
asm ( строка) ;
```


Смысл оператора **asm** не определен. Обычно он используется для передачи информации через компилятор ассемблеру.

Глава 23. Внешние определения

Программа на **C++** состоит из последовательности внешних определений. Внешнее определение описывает идентификатор как имеющий класс памяти **static** и определяет его тип. Спецификатор типа может также быть пустым, и в этом случае принимается тип **int**.

Область видимости внешних определений простирается до конца файла, в котором они описаны, так же, как действие описаний сохраняется до конца блока. Синтаксис внешних определений тот же, что и у описаний, за исключением того, что только на этом уровне и внутри описаний классов может быть задан код (текст программы) функции.

Определения функций

Определения функций имеют вид:

```
определение_функции:
спецификаторы_описания описатель_функции opt инициализатор_базового_класса
opt тело_функции
```

Единственными спецификаторами класса памяти (**sc-спецификаторами**), допустимыми среди спецификаторов описания, являются **extern**, **static**, **overload**, **inline** и **virtua**l.

Описатель функции похож на описатель **«функции, возвращающей ...»**, за исключением того, что он включает в себя имена формальных параметров определяемой функции.

Описатель функции имеет вид:

```
описатель_функции:
описатель ( список_описаний_параметров )
```

Единственный класс памяти, который может быть задан, это тот, при котором соответствующий фактический параметр будет скопирован, если это возможно, в регистр при входе в функцию. Если в качестве инициализатора для параметра задано константное выражение, то это значение используется как значение параметра по умолчанию.

Тело функции имеет вид

```
тело_функции:
составной_оператор
```

Вот простой пример полного определения функции:

```
int max (int a,int b,int c)
  {
  int m = (a > b) ? a : b;
  return (m > c) ? m : c;
  }
```

Здесь

- **int** является спецификатором типа;
- **max (int a, int b, int c)** является описателем функции;
- **{ ... }** — блок, задающий текст программы (код) оператора.

Поскольку в контексте выражения имя (точнее, имя как формальный параметр) считается означающим указатель на первый элемент массива, то описания формальных параметров, описанных как «**массив из ...**», корректируются так, чтобы читалось «**указатель на ...**».

Инициализатор базового класса имеет вид:

```
инициализатор_базового_класса:
: ( список_параметров opt )
```

Он используется для задания параметров конструктора базового класса в конструкторе производного класса.

Например:

```
struct base { base (int); ... };
struct derived : base { derived (int); ... };
derived.derived (int a) : (a+1) { ... }
derived d (10);
```

Конструктор базового класса вызывается для объекта **d** с параметром 11.

Определения внешних данных

Определения внешних данных имеют вид:

```
определение_данных:
описание
```

Класс памяти таких данных статический.

Если есть более одного определения внешних данных одного имени, то определения должны точно согласовываться по типу и классу памяти, и инициализаторы (если они есть), должны иметь одинаковое значение.

Глава 24. Командные строки компилятора

Компилятор языка **C++** содержит препроцессор, способный выполнять макроподстановки, условную компиляцию и включение именованных файлов. Строки, начинающиеся с #, относятся к препроцессору. Эти строки имеют независимый от остального языка синтаксис; они могут появляться в любом месте оказывать влияние, которое распространяется (независимо от области видимости) до конца исходного файла программы.

Заметьте, что определения **const** и **inline** дают альтернативы для большинства использований **#define**.

Замена идентификаторов

Командная строка компилятора имеет вид:

```
#define идент строка_символов
```

и вызывает замену препроцессором последующих вхождений идентификатора, заданного строкой символов. Точка с запятой внутри (или в конце) строки символов является частью этой строки.

Строка:

```
#define идент ( идент , ..., идент ) строка_символов
```

где отсутствует пробел между первым идентификатором и скобка, является макроопределением с параметрами. Последующие вхождения первого идентификатора с идущими за ним **(,** последовательностью символов, разграниченной запятыми, и **),** заменяются строкой символов, заданной в определении.

Каждое местоположение идентификатора, замеченного в списке параметров определения, заменяется соответствующей строкой из вызова. Фактическими параметрами вызова являются строки символов, разделенные запятыми; однако запятые в строке, заключенной в кавычки, или в круглых скобках не являются разделителями параметров.

Число формальных и фактических параметров должно совпадать. Строки и символьные константы в символьной строке сканируются в

поисках формальных параметров, но строки и символьные константы в остальной программе не сканируются в поисках определенных (с помощью **define**) идентификаторов.

В обоих случаях строка замещения еще раз сканируется в поисках других определенных идентификаторов. В обоих случаях длинное определение может быть продолжено на другой строке с помощью записи \ в конце продолжаемой строки.

Командная строка:

```
#undef идент
```

влечет отмену препроцессорного определения идентификатора.

Включение файлов

Командная строка компилятора:

```
#include "имя_файла"
```

вызывает замену этой строки полным содержимым файла **имя_файла**. Сначала именованный файл ищется в директории первоначального исходного файла, а затем в стандартных или заданных местах.

Альтернативный вариант. Командная строка:

```
#include <имя_файла>
```

производит поиск только в стандартном или заданном месте, и не ищет в директории первоначального исходного файла. (То, как эти места задаются, не является частью языка.)

Включения с помощью **#include** могут быть вложенными.

Условная компиляция

Командная строка компилятора:

```
#if выражение
```

проверяет, является ли результатом вычисления выражения **не-ноль**. Выражение должно быть константным выражением. Применительно к использованию данной директивы есть дополнительные ограничения: константное выражение не может содержать **sizeof** или перечислимые константы.

Кроме обычных операций **C** может использоваться унарная операция **defined**. В случае применения к идентификатору она дает значение **не-ноль**, если этот идентификатор был ранее определен с помощью **#define** и после этого не было отмены определения с помощью **#undef**; иначе ее значение **0**.

Командная строка:

```
#ifdef идент
```

проверяет, определен ли идентификатор в препроцессоре в данный момент; то есть, был ли он объектом командной строки **#define**.

Командная строка:

```
#ifndef идент
```

проверяет, является ли идентификатор неопределенным в препроцессоре в данный момент.

После строки каждого из трех видов может стоять произвольное количество строк, возможно, содержащих командную строку

```
#else
```

и далее до командной строки

```
#endif
```

Если проверенное условие истинно, то все строки между **#else** и **#endif** игнорируются. Если проверенное условие ложно, то все строки между проверкой и **#else** или, в случае отсутствия **#else**, **#endif**, игнорируются.

Эти конструкции могут быть вложенными.

Управление строкой

Для помощи другим препроцессорам, генерирующим программы на Си.

Строка:

```
#line константа "имя_файла"
```

заставляет компилятор считать, например, в целях диагностики ошибок, что константа задает номер следующей строки исходного файла, и текущий входной файл именуется идентификатором. Если идентификатор отсутствует, то запомненное имя файла не изменяется.

Глава 25. Обзор типов

Здесь кратко собрано описание действий, которые могут совершаться над объектами различных типов.

Классы

Классовые объекты могут присваиваться, передаваться функциям как параметры и возвращаться функциями. Другие возможные операции, как, например, проверка равенства, могут быть определены пользователем.

Функции

Есть только две вещи, которые можно проделывать с функцией: вызывать ее и брать ее адрес. Если в выражении имя функции возникает не в положении имени функции в вызове, то генерируется указатель на функцию. Так, для передачи одной функции другой можно написать

```
typedef int (*PF) ();
extern g (PF);
extern f ();
...
g (f);
```

Тогда определение **g** может иметь следующий вид:

```
g (PF funcp)
  {
  ...
  (*funcp) ();
  ...
  }
```

Заметьте, что **f** должна быть описана явно в вызывающей программе, поскольку ее появление в **g(f)** не сопровождалось **(**.

Массивы, указатели и индексирование

Всякий раз, когда в выражении появляется идентификатор типа массива, он преобразуется в указатель на первый член массива. Из-за преобразований массивы не являются адресами. По определению операция индексирования **[]** интерпретируется таким образом, что **E1[E2]** идентично ***((E1)+(E2))**.

В силу правил преобразования, применяемых к **+**, если **E1** массив и **E2** целое, то **E1[E2]** относится к **E2**-ому члену **E1**. Поэтому, несмотря на такое проявление асимметрии, индексирование является коммутативной операцией.

Это правило сообразным образом применяется в случае многомерного массива. Если **E** является **n**-мерным массивом ранга **i*j*...*k**, то возникающее в выражении **E** преобразуется в указатель на **(n-1)**-мерный массив ранга **j*...*k**.

Если к этому указателю, явно или неявно, как результат индексирования, применяется операция *, ее результатом является **(n-1)**-мерный массив, на который указывалось, который сам тут же преобразуется в указатель.

Например:

```
int x[3][5];
```

Здесь **x** — массив целых размером 3 на 5. Когда **x** возникает в выражении, он преобразуется в указатель на (первый из трех) массив из 5 целых. В выражении **x[i]**, которое эквивалентно ***(x+1)**, **x** сначала преобразуется, как описано, в указатель, затем 1 преобразуется к типу **x**, что включает в себя умножение 1 на длину объекта, на который указывает указатель, а именно объект из 5 целых.

Результаты складываются, и используется косвенная адресация для получения массива (из 5 целых), который в свою очередь преобразуется в указатель на первое из целых.

Если есть еще один индекс, снова используется тот же параметр; на этот раз результат является целым.

Именно из всего этого проистекает то, что массивы в C хранятся по строкам (быстрее всего изменяется последний индекс), и что в описании первый индекс помогает определить объем памяти, поглощаемый массивом, но не играет никакой другой роли в вычислениях индекса.

Явные преобразования указателей

Определенные преобразования, включающие массивы, выполняются, но имеют зависящие от реализации аспекты. Все они задаются с помощью явной операции преобразования типов.

Указатель может быть преобразован к любому из целых типов, достаточно больших для его хранения. То, какой из **int** и **long** требуется, является машинно-зависимым. Преобразующая функция также является машинно-зависимой, но предполагается, что она не содержит сюрпризов для того, кто знает структуру адресации в машине.

Объект целого типа может быть явно преобразован в указатель. Преобразующая функция всегда превращает целое, полученное из указателя, обратно в тот же указатель, но в остальных случаях является машинно-зависимой.

Указатель на один тип может быть преобразован в указатель на другой тип. Использование результирующего указателя может вызывать особые ситуации, если исходный указатель не указывает на объект, соответствующим образом выровненный в памяти.

Гарантируется, что указатель на объект данного размера может быть преобразован в указатель на объект меньшего размера и обратно без изменений.

Например, программа, выделяющая память, может получать размер (в байтах) размещаемого объекта и возвращать указатель на **char**; это можно использовать следующим образом.

```
extern void* alloc ();
double* dp;
dp = (double*) alloc (sizeof double));
*dp= 22.0 / 7.0;
```

alloc должна обеспечивать (машинно-зависимым образом) то, что возвращаемое ею значение подходит для преобразования в указатель на **double**; в этом случае использование функции мобильно. Различные машины различаются по числу бит в указателях и требованиям к выравниванию объектов. Составные объекты выравниваются по самой строгой границе, требуемой каким-либо из его составляющих.

Константные выражения

В нескольких местах C++ требует выражения, вычисление которых дает константу: в качестве границы массива, в **case** выражениях, в качестве значений параметров функции, присваиваемых по умолчанию и в инициализаторах.

В первом случае выражение может включать только целые константы, символьные константы, константы, описанные как имена, и **sizeof** выражения, возможно, связанные бинарными операциями:

- `+`
- `-`
- `*`
- `/`
- `%`
- `&`
- `|`
- `^`
- `<<`
- `>>`

- `==`
- `!=`
- `<`
- `>`
- `<=`
- `>=`
- `&&`
- `||`

или унарными операциями:

- `-`
- `~`
- `!`

или тернарными операциями:

- `?`
- `:`

Скобки могут использоваться для группирования, но не для вызова функций.

Большая широта допустима для остальных трех случаев использования; помимо константных выражений, обсуждавшихся выше, допускаются константы с плавающей точкой, и можно также применять унарную операцию **&** к внешним или статическим объектам, или к внешним или статическим массивам, индексированным константным выражением. Унарная операция **&** может также быть применена неявно с помощью употребления неиндексированных массивов и функций.

Основное правило состоит в том, что инициализаторы должны при вычислении давать константу или адрес ранее описанного внешнего или статического объекта плюс или минус константа.

Меньшая широта допустима для константных выражений после **#if:** константы, описанные как имена, **sizeof** выражения и перечислимые константы недопустимы.

Глава 26.
Соображения мобильности

Определенные части **C++** являются машинно-зависимыми по своей сути.

Как показала практика, характеристики аппаратуры в чистом виде, такие, как размер слова, свойства плавающей арифметики и целого деления, не создают особых проблем. Другие аппаратные аспекты отражаются на различных программных разработках.

Некоторые из них, особенно знаковое расширение (преобразование отрицательного символа в отрицательное целое) и порядок расположения байтов в слове, являются досадными помехами, за которыми надо тщательно следить. Большинство других являются всего лишь мелкими сложностями.

Число регистровых переменных, которые фактически могут быть помещены в регистры, различается от машины к машине, как и множество фактических типов. Тем не менее, все компиляторы на «своей» машине все делают правильно; избыточные или недействующие описания **register** игнорируются.

Некоторые сложности возникают при использовании двусмысленной манеры программирования. Писать программы, зависящие от какой-либо из этих особенностей, районе неблагоразумно.

В языке не определен порядок вычисления параметров функции. На некоторых машинах он слева направо, а на некоторых справа налево.

Порядок появления некоторых побочных эффектов также недетерминирован.

Поскольку символьные константы в действительности являются объектами типа **int**, то могут быть допустимы многосимвольные константы. Однако конкретная реализация очень сильно зависит от машины, поскольку порядок, в котором символы присваиваются слову, различается от машины к машине. На некоторых машинах поля в слове присваиваются слева направо, на других справа налево.

Эти различия невидны для отдельных программ, не позволяющих себе каламбуров с типами (например, преобразования **int** указателя в **char** указатель и просмотр памяти, на которую указывает указатель), но должны приниматься во внимание при согласовании внешне предписанных форматов памяти.

Глава 27.
Свободная память

Операция **new** вызывает функцию

```
extern void* _new (long);
```

для получения памяти. Параметр задает число требуемых байтов. Память будет инициализирована. Если **_new** не может найти требуемое количество памяти, то она возвращает ноль.

Операция **delete** вызывает функцию

```
extern void _delete (void*);
```

чтобы освободить память, указанную указателем, для повторного использования. Результат вызова **_delete()** для указателя, который не был получен из **_new()**, не определен, это же относится и к повторному вызову **_delete()** для одного и того же указателя. Однако уничтожение с помощью **delete** указателя со значением ноль безвредно.

Предоставляются стандартные версии **_new()** и **_delete()**, но пользователь может применять другие, более подходящие для конкретных приложений.

Когда с помощью операции **new** создается классовый объект, то для получения необходимой памяти конструктор будет (неявно) использовать **new**.

Конструктор может осуществить свое собственное резервирование памяти посредством присваивания указателю **this** до каких-либо использований.

С помощью присваивания **this** значения ноль деструктор может избежать стандартной операции дерезервирования памяти для объекта его класса.

Например:

```
class cl
  {
  int v[10];
  cl () { this = my_own_allocator  (sizeof (cl)); }
  ~cl () { my_own_deallocator      (this); this = 0; }
  }
```

На входе в конструктор **this** является **не-нулем**, если резервирование памяти уже имело место (как это имеет место для автоматических объектов), и **нулем** в остальных случаях.

Если производный класс осуществляет присваивание **this**, то вызов конструктора (если он есть) базового класса будет иметь место после присваивания, так что конструктор базового класса ссылаться на объект посредством конструктора производного класса.

Если конструктор базового класса осуществляет присваивание **this**, то значение также будет использоваться конструктором (если таковой есть) производного класса.

Часть 2. Турбо С ++

Глава 1. Интегрированная среда разработки

TURBO C++ упрощает процесс программирования и делает его более эффективным. При работе в TURBO C++ весь комплекс инструментальных средств, необходимых для написания, редактирования, компиляции, компоновки и отладки программ, оказывается под рукой у пользователя.

Весь этот комплекс возможностей заключен в **Интегрированной Среде Разработки** (**ИСР**).

Кроме того, Среда разработки программ TURBO C++ предоставляет следующие дополнительные возможности, которые еще больше упрощают процесс написания программ:

- Возможность отображения на экране монитора значительного числа окон, которые можно перемещать по экрану и размеры которых можно изменять.
- Наличие поддержки «мыши».
- Наличие блоков диалога.
- Наличие команд удаления и вставки (при этом допускается копирование из окна **HELP** и между окнами **EDIT**).
- Возможность быстрого вызова других программ и обратного возврата.
- Наличие в редакторе макроязыка.

ИСР содержит три визуальных компоненты: **строку меню** у верхнего края экрана, **оконную область** в средней части экрана и **строку состояния** у нижнего края экрана. В результате выбора некоторых элементов меню на экран будут выдаваться блоки диалога.

Глава 2.
Строка меню и меню

Строка меню представляет собой основное средство доступа ко всем командам меню. Строка меню оказывается невидимой лишь во время просмотра информации, отображаемой программой и во время перехода к другой программе.

Глава 3.
Окна TURBO C++

Большая часть того, что видно и делается в среде TURBO C++, происходит в окне. Окно — это область экрана, которую можно перемещать, размеры которой можно перемещать, изменять, которую можно распахивать на весь экран, ориентировать встык с другими окнами.

В TURBO C++ может существовать произвольное число окон, но в каждый момент активно только одно окно. Активным является то окно, в котором в настоящий момент происходит работа.

Любые вводимые команды или вводимый текст, как правило, относятся только к активному окну.

Существует несколько типов окон, но большая их часть имеет следующие общие элементы:

- строку заголовка;
- маркер закрытия окна;
- полосы прокрутки;
- угол изменения размера окна;
- маркер распахивания окна на весь экран;
- номер окна.

Строка состояния

Строка состояния, расположенная у нижнего края экрана, выполняет следующие функции:

- Напоминает об основных клавишах и клавишах активизации, которые в настоящий момент могут быть применены к активному окну.

- Позволяет установить указатель мыши на обозначения клавиш и кратковременно нажать кнопку мыши для выполнения указанного действия, вместо того, чтобы выбирать команды из меню или нажимать соответствующие клавиши.
- Сообщает, какое действие выполняется программой.
- Предлагает состоящие из одной строки советы и рекомендации по любой выбранной команде меню и элементам блока диалога.

Блоки диалога

Если за элементом меню располагается многоточие, то в результате выбора данной команды будет открыт блок диалога, обеспечивающий удобный способ просмотра и задания многочисленных параметров.

При задании значения в блоке диалога работа происходит с пятью базовыми типами средств управления: указателями выбора, переключателями состояния, кнопками действия, блоками ввода и блоками списка.

Глава 4.
Работа с экранным меню

Меню (системное)

Отображается у левого края строки меню. Для вызова следует нажать **ALT+пробел**. При вызове этого меню отображаются команды:

- **About**

При выборе данной команды появляется блок диалога, в котором содержится информация по авторским правам и номер версии TURBO C++. Данное окно закрывается нажатием клавиши **ESC** или **ENTER**.

- **Clear Desktop**

Закрывает все окна и стирает все списки предысторий. Эта команда полезна в тех случаях, когда начинается работа над новым проектом.

- **Repaint Desktop**

Осуществляет регенерацию изображения на экране.

Элементы подменю Transfer

В этом подменю показаны имена всех программ, которые установлены с помощью блока диалога **Transfer**, вызываемого командой **Options/Transfer**. Для запуска программы необходимо выбрать ее имя из системного меню.

Меню File (ALT F)

Это меню позволяет открывать в окнах **EDIT** и создавать исходные файлы программ, сохранять внесенные изменения, выполнять другие действия над файлами, выходить в оболочку DOS и завершать работу с TURBO C++.

Open (F3)

Команда **FILE OPEN** отображает блок диалога, в котором выбирается исходный файл программы, который будет открыт в окне **EDIT**.

Этот блок диалога содержит блок ввода, список файлов, и кнопки **OPEN**, **REPLACE**, **CANCEL** и **HELP**, а также информационную панель.

Здесь можно выполнить одно из действий:

- Ввести полное имя файла и выбрать указатель **REPLACE** или **OPEN**.

В результате выбора **Open** файл загружается в новое окно **Edit**. Для выбора **Replace** должно иметься активное окно **Edit**; в результате выполнения **Replace** содержимое окна заменяется выбранным файлом.

- Ввести имя файла с метасимволами. Это позволяет отфильтровать список файлов в соответствии со спецификацией.
- Выбрать спецификацию файла из списка предыстории, который содержит введенные ранее спецификации файлов.
- Просмотреть содержимое других каталогов, выбрав имя каталога из списка файлов.

Блок ввода позволяет явно ввести имя файла или ввести имя файла с метасимволами **DOS** (***** и **?**). Если ввести имя полностью и нажать **Enter**, Turbo C++ откроет указанный файл. Если ввести имя файла, который система Turbo C++ не может обнаружить, она автоматически создаст и откроет новый файл с таким именем.

Если нажать **?**, когда курсор находится в блоке ввода, то под этим блоком появляется список предыстории, содержащий последние восемь имен файлов, которые были введены ранее.

New

Команда **File New** позволяет открывать новое окно **Edit** со стандартным именем **NONAMExx.C** (где вместо букв **xx** задается число в диапазоне от **00** до **99**).

Файлы с именем **NONAME** используются в качестве временного буфера для редактирования; когда файл с подобным именем сохраняется на диске, Turbo C++ запрашивает действительное имя файла.

Save (F2)

Команда **File Save** записывает на диск файл, находящийся в активном окне **Edit** (если активно окно **Edit** в настоящий момент, если нет, то данным элементом меню нельзя воспользоваться).

Если файл имеет использованное по умолчанию имя **(NONAMEOO.C** и т.п.) TurboC++ откроет блок диалога **Save Editor File**, который позволяет переименовать данный файл и сохранять его в другом каталоге или на другом дисководе.

Save As

Команда **File Save As** позволяет сохранить файл в активном окне **Edit** под другим именем, в другом каталоге или на другом дисководе.

Change Dir

Команда **File Change Dir** позволяет задать идентификатор и имя каталога, которые следует сделать текущими. Текущим является тот каталог, который используется в Turbo C++ для сохранения и поиска файлов. При использовании относительных маршрутов в **Options Directories** они задаются только относительно текущего каталога.

Print

Команда **File Print** печатает содержимое активного окна **Edit** Turbo C++ «раскрывает» символы табуляции (заменяет их соответствующим числом пробелов), а затем посылает файл на устройство печати, заданное в DOS.

Данная команда будет «запрещена», если содержимое активного окна не может быть выведено на печать. Для вывода на печать только выделенного текста следует использовать **Ctrl-K P**.

Get Info

Команда **File Get Info** отображает блок, в котором содержится информация относительно текущего файла.

DOS Shell

Команда **File DOS Shell** позволяет временно выйти из Turbo C++, чтобы выполнить команду DOS или запустить программу. Для возврата в Turbo C++ необходимо ввести с клавиатуры **EXIT** и нажать **Enter**.

Иногда можно обнаружить, что во время отладки не хватает памяти для выполнения этой команды. В этом случае необходимо завершить сеанс отладки командой **Run Program Reset** (**Ctrl-F2**).

Quit (Alt-x)

Команда **File Quit** осуществляет выход из системы Turbo C++, удаляет ее из памяти и передает управление DOS. Если внесены изменения, которые еще не были сохранены, то перед выходом Turbo C++ выдаст запрос на их сохранение.

Значения блока Get Info

Current directory

Имя каталога по умолчанию.

Current file

Имя файла в активном окне.

Extended memory usage

Объем дополнительной памяти, зарезервированной для Turbo C++.

Expanded memory usage

Объем расширенной памяти, зарезервированной для Turbo C++.

Lines compiled

Число откомпилированных строк.

Total warnings

Число выданных системой предупреждающих сообщений.

Totals errors

Число сгенерированных ошибок.

Total time

Время последнего выполнения программы.

Program loaded

Статус отладки.

Program exit

Код возврата от последней завершившейся программы.

Available memory

Объем доступной памяти DOS (640 K).

Last step time

Время выполнения последнего шага отладки.

Меню Edit (Alt-E)

Позволяет выполнять удаления, копирование и вставку текста в окнах **Edit**. Можно также открыть окно текстового буфера для просмотра или редактирования его содержимого. Выбрать текст это значит выделить его цветом:

- Нажать **Shift** с одновременным нажатием стрелки.
- Нажать **Ctrl-K B**, чтобы пометить начало выделяемого блока. Затем переместить курсор в конец фрагмента текста и нажать **Ctrl-K K**.
- Для выбора строки необходимо нажать **Ctrl-K L**. После выделения фрагмента текста становятся доступными команды, из меню **Edit**, и можно использовать текстовый буфер (**Clipboard**). Он взаимодействует с командами меню **Edit**.

Restore Line

Эта команда отменяет действие последней команды редактирования, примененной к какой-либо строке. Она действует только над последней отредактированной строкой.

Cut (Shift-Del)

Удаляет выделенный фрагмент текста из документа и заносит его в текстовый буфер. Затем можно вставить текст в другой документ путем выбора **Paste**.

Copy (Ctrl-Ins)

Эта команда не изменяет выделенный текст, но заносит в текстовый буфер его точную копию. Затем можно вставить текст в другой документ командой **Paste**. Можно скопировать текст из окна **Help**; следует использовать **Shift** и клавиши управления курсором.

Paste (Shift-Ins)

Эта команда вставляет текст из текстового буфера в текущее окно в позиции курсора.

Show Clipboard

Эта команда открывает окно **Clipboard**, в котором хранятся фрагменты текста, удаленного и скопированного из других окон.

Clear (Ctrl-Del)

Эта команда удаляет выбранный фрагмент текста, но не заносит его в текстовый буфер. Это означает, что восстановить удаленный текст нельзя.

Меню Search (Alt-S)

Меню **Search** выполняет поиск текста, объявлений функций, а также местоположение ошибок в файлах.

Search Find

Команда **Search Find** отображает блок диалога **Find**, который позволяет ввести образец поиска и задать параметры, влияющие на процесс поиска.

Эта команда может быть также вызвана с помощью **Ctrl-Q-F**.

Replace (Ctrl Q A)

Команда **Search Replace** отображает блок диалога для ввода искомого текста и текста, на который его следует заменить.

Search Again (Ctrl L)

Команда **Search Again** повторяет действие последней команды **Find** или **Replace**. Все параметры, которые были заданы при последнем обращении к использованному блоку диалога (**Find** или **Replace**), остаются действительными при выборе данной команды.

Меню Run (Alt-R)

Команды этого меню выполняют программу, а также инициализируют и завершают сеанс отладки.

Run (Ctrl-F9)

Команда **Run** выполняет программу, используя те аргументы, которые переданы программе с помощью команды **Run Arguments**.

Trace Into (F7)

Эта команда выполняет программу по операторам. По достижению вызова функции будет выполняться каждый ее оператор вместо того, чтобы выполнить эту функцию за один шаг. Этой командой следует пользоваться для перемещения выполнения в функцию, которая вызывается из отлаживаемой функции.

Program Reset (Ctrl-F2)

Команда **Run Program Reset** прекращает текущий сеанс отладки, освобождает память программы и закрывает все открытые файлы, которые использовались в программе.

Over

Команда **Run Step Over** выполняет следующий оператор в текущей функции без вхождения в функции более низкого уровня, даже если эти функции доступны отладчику.

Командой **Step Over** следует пользоваться в случаях, когда необходимо отладить функцию в пооператорном режиме выполнения без вхождения в другие функции.

Arguments

Команда **Run Arguments** позволяет задать выполняемой программе аргументы командной строки точно так же, как если бы они вводились в командной строке DOS. Команды переназначения ввода/вывода DOS будут игнорироваться.

Меню Compile (C)

Команды из меню **Compile** используются для компиляции программы в активном окне, а также для полной или избирательной компиляции проекта.

EXE File

Команда **Compile Make EXE File** вызывает Менеджер проектов для создания **EXE**-файла.

Link EXE File (только при полном наборе меню)

Команда **Compile Link EXE File** использует текущие **OBJ** и **LIB**-файлы и компонует их, не производя избирательной компиляции.

Меню Debug (Alt F9)

Команды меню **Debug** управляют всеми возможностями интегрированного отладчика.

Inspect (Alt F4)

Команда **Debug Inspect** открывает окно **Inspector**, которому позволяет проанализировать и модифицировать значения элемента данных.

Меню Options (Alt-O)

Меню **Options** содержит команды, которые позволяют просматривать и модифицировать стандартные параметры, определяющие функционирование Turbo C++.

Глава 5. Структура файла, типы данных и операторов ввода-вывода

Функция Main

Каждый исполняемый файл системы Турбо C++ (программа) должен содержать функцию **main**.

Код, задающий тело функции **main**, заключается в фигурные скобки **{и}**.

Общая структура функции **main** такова:

```
main()
{
/* Код, реализующий main */
}
```

Комментарии

Текст на Турбо C++, заключенный в скобки /* и */, компилятором игнорируется.

Комментарии служат двум целям: документировать код и облегчить отладку. Если программа работает не так, как надо, то иногда оказывается полезным закомментировать часть кода (т.е. вынести ее в комментарий), заново скомпилировать программу и выполнить ее.

Если после этого программа начнет работать правильно, то значит, закомментированный код содержит ошибку и должен быть исправлен.

Директивы Include

Во многие программы на Турбо C++ подставляются один или несколько файлов, часто в самое начало кода главной функции **main**.

Появление директив

```
#include <файл_1>
#include "файл_2"
...
#include <файл_n>
```

приводит к тому, что препроцессор подставляет на место этих директив тексты файлов **файл_1**, **файл_2** , **...** , **файл_n** соответственно.

Если имя файла заключено в угловые скобки **<...>**, то поиск файла производится в специальном разделе подстановочных файлов. В отличие от многих других операторов Турбо C++ директива **Include** не должна оканчиваться точкой с запятой.

Макро

С помощью директивы **#define**, вслед за которой пишутся имя макро и значение макро, оказывается возможным указать препроцессору, чтобы он при любом появлении в исходном файле на Турбо C++ данного имени макро заменял это имя на соответствующие значения макро.

Например, директива

```
#define pi 3.1415926
```

связывает идентификатор **pi** со значением **3.1415926**. После значения макро **(;)** не ставится.

Типы данных

В Турбо C++ переменные должны быть описаны, а их тип специфицирован до того, как эти переменные будут использованы.

При описании переменных применяется префиксная запись, при которой вначале указывается тип, а затем — имя переменной.

Например:

```
float weight;
int exam_score;
char ch;
```

С типом данных связываются и набор предопределенных значений, и набор операций, которые можно выполнять над переменной данного типа.

Переменные можно инициализировать в месте их описаний.

Пример:

```
int height = 71 ;
float income =26034.12 ;
```

Простейшими скалярными типами, предопределёнными в Турбо С++, являются

- **char** — представляется как однобайтовое целое число
- **int** — двубайтовое целое
- **long** — четырёхбайтовое целое
- **float** — четырёхбайтовое вещественное
- **double** — восьмибайтовое вещественное

Оператор printf: вывод на терминал

Функцию **printf** можно использовать для вывода любой комбинации символов, целых и вещественных чисел, строк, беззнаковых целых, длинных целых и беззнаковых длинных целых.

Пример:

```
printf("\nВозраст Эрика - %d. Его доход $%.2f",age,income);
```

Предполагается, что целой переменной **age** (возраст) и вещественной переменной **income** (доход) присвоены какие-то значения.

Последовательность символов «**\n**» переводит курсор на новую строку.

Последовательность символов «**Возраст Эрика**» будет выведена с начала новой строки. Символы **%d** — это спецификация для целой переменной **age**.

Следующая литерная строка «**Его доход $**». **%2f** — это спецификация (символ преобразования формата) для вещественного значения, а также указание формата для вывода только двух цифр после десятичной точки. Так выводится значение переменной **income**.

Символ формата Тип выводимого объекта

%c **char**

%s строка

%d **int**

%o **int** (в восьмеричном виде)

%u **unsigned int**

%x **int** (в шестнадцатеричном виде)

%ld **long** (в десятичном виде)

%lo **long** (в восьмеричном виде)

%lu **unsigned long**

%lx **long** (в шестнадцатеричном виде)

%f **float**/double (с фиксированной точкой)

%e **float**/double (в экспоненциальной форме)

%g **float**/double (в виде **f** или **e** в зависимости от значения)

%lf **long float** (с фиксированной точкой)

%le **long float** (в экспоненциальной форме)

%lg **long float** (в виде **f** или **e** в зависимости от значения)

Оператор scanf: ввод с клавиатуры

Оператор **scanf** является одной из многих функций ввода, имеющихся во внешних библиотеках.

Каждой вводимой переменной в строке функции **scanf** должна соответствовать спецификация. Перед именами переменных необходимо оставить символ **&**. Этот символ означает «**взять адрес**».

Пример:

```
#include<stdio.h>
main()
{
int weight, /*вес*/ height; /*рост*/
printf(" Введите ваш вес: ");
scanf("%d", &weight);
printf(" Введите ваш рост: ");
scanf("%d", &height);
```

```
printf("\n\nВес = %d, рост = %d\n",
weight,height);
}
```

Глава 6.
Арифметические, логические операции и операции отношения и присваивания

Основу языка Турбо С++ составляют операторы. Оператором выражения называют выражение, вслед за которым стоит точка с запятой. В Турбо С++ точки с запятой используются для разделения операторов. Принято группировать все операторы в следующие классы:

- присваивания,
- вызов функции,
- ветвления,
- цикла.

В операторе присваивания используется операция присваивания **=.**

Например:

```
c = a * b;
```

Действие такого оператора можно описать следующими словами: «с присваивается значение **a**, умножение на **b**». Значение, присваиваемое переменной **c**, равняется произведению текущих значений переменных **a** и **b**.

Операторы часто относятся более чем к одному из четырех классов.

Например, оператор

```
if ( ( c = cube( a * b ) ) > d )
...
```

составлен из представителей следующих классов: присваивания, вызов функции, и ветвление.

К понятию оператора вплотную примыкает понятие **операции**.

Различают следующие группы операций Турбо С++:

- арифметические операции

- операции отношения
- операции присваивания
- логические операции
- побитовые операции
- операция вычисления размера (sizeof)
- операция следования (запятая).

Арифметические операции

К арифметическим операциям относятся:

- сложение (+)
- вычитание (-)
- деление (/)
- умножение (*)
- остаток (%).

Все операции (за исключением остатка) определены для переменных типа **int**, **char**, **float**. Остаток не определен для переменных типа **float**. Все арифметические операции с плавающей точкой производятся над операндами двойной точности.

Операции отношения

В языке определены следующие операции отношения:

- проверка на равенство (==)
- проверка на неравенство (!=)
- меньше (<)
- меньше или равно (<=)
- больше (>)
- больше или равно (>=).

Все перечисленные операции вырабатывают результат типа **int**. Если данное отношение между операндами истинно, то значение этого целого — единица, а если ложно, то нуль.

Все операции типа больше-меньше имеют равный приоритет, причем он выше, чем приоритет операций **==** и **!=**. Приоритет операции

присваивания ниже приоритета всех операций отношений. Для задания правильного порядка вычислений используются скобки.

Логические операции

В языке имеются три логические операции:

- && операции И (and)
- || операции ИЛИ (or)
- ! отрицание

Аргументами логических операций могут быть любые числа, включая задаваемые аргументами типа **char**. Результат логической операции — единица, если истина, и нуль, если ложь. Вообще все значения, отличные от нуля, интерпретируются как истинные.

Логические операции имеют низкий приоритет, и поэтому в выражениях с такими операциями скобки используются редко.

Вычисление выражений, содержащих логические операции, производится слева направо и прекращается (усекается), как только удается определить результат.

Если выражение составлено из логических утверждений (т.е. выражения, вырабатывающие значения типа **int**), соединенных между собой операцией **И** (**&&**), то вычисление выражения прекращается, как только хотя бы в одном логическом утверждении вырабатывается значение нуль.

Если выражение составлено из логических утверждений, соединенных между собой операцией **ИЛИ** (||), то вычисление выражения прекращается, как только хотя бы в одном логическом утверждении вырабатывается ненулевое значение.

Вот несколько примеров, в которых используются логические операции:

```
if( i > 50 && j == 24)
   ...
   if( value1 < value2 && (value3 > 50 || value4 < 20) )
   ...
```

Операции присваивания

К операциям присваивания относятся **=, +=, -=, *=** и **/=,** а также **префиксные** и **постфиксные** операции **++** и **--**.

Все операции присваивания присваивают переменной результат вычисления выражения. Если тип левой части присваивания отличается от типа правой части, то тип правой части приводится к типу левой.

В одном операторе операция присваивания может встречаться несколько раз. Вычисления производятся справа налево.

Например:

```
a = ( b = c ) * d;
```

Вначале переменной **d** присваивается значение **c**, затем выполняется операция умножения на **d**, и результат присваивается переменной **a**.

Операции **+=, -=, *=** и **/=** являются укороченной формой записи операции присваивания. Их применение проиллюстрируем при помощи следующего описания:

```
a += b означает a = a + b
a -= b означает a = a – b
a *= b означает a = a * b
a /= b означает a = a / b
```

Префиксные и постфиксные операции **++** и **--** используют для увеличения (инкремент) и уменьшения (декремент) на единицу значения переменной.

Семантика указанных операций следующая:

- **++a** — увеличивает значение переменной **a** на единицу до использования этой переменной в выражении.
- **a++** — увеличивает значение переменной **a** на единицу после использования этой переменной в выражении.
- **--a** — уменьшает значение переменной **a** на единицу до использования этой переменной в выражении.
- **a--** — уменьшает значение переменной **a** на единицу после использования этой переменной в выражении.

Операцию **sizeof** (размер) можно применить к константе, типу или переменной. В результате будет получено число байтов, занимаемых операндом.

Например:

```
printf ( "\nРазмер памяти под целое %d", sizeof( int) );
printf ( "\nРазмер памяти под символ %d", sizeof( char) );
```

Глава 7.
Логическая организация программы и простейшее использование функций

Процесс разработки программного обеспечения предполагает разделение сложной задачи на набор более простых задач и заданий. В Турбо С++ поддерживаются функции как логические единицы (блоки текста программы), служащие для выполнения конкретного задания. Важным аспектом разработки программного обеспечения является функциональная декомпозиция.

Функции имеют нуль или более формальных параметров и возвращают значение скалярного типа, типа **void** (пусто) или указатель. При вызове функции значения, задаваемые на входе, должны соответствовать числу и типу формальных параметров в описании функции.

Если функция не возвращает значения (т.е. возвращает **void**), то она служит для того, чтобы изменять свои параметры (вызывать побочный эффект) или глобальные для функции переменные.

Например, функция, возвращающая куб ее вещественного аргумента:

```
double cube( double x )
{
return x * x * x ;
}
```

Аргумент **x** типа **double** специфицируется вслед за первой открывающей скобкой. Описание **extern**, помещаемое в функцию **main**, является ссылкой вперед, позволяющей использовать функцию **cube** в функции **main**. Ключевое слово **extern** можно опускать, но сама ссылка вперед на описание функции является обязательной.

Глава 8.
Логическая организация простой программы

Турбо С++ предоставляет необычайно высокую гибкость для физической организации программы или программной системы.

Структура каждой функции совпадает со структурой главной программы (**main**). Поэтому функции иногда еще называют подпрограммами. Подпрограммы решают небольшую и специфическую часть общей задачи.

Глава 9.
Использование констант различных типов

В языке Турбо С++ имеются четыре типа констант:

- целые
- вещественные (с плавающей точкой)
- символьные
- строковые.

Константы целого типа

Константы целого типа могут задаваться в десятичной, двоичной, восьмеричной или шестнадцатеричной системах счисления.

Десятичные целые константы образуются из цифр. Первой цифрой не должен быть нуль.

Восьмеричные константы всегда начинаются с цифры нуль, вслед за которой либо не стоит ни одной цифры, либо стоят несколько цифр от нуля до семерки.

Шестнадцатеричные константы всегда начинаются с цифры нуль и символа **x** или **X**, все, за которыми может стоять одна или более шестнадцатеричных цифр.

Шестнадцатеричные цифры — это десятичные цифры от **0** до **9** и латинские буквы: **a, b, c, d, e, f**, или **A, B, C, D, E, F**.

Например: задание константы **3478** в десятичном, восьмеричном и шестнадцатеричном виде:

```
int a = 3478,
b = 06626,
c = 0xD96;
```

К любой целой константе можно справа приписать символ **l** или **L**, и это будет означать, что константа — длинная целая (**long integer**). Символ **u** или **U**, приписанный к константе справа, указывает на то, что константа целая без знака (**unsigned long**).

Считается, что значение любой целой константы всегда неотрицательно. Если константе предшествует знак минус, то он трактуется как операция смены знака, а не как часть константы.

Константы вещественного типа

Константы с плавающей точкой (называемые вещественными) состоят из цифр, десятичной точки и знаков десятичного порядка **e** или **E**.

```
1.  2e1     .1234  .1e3
.1  2E1     1.234  0.0035e-6
1.0         2e-1     2.1e-12  .234
```

Символьные константы

Символьные константы заключаются в апострофы (кавычки). Все символьные константы имеют в Турбо C++ значение типа **int** (целое), совпадающее с кодом символа в кодировке **ASCII**.

Одни символьные константы соответствуют символам, которые можно вывести на печать, другие — управляющим символам, задаваемым с помощью **esc**-последовательности, третьи — форматирующими символами, также задаваемым с помощью **esc**-последовательности.

Например:

- символ «апостроф» задается как '\''
- переход на новую строку — как '\'
- обратный слэш — как '\\'

Каждая **esc**-последовательность должна быть заключена в кавычки.

Управляющие коды

- **\n** — Новая строка
- **\t** — Горизонтальная табуляция
- **\v** — Вертикальная табуляция
- **\b** — Возврат на символ
- **\r** — Возврат в начало строки
- **\f** — Прогон бумаги до конца страницы
- \\ — Обратный слэш

- **\'** — Одинарная кавычка
- **\"** — Двойная кавычка
- **\a** — Звуковой сигнал
- **\?** — Знал вопроса
- **\ddd** — Код символа в **ASCII** от одной до трех восьмеричных цифр
- **\xhhh** — Код символа в **ASCII** от одной до трех шестнадцатеричных цифр.

Строковые константы

Строковые константы состоят из нуля или более символов, заключенных в двойные кавычки. В строковых константах управляющие коды задаются с помощью **esc**-последовательности. Обратный слэш используется как символ переноса текста на новую строку.

Пример описания строковых констант:

```
# include <stdio.h>
main( )
{
char *str1, *str2;
str1=" Пример использования\n\n";
str2="строковых\ констант.\n\n";
printf(str1);
printf(str2);
}
```

Глава 10. Управляющие структуры

Управляющие структуры или операторы управления служат для управления последовательностью вычислений в программе. Операторы ветвления и циклы позволяют переходить к выполнению другой части программы или выполнять какую-то часть программы многократно, пока удовлетворяется одно или более условий.

Блоки и составные операторы

Любая последовательность операторов, заключенная в фигурные скобки, является составным оператором (блоком). Составной оператор не должен заканчиваться **(;)**, поскольку ограничителем блока служит са-

ма закрывающаяся скобка. Внутри блока каждый оператор должен ограничиваться **(;)**.

Составной оператор может использоваться везде, где синтаксис языка допускает применение обычного оператора.

Пустой оператор

Пустой оператор представляется символом **(;)**, перед которым нет выражения. Пустой оператор используют там, где синтаксис языка требует присутствия в данном месте программы оператора, однако по логике программы оператор должен отсутствовать.

Необходимость в использовании пустого оператора часто возникает, когда действия, которые могут быть выполнены в теле цикла, целиком помещаются в заголовке цикла.

Операторы ветвления

К операторам ветвления относятся **if**, **if else**, **?**, **switch** и **go to**. Общий вид операторов ветвления следующий:

```
if (логическое выражение)
оператор;
-----------------------------------------
if (логическое выражение)
оператор_1;
else
оператор_2;
-----------------------------------------
<логическое выражение> ? <выражение_1> : <выражение_2>;
Если значение логического выражения истинно, то вычисляется
выражение_1, в противном случае вычисляется выражение_2.
-----------------------------------------
switch (выражение целого типа)
{
case значение_1:
последовательность_операторов_1;
break;
case значение_2:
последовательность_операторов_2;
break;
. . .
case значение_n:
последовательность_операторов_n;
break;
```



```
default:
последовательность_операторов_n+1;
}
```

Ветку **default** можно не описывать. Она выполняется, если ни одно из вышестоящих выражений не удовлетворено.

Оператор цикла

В Турбо С++ имеются следующие конструкции, позволяющие программировать циклы: **while**, **do while** и **for**. Их структуру можно описать следующим образом:

```
while( логическое выражение)
  оператор;
Цикл с проверкой условия наверху
---------------------------
do
  оператор;
while (логическое выражение);
Цикл с проверкой условия внизу
---------------------------
for (инициализация, проверка, новое_значение)
оператор;
---------------------------
```

Глава 11. Приемы объявления и обращения к массивам, использование функций и директивы define при работе с массивами

Массивы — это набор объектов одинакового типа, доступ к которым осуществляется прямо по индексу в массиве. Обращение к массивам в Турбо С++ осуществляется и с помощью указателей.

Массивы можно описывать следующим образом:

```
тип_данных имя_массива [размер массива];
```

Используя имя массива и индекс, можно адресоваться к элементам массива:

```
имя_массива [значение индекса]
```

Значения индекса должны лежать в диапазоне от нуля до величины, на единицу меньшей, чем размер массива, указанный при его описании.

Вот несколько примеров описания массивов:

```
char name [ 20 ];
int grades [ 125 ];
float income [ 30 ];
double measurements [ 1500 ];
```

Первый из массивов (**name**) содержит 20 символов.

Обращением к элементам массива может быть **name [0]**, **name [1]**, **..., name [19]**.

Второй массив (**grades**) содержит 125 целых чисел. Обращением к элементам массива может быть **grades [0]**, **grades [1]**, **...**, **grades [124]**.

Третий массив (**incom**) содержит 30 вещественных чисел. Обращением к элементам массива может быть **income [0]**, **incom [1]**, **...**, **income [29]**.

Четвертый массив (**measurements**) содержит 1500 вещественных чисел с двойной точностью. Обращением к элементам массива может быть **measurements [0]**, **measurements [1]**, **...**, **measurements [1499]**.

Вот программа, иллюстрирующая использование массивов (Файл **array.c**):

```
#include <stdio.h>
#define size 1000
int data [size];
main ( )
{
extern float average (int a[], int s );
int i;
for ( i=0; i<size ; i++_)
data [ i ]= i;
printf ( "\nСреднее значение массива data =%f\n",average
(data,size));
}
float average (int a[ ] ,int s )
{
 float sum=0.0;
 int i;
 for ( i=0; i<s ; i ++)
 sum+=a[ i ];
 return sum/s;
}
```


В программе заводится массив на 1000 целых чисел. При помощи функции **average** подсчитывается сумма элементов этого массива.

Первым формальным параметром функции **average** является массив. В качестве второго параметра функции передается число суммируемых значений в массиве **a**.

Обратите внимание на использование константы **size** (размер). Если изменяется размерность массива, задаваемая этой константой, то это не приводит к необходимости менять что-либо в самом коде программы.

Часть 3. От теории к практике

Глава 1. Правило «право-лево»

Существенный принцип анализа сложных синтаксических конструкций языка, вроде «указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие значение **int**» чётко формализован в виде правила «право-лево». Всё предельно просто. Имеем:

- **()** — функция, возвращающая...
- **[]** — массив из...
- ***** — указатель на...

Первым делом находим имя, от которого и будем плясать.

Следующий шаг — шаг вправо. Что там у нас справа? Если **()**, то говорим, что «Имя есть функция, возвращающая...». (Если между скобок что-то есть, то «Имя есть функция, принимающая то, что между скобок, и возвращающая...»). Если там **[]**, то «Имя есть массив из...». И подобным вот образом мы идём вправо до тех пор, пока не дойдём до конца объявления или правой «**)**» скобки. Тут тормозим...

...и начинаем танцевать влево. Что у нас слева? Если это что-то не из приведенного выше (то есть не **()**, **[]**, *****), то попросту добавляем к уже существующей расшифровке. Если же там что-то из этих трёх символов, то добавляем то, что написано выше. И так танцуем до тех пор, пока не дотанцуем до конца (точнее — начала объявления) или левой «**(**» скобки. Если дошли до начала, то всё готово. А если дошли до «**(**», то по уже означенной итеративности переходим к шагу «Пляски вправо» и продолжаем.

Пример:

```
int (*(*(*fptr)())[3])();
```

Находим имя и записываем «**fptr есть...**».

Шаг вправо, но там «**)**», потому идём влево:

```
int (*(*(*fun)())[3])();
```

и получаем «**fptr есть указатель на...**».

Продолжаем ехать влево, но тут «**(**». Идём вправо:

```
int (*(*(*fun)())[3])();
```

получаем «**fptr есть указатель на функцию, возвращающую...**» Снова «**)**», опять влево. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на...**» Слева опять «**(**», идём вправо. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх...**» И снова справа «**)**», отправляемся влево. Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на...**» Снова разворот вправо по причине «**(**». Получаем:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие...**» Тут конец описания, поехали влево и получили окончательную расшифровку:

```
int (*(*(*fun)())[3])();
```

«**fptr есть указатель на функцию, возвращающую указатель на массив из трёх указателей на функции, возвращающие int**».

Именно то, чего мы хотели.

Глава 2. STLport

STLport — это свободно распространяемая реализация стандартной библиотеки шаблонов для множества различных компиляторов и операционных систем. Помимо всего прочего, STLport доступен не только для современных компиляторов, более или менее удовлетворяющих стандарту языка, но и для некоторых старых компиляторов, в частности Borland C++ 5.02 или MS Visual C++ 4.0.

Четвертая версия STLport отличается от предыдущей главным образом тем, что теперь в нее входит полная поддержка потоков (ранее приходилось использовать потоки из библиотеки, поставляемой с кон-

кретным компилятором). Реализация потоков взята из SGI (как, впрочем, и весь STLport). Вообще, STLport начал развиваться как попытка перенести известную библиотеку SGI STL на gcc и sun cc. Таким образом, с выходом четвертой версии, STLport стал полноценной библиотекой, соответствующей стандарту языка, во всяком случае, у него появились претензии на это.

Понятно, что применение одной и той же библиотеки на разных платформах, это уже большой плюс — потому что никогда точно заранее не известно, что, где и как будет плохо себя вести. Можно только лишь гарантировать, что программа, при переносе с одного компилятора на другой, все-таки будет себя плохо вести даже в том случае, если скомпилируется. Использование одной библиотеки шаблонов очень сильно увеличивает шансы на то, что не будет проблем тогда, когда программист увидит отсутствие в STL нового компилятора какого-нибудь контейнера. К примеру, в g++-stl-3 нет std::wstring. То есть, шаблон std::basic_string есть, и std::string является его инстанционированием на char, но попытка подставить туда же wchar_t ни к чему хорошему не приведет (в частности, из-за того, что в методе c_str() есть исключительная строчка вида return "").

Но и кроме единых исходных текстов у STLport есть еще несколько интересных возможностей и особенностей. Во-первых, это **debug mode**, при котором проверяются все условия, которые только возможны. В частности, в этом режиме при попытке работать с неинициализированным итератором будет выдано соответствующее ругательство. Согласитесь, это удобно.

Во-вторых, в STLport есть несколько нестандартных контейнеров, таких как **hash_map**, например. Зачем? Ну, в силу того что стандартный **map** как правило реализован на сбалансированных деревьях поиска (как более общий способ обеспечения быстрого поиска при разнородных данных), и что делать в том случае, когда все-таки известна хорошая хеш-функция для определенных элементов, не особенно понятно (ну, за исключением того, чтобы написать подобный контейнер самостоятельно).

В третьих, поддержка многопоточности. То есть, STLport можно безопасно использовать в программах, у которых более одного потока выполнения. Это досталось STLport еще от SGI STL, в которой очень много внимания уделялось именно безопасности использования.

Помимо того, если вдруг возникли какие-то проблемы с STL, то можно попытаться взять STLport — быть может, проблем станет поменьше.

Глава 3.
Язык программирования от Microsoft: C#

Фирма Microsoft создала новый язык программирования, сделанный на основе C и C++ и Java, который она назвала C# (C sharp).

Чтобы реально посмотреть язык программирования, возьмем программу «Hello, world!» из C# Language Reference:

```
using System;
class Hello
{
 static void Main() {
  Console.WriteLine("Hello, world");
 }
}
```

Это очень сильно напоминает Java. Таким образом, что имеется в наличии:

- Убрали селектор **->**, впрочем это, возможно и правильно: и «точка» и «стрелка» выполняют, в принципе, одни и те же функции с точки зрения ООП, так что в этом есть намеки на концептуальность. В принципе, это стало вероятным благодаря тому, что в C# есть типы «значения» (такие как, **int**, **char**, структуры и перечисления) и типы «ссылки», которыми являются объекты классов, массивы.
- Точно так же, как и в Java, перенесли метод **main** внутрь класса.
- Точно так же, как и в Java, в программах на C# теперь нет необходимости в декларациях без дефиниций, т.е. компилятор многопроходный.
- Конечно же, не смогли обойтись без автоматического сборщика мусора, так что в C#, так же как и в Java, не требуется беспокоиться об удалении памяти из-под объектов. Тем не менее, введена такая возможность, под названием «**unsafe code**», используя которую можно работать с указателями напрямую.
- Появился тип **object** с понятными последствиями: все типы (включая типы «значения») являются потомками **object**.

- Между **bool** и **integer** нет кастинга по умолчанию. Тип **char** — это **Unicode** символ (так же, как и в Java).
- Есть поддержка настоящих многомерных массивов (а не массивов массивов).

В отличие от Java, в C# выжил оператор **goto**.

Появился оригинальный оператор **foreach**:

```
static void WriteList(ArrayList list) {
 foreach (object o in list)
 Console.WriteLine(o);
}
```

который позволяет обойти контейнер.

Есть еще два интересных оператора: **checked** и **unchecked**. Они позволяют выполнять арифметические операции с проверкой на переполнение и без него.

Существует поддержка многопоточности при помощи оператора **lock**.

Отсутствует множественное наследование — вместо него, как и в Java, добавлена поддержка интерфейсов. Кстати сказать, структуры теперь совсем не тоже самое, что и классы. В частности, структуры не могут быть наследованы.

Добавлена поддержка свойств (property).

На языковом уровне введена поддержка отклика на события.

Введены определяемые пользователями атрибуты для поддержки систем автодокументации.

Кардинальное отличие от Java — наличие компилятора в машинный код. То есть, можно предположить, что программы на C# будут выполняться несколько быстрее, чем написанные на Java.

Вообще, можно говорить о том, что Microsoft учла традиционные нарекания в сторону Java в своем новом языке. В частности, оставлена от C++ перегрузка операторов.

Компания Microsoft утверждает, что создала язык для написания переносимых web-приложений и старается всячески показать свою собственную активность в этом направлении. В частности, компания Microsoft направила запрос на стандартизацию C#.

В принципе, ясно, зачем все это нужно. Компании Microsoft, несомненно, понадобился свой язык программирования такого же класса, как и Java. Пускать же Java к себе в Microsoft никто не намеревался, вот и получился C#. Понятно, что в данном случае язык программирования сам по себе представляет достаточно малую ценность, в силу того что Java хороша своей переносимостью, а переносимость ей гарантирует мощная и обширная стандартная библиотека, употребляя которую нет надобности вызывать какие-то системно- или аппаратно-зависимые фрагменты кода. Поэтому на текущий момент ничего определенного сказать о судьбе C# нельзя — хотя бы потому, что у него пока что нет подобной библиотеки.

Все же, в ближайшие несколько лет будет очень интересно следить за развитием C# и Java. В принципе, еще недавно представлялось, что уже невозможно вытеснить Java из своей ниши инструмента для относительно простого создания переносимых приложений, но вот, Microsoft решилась на эту попытку. Учитывая то, что в свое время было очевидно главенство Netscape на рынке броузеров, ожидать можно всего.

Глава 4. C++ Builder

В первую очередь оговоримся, что здесь мы будем рассматривать C++ Builder именно как «builder», т.е. программный инструмент класса RAD (Rapid Application Development, быстрое создание приложений) и, в общем-то, большая часть здесь написанного в одинаковой степени применимо ко всем подобным средствам.

Итак, C++ Builder облегчает процесс создания программ для ОС Windows с графическим интерфейсом пользователя. При его помощи одинаково просто создать диалог с тремя кнопочками «Yes», «No», «Cancel» или окно текстового WYSIWYG редактора с возможностью выбора шрифтов, форматирования, работы с файлами формата rtf. При этом C++ Builder автоматически создает исходный текст для пользовательского интерфейса: создает новые классы, объекты, добавляет необходимые переменные и функции. После всего этого «рисование» пользовательского интерфейса превращается, буквально, в наслаждение для эстетов: сюда добавим градиент, здесь цвет изменим, тут шрифт поменяем, а сюда мы поместим картинку.

После того, как вся эта красота набросана, наступает менее привлекательная работа — написание функциональной части. Тут C++ Builder ничем помочь не может, все приходиться делать по старинке, позабыв про «манипулятор мышь» и касаясь исключительно клавиатуры.

Итог?.. Как обычно: красота неписанная на экране. Этих программ, которые рисовали эстетствующие программисты в настоящее время видимо-невидимо, ими можно любоваться, распечатывать картинки с экрана и делать из них художественные галереи...

Что же тут плохого? Ничего, если не считать того, что при таком подходе к программированию создание программного продукта начинает идти не от его «внутренностей» (функционального наполнения), а от пользовательского интерфейса и в итоге получается, если «наполнение» достаточно сложное (сложнее передачи текста от одного элемента пользовательского интерфейса другому), то оно становится не только системнозависимым, но и компиляторозависимым, что уж абсолютно неприятно.

Кроме того, простота «рисования» пользовательского интерфейса, а, вернее, ненаказуемость (например, объемным программированием) использования всевозможных сложных компонентов скрывает в себе некоторые опасности. Связано это с тем, что создание удобного пользовательского интерфейса это задача сама по себе довольно трудная и требующая особенного образования. При этом всякий уважающий себя программист всегда уверен в том, что уж он-то точно сможет сделать пользовательский интерфейс предельно удобным и красивым.

Почему настолько разительно отличаются домашние страницы и страницы профессиональных web-дизайнеров? Вследствие того что последние имеют очень много узкоспециализированных знаний о восприятии человеком информации на экране монитора и, благодаря этому, могут разместить информацию не «красиво», а так, как это будет удобным. То же самое и с пользовательским интерфейсом — вопрос о том, как должна выглядеть конкретная кнопка и в каком месте она должна находиться не так прост, как кажется. Вообще, дизайнер пользовательского интерфейса это совершенно исключительная профессия, которая, к сожалению, у нас еще не распространена.

Тот факт, что функциональное наполнение становится зависимым от используемой библиотеки пользовательского интерфейса, просто абсурден. Подставьте в предыдущее предложение взамен «функционального наполнения» конкретный продукт, и вы уясните о чем мы хотим сказать: «расчет химических реакций», «анализ текста» и т.д.

Помимо того, сама библиотека пользовательского интерфейса в C++ Builder довольно оригинальна. Это VCL (Visual Component Library), всецело позаимствованная из Delphi, т.е. написанная на Паскале. По Паскалевским исходникам автоматически создаются заголовочные файлы, которые в дальнейшем включаются в файлы, написанные на C++. Необходимо сказать, что классы, которые представляют из себя VCL-компоненты это не обычные C++ классы; для совместимости с Delphi их пришлось отчасти изменить (скажем, VCL-классы не могут участвовать во множественном наследовании); т.е. в C++ Builder есть два вида классов: обычные C++ классы и VCL-классы.

Помимо всего прочего, C++ Builder еще и вреден. Вследствие того что очень много начинающих программистов используют его, расхваливают за то, что при его помощи все так просто делается и не подозревают о том, что это, на самом деле, не правильно. Ведь область применения C++ Builder, в общем-то, достаточно хорошо определена — это клиентские части для каких-либо БД. В нем все есть для этого: быстрое создание интерфейса, генераторы отчетов, средства сопряжения с таблицами. Но все, что выходит за границы данной области, извините, надо писать «как обычно».

Связано это с тем, что, создание программ, которые в принципе не переносимы — это просто издевательство над идеями C++. Ясно, что написать программу, которая компилируется несколькими компиляторами это в принципе сложно, но сделать так, чтобы это было ко всему прочему и невозможно, до чрезвычайности неприлично. Всякая программа уже должна изначально (и это даже не вопрос для обсуждения) иметь очень отчетливую грань между своим «содержанием» и «пользовательским интерфейсом», между которыми должна быть некоторая прослойка (программный интерфейс) при помощи которой «пользовательский интерфейс» общается с «содержанием». В подобном виде можно сделать хоть десяток пользовательских интерфейсов на различных платформах, очень просто «прикрутить» COM или CORBA, написать соответствующий этой же программе CGI скрипт и т.д. В общем, немало достоинств по сравнению с жестким внедрением библиотеки пользовательского интерфейса внутрь программы против одного преимущества обратного подхода: отсутствие необходимости думать перед тем, как начать программировать.

Необходимо сказать, что C++ Builder или Delphi такой популярности как у нас, за границей не имеют. Там эту же нишу прочно занял Visual Basic, что достаточно точно говорит об области применения RAD-средств.

C++ Builder буквально навязывает программисту свой собственный стиль программирования, при котором, даже при особом желании, перейти с C++ Builder на что-то другое уже не предоставляется возможным. Помимо того, быстрое создание интерфейса это еще не панацея от всех бед, а, скорее, еще одна новая беда, в частности из-за того, что программисту приходится выполнять не свойственную ему задачу построения пользовательского интерфейса.

Глава 5.
Применение «умных» указателей

Принципы использования «умных» указателей знакомы любому программисту на C++. Идея предельно проста: взамен того, чтобы пользоваться объектами некоторого класса, указателями на эти объекты или ссылками, определяется новый тип для которого переопределен селектор **->**, что позволяет использовать объекты такого типа в качестве ссылок на реальные объекты. На всякий случай, приведем следующий пример:

```
class A {
public:
 void method();
};

class APtr {
protected:
 A* a;
public:
 APtr();
 ~APtr();
 A* operator->();
};

inline APtr::APtr() : a(new A)
{ }

inline APtr::~APtr()
{
 delete a;
}

inline A* APtr::operator->()
{
 return a;
}
```

Теперь для объекта, определенного как

```
APtr: aptr;
```

можно использовать следующую форму доступа к члену **a**:

```
aptr->method();
```


Тонкости того, почему **operator->()** возвращает именно указатель **A*** (у которого есть свой селектор), а не, скажем, ссылку **A&** и все равно все компилируется таким образом, что выполнение доходит до метода **A::method()**, пропустим за ненадобностью — здесь мы не планируем рассказывать о том, как работает данный механизм и какие приемы применяются при его использовании.

Достоинства подобного подхода, в принципе, очевидны: возникает возможность контроля за доступом к объектам; малость тривиальных телодвижений и получается указатель, который сам считает количество используемых ссылок и при обнулении автоматически уничтожает свой объект, что позволяет не заботиться об этом самостоятельно... не важно? Почему же: самые трудно отлавливаемые ошибки — это ошибки в употреблении динамически выделенных объектов. Сплошь и рядом можно встретить попытку использования указателя на удаленный объект, двойное удаление объекта по одному и тому же адресу или неудаление объекта. При этом последняя ошибка, в принципе, самая невинная: программа, в которой не удаляются объекты (значит, теряется память, которая могла бы быть использована повторно) может вполне спокойно работать в течение некоторого периода (причем это время может спокойно колебаться от нескольких часов до нескольких дней), чего вполне хватает для решения некоторых задач. При этом заметить такую ошибку довольно просто: достаточно наблюдать динамику использования памяти программой; кроме того, имеются специальные средства для отслеживания подобных казусов, скажем, BoundsChecker.

Первая ошибка в данном списке тоже, в принципе, довольно элементарная: использование после удаления скорее всего приведет к тому, что операционная система скажет соответственное системное сообщение. Хуже становится тогда, когда подобного сообщения не возникает (т.е., данные достаточно правдоподобны или область памяти уже занята чем-либо другим), тогда программа может повести себя каким угодно образом.

Вторая ошибка может дать самое большое количество неприятностей. Все дело в том, что, хотя на первый взгляд она ничем особенным не отличается от первой, однако на практике вторичное удаление объекта приводит к тому, что менеджер кучи удаляет что-то совсем немыслимое. Вообще, что значит «удаляет»? Это значит, что помечает память как пустую (готовую к использованию). Как правило, менеджер кучи, для того чтобы знать, сколько памяти удалить, в блок выделяемой памяти вставляет его размер. Так вот, если память уже была занята чем-то другим, то по «неверному» указателю находится неправильное значение размера блока, вследствие этого менеджер кучи удалит некоторый случайный размер используемой памяти. Это даст следующее: при следующих выде-

лениях памяти (рано или поздно) менеджер кучи отдаст эту «неиспользуемую» память под другой запрос и... на одном клочке пространства будут ютиться два разных объекта. Крах программы произойдет почти обязательно, это лучшее что может произойти. Значительно хуже, если программа останется работать и будет выдавать правдоподобные результаты. Одна из самых оригинальных ошибок, с которой можно столкнуться и которая, скорее всего, будет вызвана именно повторным удалением одного и того же указателя, то, что программа, работающая несколько часов, рано или поздно «падет» в функции **malloc()**. Причем проработать она должна будет именно несколько часов, иначе эта ситуация не повторится.

Таким образом, автоматическое удаление при гарантированном неиспользовании указателя, это очевидный плюс. В принципе, можно позавидовать программистам на Java, у которых аналогичных проблем не возникает; зато, у них возникают другие проблемы.

Целесообразность использования «умных» указателей хорошо видно в примерах реального использования. Вот, к примеру, объявление «умного» указателя с подсчетом ссылок:

```
template<class T>
class MPtr
{
public:
 MPtr();
 MPtr(const MPtr<T>& p);
 ~MPtr();
 MPtr(T* p);

 T* operator->() const;
 operator T*() const;
 MPtr<T>& operator=(const MPtr<T>& p);
protected:
 struct RealPtr
 {
 T* pointer;
 unsigned int count;

 RealPtr(T* p = 0);
 ~RealPtr();
 };
 RealPtr* pointer;
private:
};
```


Особенно стоит оговорить здесь конструктор **MPtr::MPtr(T* p)**, который несколько выбивается из общей концепции. Все дело в том, что гарантировать отсутствие указателей на реальный объект может лишь создание такого объекта где-то внутри, это сделано в **MPtr::MPtr()**, где вызов **new** происходит самостоятельно. В итоге некоторая уверенность в том, что значение указателя никто нигде не сохранил без использования умного указателя, все-таки есть. Однако, очень нередко встречается такое, что у типа **T** может и не быть конструктора по умолчанию и объекту такого класса непременно при создании требуются какие-то аргументы для правильной инициализации. Совершенно правильным будет для подобного случая породить из **MPtr** новый класс, у которого будут такие же конструкторы, как и у требуемого класса. Оттого что подобный конструктор **MPtr::MPtr(T* p)** будет использоваться только лишь как **MPtr<T> ptr(new T(a,b,c))** и никак иначе, этот конструктор введен в шаблон.

Еще один спорный момент: присутствие оператора преобразования к **T***. Его наличие дает потенциальную возможность где-нибудь сохранить значение реального указателя.

Помимо **MPtr** можно использовать еще одну разновидность «умных» указателей, которая закономерно вытекает из описанной выше и отличается только лишь одной тонкостью:

```
template<class T>
class MCPtr
{
public:
 MCPtr(const MPtr<T>& p);
 MCPtr(const MCPtr<T>& p);
 ~MCPtr();

 const T* operator->() const;
 operator const T*() const;
 MCPtr<T>& operator=(const MPtr<T>& p)
 MCPtr<T>& operator=(const MCPtr<T>& p);
protected:
 MPtr<T> ptr;
private:
 MCPtr();
};
```

Во-первых, это надстройка (адаптер) над обычным указателем. А во-вторых, его главное отличие, это то, что **operator->** возвращает константный указатель, а не обычный. Это очень просто и, на самом деле, очень полезно: все дело в том, что это дает использовать объект в двух контекстах — там, где его можно изменять (скажем, внутри другого объ-

екта, где он был создан) и там, где можно пользоваться лишь константным интерфейсом (т.е., где изменять нельзя; к примеру, снаружи объекта-фабрики). Это разумно вытекает из простых константных указателей. Для того, чтобы пользоваться **MCPtr** требуется единственное (хотя и достаточно строгое) условие: во всех классах должна быть корректно расставлена константность методов. Вообще, это — признак профессионального программиста: использование модификатора **const** при описании методов.

Как правило используют «умные» указатели в том, что называется фабриками объектов (или, в частности, производящими функциями): т.е., для того, чтобы вернуть объект, удовлетворяющий какому-то интерфейсу. При использовании подобного рода указателей клиентской части становится очень удобно — опускаются все проблемы, связанные с тем, когда можно удалить объект, а когда нельзя (скажем, при совместном использовании одного и того же объекта разными клиентами — клиенты не обязаны знать о существовании друг друга).

Помимо всего прочего, переопределение селектора позволяет простым образом вставить синхронизацию при создании многопоточных приложений. Вообще, подобные «обертки» чрезвычайно полезны, им можно найти массу применений.

Несомненно, использовать «умные» указатели необходимо с осторожностью. Все дело в том, что, как у всякой простой идеи, у нее есть один очень большой недостаток: несложно придумать пример, в котором два объекта ссылаются друг на друга через «умные» указатели и... никогда не будут удалены. Почему? Потому что счетчики ссылок у них всегда будут как минимум 1, при том, что снаружи на них никто не ссылается. Есть рекомендации по поводу того, как распознавать такие ситуации во время выполнения программы, но они очень громоздки и, поэтому не годятся к применению. Ведь что прельщает в «умных» указателях? Простота. Фактически, ничего лишнего, а сколько можно при желании извлечь пользы из их применения.

Посему надо тщательно следить еще на стадии проектирования за тем, чтобы подобных цепочек не могло бы возникнуть в принципе. Потому как если такая возможность будет, то в конце концов она проявит себя.

«Умные» указатели активно используются в отображении COM-объектов и CORBA-объектов на C++: они позволяют прозрачно для программиста организовать работу с объектами, которые реально написаны на другом языке программирования и выполняются на другой стороне земного шара.

Техника подсчета ссылок в явном виде (через вызов методов интерфейса AddRef() и Release()) используется в технологии COM.

Еще стоит сказать про эффективность использования «умных» указателей. Возможно, это кого-то удивит, но затраты на их использование при выполнении программы минимальны. Почему? Потому что используются шаблоны, а все методы-члены классов (и, в особенности селектор) конечно же объявлены как **inline**. Подсчет ссылок не сказывается на обращении к объекту, только на копировании указателей, а это не такая частая операция. Ясно, что использование шаблонов усложняет работу компилятора, но это не так важно.

Глава 6. Рассуждения на тему «умных» указателей

При изучении C++, не раз можно встретиться с «умными» указателями. Они встречаются везде и все время в разных вариантах. Без стандартизации.

Вообще, мысль об упрощении себе жизни вертится в головах программистов всегда: «Лень — двигатель прогресса». Поэтому и были придуманы не просто указатели, а такие из них, которые брали часть умственного напряга на себя, тем самым, делая вид, что они нужны.

Итак, что такое **SmartPointer**-ы? По сути это такие классы, которые умеют чуть больше... — а в общем, смотрим пример:

```
class A
{
 private:
 int count;
 public:
 A(){count = 0;}
 void addref(){count++;}
 void release(){if(--count == 0) delete this;}
 protected:
 ~A();
 public:
 void do_something(){cout << "Hello";}
};
```

Сначала придумали внутри объекта считать ссылки на него из других объектов при помощи «механизма подсчета ссылок». Суть здесь в том, что когда вы сохраняете ссылку на объект, то должны вызвать для него **addref**, а когда избавляетесь от объекта, то вызвать **release**. Сложно?

Совсем нет — это дело привычки. Таким образом, объект умеет сам себя удалять. Здорово? Так оно и есть.

Кстати такой объект может существовать только в куче, поскольку деструктор в «защищенной» зоне и по той же причине нельзя самому сделать «**delete a**» обойдя **release**.

Теперь переходим к собственно самим «умным» указателям и опять пример:

```
class PA
{
 private:
 A* pa;
 public:
 PA(A* p){pa = p; p->addref();}
 ~PA(){pa->release();}
 A* operator ->(){return pa;}
};
```

Что мы видим? Есть класс **PA**, который умеет принимать нормальный указатель на объект класса **A** и выдавать его же по обращению к селектору членов класса. «Ну и что? — скажете вы, — Как это может помочь?». За помощью обратимся к двум примерам, которые иллюстрирует эффективность использования класса **PA**:

```
...
{
 A* pa = new A();
 pa->addref();
 pa->do_something();
 pa->release();
}


...
{
 PA pa = new A();
 pa->do_something();
}
```

Посмотрим внимательнее на эти два отрывка... Что видим? Видим экономию двух строчек кода. Здесь вы наверное скажете: «И что, ради этого мы столько старались?». Но это не так, потому что с введением класса **PA** мы переложили на него все неприятности со своевременными вызовами **addref** и **release** для класса **A**. Вот это уже что-то стоит!

Дальше больше, можно добавить всякие нужные штучки, типа оператора присваивания, еще одного селектора (или как его некоторые

называют «разименователь» указателя) и так далее. Таким образом получится удобная вещь (конечно, если все это дело завернуть в шаблон.

Теперь немного сменим направление рассуждений. Оказывается существуют такие вещи, как «Мудрые указатели», «Ведущие указатели», «Гениальные указатели», «Грани», «Кристаллы» — в общем, хватает всякого добра. Правда, в практичности этих поделок можно усомниться, несмотря на их «изящество». То есть, конечно, они полезны, но не являются, своего рода, панацеей (кроме «ведущих» указателей).

В общем особую роль играют только «ведущие» и «умные» указатели.

Начнем с такого класса как **Countable**, который будет отвечать за подсчет чего либо.

Итак он выглядит примерно так (в дальнейшем будем опускать реализации многих функций из-за их тривиальности, оставляя, тем самым, только их объявления:

```
class Countable
{
 private:
 int count;
 public:
 int increment ();
 int decrement ();
};
```

Здесь особо нечего говорить, кроме того, что, как всегда, этот класс можно сделать более «удобным», добавив такие вещи, как поддержку режима многопоточности и т.д.

Следующий простой класс прямо вытекает из многопоточности и осуществляет поддержку этого режима для своих детей:

```
class Lockable
{
 public:
 void lock();
 void unlock();
 bool islocked();
};
```

Этот класс не вносит никакой новизны, но стандартизует поддержку многопоточности при использовании, например, различных платформ.

Теперь займемся собственно указателями:

```
class AutoDestroyable : public Countable
{
 public:
 virtual int addref ();
 virtual int release ();
 protected:
 virtual ~AutoDestroyable();
 ...
};
```

Из кода видно, что этот класс занимается подсчетом ссылок и «убивает» себя если «пришло время».

А сейчас процитируем код «умного» указателя, для того чтобы синхронизовать наши с вами понимание этого чуда.

```
template <class T>
class SP
{
 private:
 T* m_pObject;
 public:
 SP ( T* pObject){ m_pObject = pObject; }
 ~SP () { if( m_pObject ) m_pObject->release (); }
 T* operator -> ();
 T& operator * ();
 operator T* ();
 ...
};
```

Это уже шаблон, он зависит от объекта класса, к которому применяется. Задача «умного» указателя была рассмотрена выше и итог при сравнении с ситуацией без его использования положителен только тем, что для объекта, создаваемого в куче, не надо вызывать оператор **delete** — он сам вызовется, когда это понадобится.

Теперь остановимся на минутку и подумаем, когда мы можем использовать этот тип указателей, а когда нет.

Главное требование со стороны **SP**, это умение основного объекта вести подсчет ссылок на себя и уничтожать себя в тот момент, когда он не становится нужен. Это серьезное ограничение, поскольку не во все используемые классы вы сможете добавить эти возможности. Вот несколько причин, по которым вы не хотели бы этого (или не можете):

- Вы используете закрытую библиотеку (уже скомпилированную) и физически не можете добавить кусок кода в нее.
- Вы используете открытую библиотеку (с исходными кодами), но не хотите изменять ее как-либо, потому что все время меняете ее на более свежую (кто-то ее делает и продвигает за вас).
- И, наконец, вы используете написанный вами класс, но не хотите по каким-либо причинам вставлять поддержку механизма подсчета ссылок.

Итак, причин много или по крайней мере достаточно для того, чтобы задуматься над более универсальным исполнением **SP**. Посмотрим на схематичный код «ведущих» указателей и «дескрипторов»:

```
template <class T>
clacc MP : public AutoDestroyable
{
 private:
 T* m_pObj;
 public:
 MP(T* p);
 T* operator ->();
 protected:
 operator T*();
};

template <class T>
class H
{
 private:
 MP<T>* m_pMP;
 public:
 H(T*);
 MP& operator T->();
 bool operator ==(H<T>&);
 H operator =(H<T>&);
};
```

Что мы видим на этот раз? А вот что. **MP** — это «ведущий» указатель, т.е. такой класс, который держит в себе объект основного класса и не дает его наружу. Появляется только вместе с ним и умирает аналогично. Его главная цель, это реализовывать механизм подсчета ссылок.

В свою очередь **H** — это класс очень похожий на **SP**, но общается не с объектом основного класса, а с соответствующим ему **MP**.

Результат этого шаманства очевиден — мы можем использовать механизм «умных» указателей для классов не добавляя лишнего кода в их реализацию.

И это действительно так, ведь широта использования такого подхода резко увеличивается и уже попахивает панацеей.

Что же можно сказать о многопоточности и множественном наследовании: при использовании классов **MP** и **H** — нет поддержки этих двух вещей. И если с первой все понятно (нужно наследовать MP от Lockable), то со вторым сложнее.

Итак, посвятим немного времени описанию еще одного типа указателей, которые призваны «решить» проблему множественного наследования (а точнее полиморфизма).

Рассмотрим классы **PP** и **TP**:

```
class PP : public AutoDestroyable
{};

template<class T>
class TP
{
 protected:
 T* m_pObject;
 PP* m_pCounter;

 public:
 TP ();
 TP ( T* pObject );
 TP<T>& operator = ( TP<T>& tp );
 T* operator -> ();
 T& operator * ();
 operator T* ();
 bool operator == ( TP<T>& tp );
};
```

Класс **PP** является «фиктивным ребенком» **AutoDestroyable** и вы не забивайте себе этим голову. А вот класс **TP** можно посмотреть и попристальнее.

Схема связей в этом случае выглядит уже не **H->MP->Obj**, а **PP<-TP->Obj**, т.е. Счетчик ссылок (а в данном случае, это **PP**) никак не связан

с основным объектом или каким-либо другим и занимается только своим делом — ссылками. Таким образом, на класс **TP** ложится двойная обязанность: выглядеть как обычный указатель и отслеживать вспомогательные моменты, которые связаны со ссылками на объект.

Как же нам теперь использовать полиморфизм? Ведь мы хотели сделать что-то вроде:

```
class A : public B
...
TP<A> a;
TP<B> b;
a = new B;
b = (B*)a;
...
```

Для этого реализуем следующую функцию (и внесем небольшие изменения в класс **TP** для ее поддержки):

```
template <class T, class TT>
TP<T> smart_cast ( TP<TT>& tpin );
```

Итак, теперь можно написать что-то вроде (и даже будет работать):

```
class A : public B
...
TP<A> a;
TP<B> b;
a = new B;
b = smart_cast<B, A>(a);

// или если вы используете Visual C++, то даже
b = smart_cast<B>(a);
...
```

Вам ничего не напоминает? Ага, схема та же, что и при использовании **static_cast** и **dynamic_cast**. Так как схожесть очень убедительна, можно заключить, что такое решение проблемы более чем изящно.

Глава 7. Виртуальные деструкторы

В практически любой мало-мальски толковой книге по C++ рассказывается, зачем нужны виртуальные деструкторы и почему их надо

использовать. При всем при том, как показывает практика, ошибка, связанная с отсутствием виртуальных деструкторов, повсеместно распространена.

Итак, рассмотрим небольшой пример:

```
class A
{
public:
 virtual void f() = 0;
 ~A();
};
class B : public A
{
public:
 virtual void f();
 ~B();
};
```

Вызов компилятора **gcc** строкой:

```
g++ -c -Wall test.cpp
```

даст следующий результат:

```
test.cpp:6: warning: `class A' has virtual functions but non-virtual destructor
test.cpp:13: warning: `class B' has virtual functions but non-virtual destructor
```

Это всего лишь предупреждения, компиляция прошла вполне успешно. Однако, почему же **gcc** выдает подобные предупреждения?

Все дело в том, что виртуальные функции используются в C++ для обеспечения полиморфизма — т.е., клиентская функция вида:

```
void call_f(A* a)
{
 a->f();
}
```

никогда не «знает» о том, что конкретно сделает вызов метода **f()** — это зависит от того, какой в действительности объект представлен указателем **a**. Точно так же сохраняются указатели на объекты:

```
std::vector<A*> a_collection;
a_collection.push_back(new B());
```


В результате такого кода теряется информация о том, чем конкретно является каждый из элементов **a_collection** (имеется в виду, без использования RTTI). В данном случае это грозит тем, что при удалении объектов:

```
for(std::vector<A*>::iterator i = ... )
 delete *i;
```

все объекты, содержащиеся в **a_collection**, будут удалены так, как будто это — объекты класса **A**.

В этом можно убедиться, если соответствующим образом определить деструкторы классов **A** и **B**:

```
inline A::~A()
{
 puts("A::~A()");
}

inline B::~B()
{
 puts("B::~B()");
}
```

Тогда выполнение следующего кода:

```
A* ptr = new B();
delete ptr;
```

приведет к следующему результату:

```
A::~A()
```

Если же в определении класса **A** деструктор был бы сделан виртуальным (**virtual ~A();**), то результат был бы другим:

```
B::~B()
A::~A()
```

В принципе, все сказано. Но, несмотря на это, очень многие программисты все равно не создают виртуальных деструкторов. Одно из распространенных заблуждений — виртуальный деструктор необходим только в том случае, когда на деструктор порожденных классов возлагаются какие-то нестандартные функции; если же функционально деструктор порожденного класса ничем не отличается от деструктора предка, то делать его виртуальным совершенно необязательно. Это неправда, потому что даже если деструктор никаких специальных действий не выполняет, он все равно должен быть виртуальным, иначе не будут вызваны деструкторы для объектов-членов класса, которые появились по отношению к предку. То есть:

```
#include <stdio.h>
```

```
class A
{
public:
 A(const char* n);
 ~A();
protected:
 const char* name;
};

inline A::A(const char* n) : name(n)
{ }
inline A::~A()
{
 printf("A::~A() for %s.\n", name);
}

class B
{
public:
 virtual void f();
 B();
 ~B();
protected:
 A a1;
};

inline B::~B()
{ }

inline B::B() : a1("a1")
{ }
void B::f() { }
class C : public B
{
public:
 C();
protected:
 A a2;
};

inline C::C() : a2("a2")
{ }
```



```
int main()
{
 B* ptr = new C();
 delete ptr;
 return 0;
}
```

Компиляция данного примера проходит без ошибок (но с предупреждениями), вывод программы следующий:

```
A::~A() for a1
```

Немного не то, что ожидалось? Тогда поставим перед названием деструктора класса **B** слово **virtual**. Результат изменится:

```
A::~A() for a2
A::~A() for a1
```

Сейчас вывод программы несколько более соответствует действительности.

Глава 8. Запись структур данных в двоичные файлы

Чтение и запись данных, вообще говоря, одна из самых часто встречающихся операций. Сложно себе представить программу, которая бы абсолютно не нуждалась бы в том, чтобы отобразить где-нибудь информацию, сохранить промежуточные данные или, наоборот, восстановить состояние прошлой сессии работы с программой.

Собственно, все эти операции достаточно просто выполняются — в стандартной библиотеке любого языка программирования обязательно найдутся средства для обеспечения ввода и вывода, работы с внешними файлами. Но и тут находятся некоторые сложности, о которых, обычно, не задумываются.

Итак, как все это выглядит обычно? Имеется некоторая структура данных:

```
struct data_item
{
 type_1 field_1;
 type_2 field_2;
 // ...
 type_n field_n;
};
data_item i1;
```

Каким образом, например, сохранить информацию из **i1** так, чтобы программа во время своего повторного запуска, смогла восстановить ее? Наиболее частое решение следующее:

```
FILE* f = fopen("file", "wb");
fwrite((char*)&i1, sizeof(i1), 1, f);
fclose(f);
```

assert расставляется по вкусу, проверка инвариантов в данном примере не является сутью. Тем не менее, несмотря на частоту использования, этот вариант решения проблемы не верен.

Нет, он будет компилироваться и, даже будет работать. Мало того, будет работать и соответствующий код для чтения структуры:

```
FILE* f = fopen("file", "rb");
fread((char*)&i1, sizeof(i1), 1, f);
fclose(f);
```

Что же тут неправильного? Ну что же, для этого придется немного пофилософствовать. Как бы много не говорили о том, что C — это почти то же самое, что и ассемблер, не надо забывать, что он является все-таки языком высокого уровня. Следовательно, в принципе, программа написанная на C (или C++) может (теоретически) компилироваться на разных компиляторах и разных платформах. К чему это? К тому, что данные, которые сохранены подобным образом, в принципе не переносимы.

Стоит вспомнить о том, что для структур неизвестно их физическое представление. То есть, для конкретного компилятора оно, быть может, и известно (для этого достаточно посмотреть работу программы «вооруженным взглядом», т.е. отладчиком), но о том, как будут расположены в памяти поля структуры на какой-нибудь оригинальной машине, неизвестно. Компилятор со спокойной душой может перетасовать поля (это, в принципе, возможно) или выровнять положение полей по размеру машинного слова (встречается сплошь и рядом). Для чего?

Для увеличения скорости доступа к полям. Понятно, что если поле начинается с адреса, не кратного машинному слову, то прочитать его содержимое не так быстро, как в ином случае. Таким образом, сохранив данные из памяти в бинарный файл напрямую мы получаем дамп памяти конкретной архитектуры (не говоря о том, что **sizeof** совершенно не обязан возвращать количество байт).

Плохо это тем, что при переносе данных на другую машину при попытке прочитать их той же программой (или программой, использующую те же структуры) вполне можно ожидать несколько некорректных результатов. Это связано с тем, что структуры могут быть представлены

по другому в памяти (другое выравнивание), различается порядок следования байтов в слове и т.п. Как этого избежать?

Обычный «костыль», который применяется, например, при проблемах с выравниванием, заключается в том, что компилятору явно указывается как надо расставлять поля в структурах. В принципе, любой компилятор дает возможность управлять выравниванием. Но выставить одно значение для всего проекта при помощи ключей компилятора (обычно это значение равно 1, потому что при этом в сохраненном файле не будет пустых мест) нехорошо, потому что это может снизить скорость выполнения программы. Есть еще один способ указания компилятору размера выравнивания, он заключается в использовании директивы препроцессора **#pragma**. Это не оговорено стандартом, но обычно есть директива **#pragma pack**, позволяющая сменить выравнивание для определенного отрезка исходного текста. Выглядит это обычно примерно так:

```
#pragma pack(1)
struct { /* ... */ };
#pragma pack(4)
```

Последняя директива **#pragma pack(4)** служит для того, чтобы вернуться к более раннему значению выравнивания. В принципе, конечно же при написании исходного текста никогда доподлинно заранее неизвестно, какое же было значение выравнивания до его смены, поэтому в некоторых компиляторах под Win32 есть возможность использования стека значений (пошло это из MS Visual C++):

```
#pragma pack(push, 1)
struct { /* ... */ };
#pragma pack(pop)
```

В примере выше сначала сохраняется текущее значение выравнивания, затем оно заменяется 1, затем восстанавливается ранее сохраненное значение. При этом, подобный синтаксис поддерживает даже **gcc** для win32 (еще стоит заметить, что, вроде, он же под Unix использовать такую запись **#pragma pack** не дает). Есть альтернативная форма **#pragma pack()**, поддерживаемая многими компиляторами (включая **msvc** и **gcc**), которая устанавливает значение выравнивания по умолчанию.

И, тем не менее, это не хорошо. Опять же, это дает очень интересные ошибки. Представим себе следующую организацию исходного текста. Сначала заголовочный файл **inc.h**:

```
#ifndef __inc_h__
#define __inc_h__

class Object
{
```

```
 // ...
};

#endif // __inc_h__
```

Представьте себе, что существуют три файла **file1.cpp**, **file2.cpp** и **file2.h**, которые этот хидер используют. Допустим, что в **file2.h** находится функция **foo**, которая (например) записывает **Object** в файл:

```
// file1.cpp
#include "inc.h"
#include "file2.h"

int main()
{
 Object* obj = new Object();
 foo(obj, "file");

 delete obj;

 return 0;
}

// file2.h
#ifndef __file2_h__
#define __file2_h__

#pragma pack(1)

#include "inc.h"

void foo(const Object* obj, const char* fname);

#pragma pack(4)

#endif // __file2_h__

// file2.cpp
#include "file2.h"

void foo(const Object* obj, const char* fname)
{
 // ...
}
```


Это все скомпилируется, но работать не будет. Почему? Потому что в двух разных единицах компиляции (**file1.cpp** и **file2.cpp**) используется разное выравнивание для одних и тех же структур данных (в данном случае, для объектов класса **Object**). Это даст то, что объект переданный по указателю в функцию **foo()** из функции **main()** будет разным (и, конечно же, совсем неправдоподобным). Понятно, что это явный пример «плохой» организации исходных текстов — использование директив компилятора при включении заголовочных файлов, но, поверьте, он существует.

Отладка программы, содержащую подобную ошибку, оказывается проверкой на устойчивость психики. Потому что выглядит это примерно так: следим за объектом, за его полями, все выглядит просто замечательно и вдруг, после того как управление передается какой-то функции, все, что содержится в объекте, принимает «бредовые» формы, какие-то неизвестно откуда взявшиеся цифры...

На самом деле **#pragma pack** не является панацеей. Мало того, использование этой директивы практически всегда неправомерно. Можно даже сказать, что эта директива в принципе редко когда нужна (во всяком случае, при прикладном программировании).

Правильным же подходом является сначала записать все поля структуры в нужном порядке в некоторый буфер и скидывать в файл уже содержимое буфера. Это очень просто и очень эффективно, потому что все операции чтения/записи можно собрать в подпрограммы и менять их при необходимости таким образом, чтобы обеспечить нормальную работу с внешними файлами. Проиллюстрируем этот подход:

```
template<class T>
inline size_t get_size(const T& obj)
{
 return sizeof(obj);
}
```

Эта функция возвращает размер, необходимый для записи объекта. Зачем она понадобилась? Во-первых, возможен вариант, что **sizeof** возвращает размер не в байтах, а в каких-то собственных единицах. Во-вторых, и это значительно более необходимо, объекты, для которых вычисляется размер, могут быть не настолько простыми, как **int**. Например:

```
template<>
inline size_t get_size<std::string>(const std::string& s)
{
 return s.length() + 1;
}
```

Надеемся, понятно, почему выше нельзя было использовать **sizeof**.

Аналогичным образом определяются функции, сохраняющие в буфер данные и извлекающие из буфера информацию:

```
typedef unsigned char byte_t;
template<class T>
inline size_t save(const T& i, byte_t* buf)
{
 *((T*)buf) = i;
 return get_size(i);
}

template<class T>
inline size_t restore(T& i, const byte_t* buf)
{
 i = *((T*)buf);
 return get_size(i);
}
```

Понятно, что это работает только для простых типов (**int** или **float**), уж очень много чего наворочено: явное приведение указателя к другому типу, оператор присваивания... конечно же, очень нехорошо, что такой **save()** доступен для всех объектов. Понятно, что очень просто от него избавиться, убрав шаблонность функции и реализовав аналогичный **save()** для каждого из простых типов данных. Тем не менее, это всего-лишь примеры использования:

```
template<>
inline size_t save<MyObject>(const MyObject& s, byte_t* buf)
{
 // ...
}
```

Можно сделать и по другому. Например, ввести методы **save()** и **restore()** в каждый из сохраняемых классов, но это не столь важно для принципа этой схемы. Поверьте, это достаточно просто использовать, надо только попробовать. Мало того, здесь можно вставить в **save<long>()** вызов **htonl()** и в **restore<long>()** вызов **ntohl()**, после чего сразу же упрощается перенос двоичных файлов на платформы с другим порядком байтов в слове... в общем, преимуществ — море. Перечислять все из них не стоит, но как после этого лучше выглядит исходный текст, а как приятно вносить изменения...

Глава 9.
Оператор безусловного перехода goto

Так уж сложилось, что именно присутствие или отсутствие этого оператора в языке программирования всегда вызывает жаркие дебаты среди сторонников «хорошего стиля» программирования. При этом, и те, кто «за», и те, кто «против» всегда считают признаком «хорошего тона» именно использование **goto** или, наоборот, его неиспользование. Не вставая на сторону ни одной из этих «школ», просто покажем, что действительно есть места, где использование **goto** выглядит вполне логично.

Но сначала о грустном. Обычно в вину **goto** ставится то, что его присутствие в языке программирования позволяет делать примерно такие вещи:

```
int i, j;
for(i = 0; i < 10; i++)
{
 // ...

 if(condition1)
 {
  j = 4;
  goto label1;
 }

 // ...

 for(j = 0; j < 10; j++)
 {
 // ...
 label1:
 // ...
 if(condition2)
  {
  i = 6;
  goto label2;
  }
 }

 // ...
label2:
 // ...
}
```

Прямо скажем, что такое использование **goto** несколько раздражает, потому что понять при этом, как работает программа при ее чтении будет очень сложно. А для человека, который не является ее автором, так и вообще невозможно. Понятно, что вполне вероятны случаи, когда такого подхода требует какая-нибудь очень серьезная оптимизация работы программы, но делать что-то подобное программист в здравом уме не должен. На самом деле, раз уж мы привели подобный пример, в нем есть еще один замечательный нюанс — изменение значения переменной цикла внутри цикла. Смеем вас заверить, что такое поведение вполне допустимо внутри **do** или **while**; но когда используется **for** — такого надо избегать, потому что отличительная черта **for** как раз и есть жестко определенное местоположение инициализации, проверки условия и инкремента (т.е., изменения переменной цикла). Поэтому читатель исходного текста, увидев «полный» **for** (т.е. такой, в котором заполнены все эти три места) может и не заметить изменения переменной где-то внутри цикла. Хотя для циклов с небольшим телом это, наверное, все-таки допустимо — такая практика обычно применяется при обработке строк (когда надо, например, считать какой-то символ, который идет за «спецсимволом», как «\\» в строках на Си; вместо того, чтобы вводить дополнительный флаг, значительно проще, увидев «\», сразу же сдвинуться на одну позицию и посмотреть, что находится там). В общем, всегда надо руководствоваться здравым смыслом и читабельностью программы.

Если здравый смысл по каким-то причинам становится в противовес читабельности программы, то это место надо обнести красными флагами, чтобы читатель сразу видел подстерегающие его опасности.

Тем не менее, вернемся к **goto**. Несмотря на то, что такое расположение операторов безусловного перехода несколько нелогично (все-таки, вход внутрь тела цикла это, конечно же, неправильно) — это встречается.

Итак, противники использования **goto** в конечном итоге приходят к подобным примерам и говорят о том, что раз такое его использование возможно, то лучше чтобы его совсем не было. При этом, конечно же, никто обычно не спорит против применения, например, **break**, потому что его действие жестко ограничено. Хочется сказать, что подобную ситуацию тоже можно довести до абсурда, потому что имеются программы, в которых введен цикл только для того, чтобы внутри его тела использовать **break** для выхода из него (т.е., цикл делал только одну итерацию, просто в зависимости от исходного состояния заканчивался в разных местах). И что помешало автору использовать **goto** (раз уж хотелось), кроме догматических соображений, не понятно.

Собственно, мы как раз подошли к тому, что обычно называется «разумным» применением этого оператора. Вот пример:

```
switch(key1)
{
case q1 :
 switch(key2)
 {
  case q2 : break;
 }
 break;
}
```

Все упрощено до предела, но, в принципе, намек понятен. Есть ситуации, когда нужно что-то в духе **break**, но на несколько окружающих циклов или операторов **switch**, а **break** завершает только один. Понятно, что в этом примере читабельность, наверное, не нарушена (в смысле, использовался бы вместо внутреннего **break goto** или нет), единственное, что в таком случае будет выполнено два оператора перехода вместо одного (**break** это, все-таки, разновидность **goto**).

Значительно более показателен другой пример:

```
bool end_needed = false;
for( ... )
{
 for( ... )
 {
  if(cond1) { end_needed = true; break; }
 }
 if(end_needed) break;
}
```

Т.е., вместо того, чтобы использовать **goto** и выйти из обоих циклов сразу, пришлось завести еще одну переменную и еще одну проверку условия. Тут хочется сказать, что **goto** в такой ситуации выглядит много лучше — сразу видно, что происходит; а то в этом случае придется пройти по всем условиями и посмотреть, куда они выведут. Надо сказать (раз уж мы начали приводить примеры из жизни), что не раз можно видеть эту ситуацию, доведенную до крайности — четыре вложенных цикла (ну что поделать) и позарез надо инициировать выход из самого внутреннего. И что? Три лишних проверки... Кроме того, введение еще одной переменной, конечно же, дает возможность еще раз где-нибудь допустить ошибку, например, в ее инициализации. Опять же, читателю исходного текста придется постоянно лазить по тексту и смотреть, зачем была нужна эта переменная... в общем: не плодите сущностей без надобности. Это только запутает.

Другой пример разумного использования **goto** следующий:

```
int foo()
{
 int res;

 // ...
 if(...)
 {
  res = 10;
  goto finish;
 }
 // ...

finish:
 return res;
}
```

Понятно, что без **goto** это выглядело бы как **return 10** внутри **if**. Итак, в чем преимущества такого подхода. Ну, сразу же надо вспомнить про концептуальность — у функции становится только один «выход», вместо нескольких (быстро вспоминаем про IDEF). Правда, концептуальность — это вещь такая... Неиспользование **goto** тоже в своем роде концептуальность, так что это не показатель (нельзя противопоставлять догму догме, это просто глупо). Тем не менее, выгоды у такого подхода есть. Во-первых, вполне вероятно, что перед возвратом из функции придется сделать какие-то телодвижения (закрыть открытый файл, например). При этом, вполне вероятно, что когда эта функция писалась, этого и не требовалось — просто потом пришлось дополнить. И что? Если операторов **return** много, то перед каждым из них появится одинаковый кусочек кода. Как это делается? Правильно, методом «cut&paste». А если потом придется поменять? Тоже верно, «search&replace». Объяснять, почему это неудобно не будем — это надо принять как данность.

Во-вторых, обработка ошибок, которая также требует немедленного выхода с возвратом используемых ресурсов. В принципе, в C++ для этого есть механизм исключительных ситуаций, но когда он отсутствует (просто выключен для повышения производительности), это будет работать не хуже. А может и лучше по причине более высокой скорости.

В-третьих, упрощается процесс отладки. Всегда можно проверить что возвращает функция, поставить точку останова (хотя сейчас некоторые отладчики дают возможность найти все **return** и поставить на них точки останова), выставить дополнительный **assert** или еще что-нибудь в этом духе. В общем, удобно.

Еще **goto** очень успешно применятся при автоматическом создании кода — читателя исходного текста там не будет, он будет изучать то, по чему исходный текст был создан, поэтому можно (и нужно) допускать различные вольности.

В заключение скажем, что при правильном использовании оператор **goto** очень полезен. Надо только соблюдать здравый смысл, но это общая рекомендация к программированию на C/C++ (да и вообще, на любом языке программирования), поэтому непонятно почему **goto** надо исключать.

Глава 10. Виртуальный конструктор

Предупреждение: то, что описано — не совсем уж обычные объекты. Возможно только динамическое их создание и только в отведённой уже памяти. Ни о каком статическом или автоматическом их создании не может быть и речи. Это не цель и не побочный эффект, это расплата за иные удобства.

Краткое описание конкретной ситуации, где всё это и происходило. В некотором исследовательском центре есть биохимическая лаборатория. Есть в ней куча соответствующих анализаторов. Железки они умные, работают самостоятельно, лишь бы сунули кассету с кучей материалов и заданиями. Всякий анализатор обрабатывает материалы только определённой группы. Со всех них результаты текут по одному шлангу в центр всяческих обработок и складирования. Масса частностей, но нам они неинтересны. Суть — всякий результат есть результат биохимического анализа. Текущий потоком байт с соответствующими заголовками и всякими телами. Конкретный тип реально выяснить из первых шестнадцати байт. Максимальный размер — есть. Но он лишь максимальный, а не единственно возможный.

Не вдаваясь в подробности принятого решения (это вынудит вдаваться в подробности задания), возникла конкретная задача при реализации — уже во время исполнения конструктора объекта конкретный тип результата анализа неизвестен. Неизвестен (соответственно) и его размер. Известен лишь размер пула для хранения некоторого количества этих объектов. Две проблемы — сконструировать объект конкретного типа в конструкторе объекта другого (обобщающего) типа и положить его на это же самое место в памяти. При этом память должно использовать эффективно, всячески минимизируя (главная проблема) фрагментацию пула, ибо предсказать время, в течение которого результат будет оста-

ваться нужным (в ожидании, в частности, своих попутчиков от других анализаторов), невозможно. Это — не очередь (иначе всё было бы значительно проще).

Решение: от классической идиомы envelope/letter (которая сама по себе основа кучи идиом) к «виртуальному» конструктору с особым (либо входящим в состав, либо находящимся в дружеских отношениях) менеджером памяти. Излагается на смеси C++ и недомолвок (некритичных) в виде «. . .»:

```
class BCAR { // Bio-Chemical Analysis Result

friend class BCAR_MemMgr;

protected:
 BCAR() { /* Должно быть пусто!!! */ }
public:
 BCAR( const unsigned char * );
 void *operator new( size_t );
 void operator delete( void * );
 virtual int size() { return 0; }
 . . .

private:
 struct {
  // трали-вали
 } header;
 . . .
};
```

Это был базовый класс для всех прочих конкретных результатов анализов. У него есть свой **new**. Но для реализации идеи используется не дефолтовый **new** из C++ **rtl**, а используется следующее:

```
inline void *operator new( size_t, BCAR *p ) {
 return p;
}
```

Именно за счёт его мы получим **in place** замену объекта одного класса (базового) объектом другого (производного). Раньше было проще — **this** допускал присваивание.

Теперь — менеджер памяти.

```
class BCAR_MemMgr {

friend BCAR;
```



```
public:
 BCAR_MemMgr();
 void alloc( int );
 void free( BCAR *, int );
 BCAR *largest();

private:
 . . .
};
```

Это примерный его вид. Он создаётся в единственном экземпляре:

```
static BCAR_MemMgr MemoryManager;
```

и занимается обслугой пула памяти под все объекты. В открытом интерфейсе у него всего три функции, назначение **alloc/free** любому понятно (хотя **alloc** в действительности ничего не аллоцирует, а делает «обрезание» того, что даёт **largest** и соответствующим образом правит списки менеджера), а **largest** возвращает указатель на самый большой свободный блок. В сущности, она и есть **BCAR::new**, которая выглядит так:

```
void *BCAR::operator new( size_t ) {
 return MemoryManager.largest();
}
```

Зачем самый большой? А затем, что при создании объекта его точный тип ещё неизвестен (ибо создаваться будет через **new BCAR**), поэтому берём по максимуму, а потом **alloc** всё подправит.

Теперь собственно классы для конкретных результатов. Все они выглядят примерно одинаково:

```
class Phlegm: public BCAR {

friend BCAR;

private:
 int size() { retrurn sizeof( Phlegm ); }
 struct PhlegmAnalysisBody {
  // тут всякие его поля
 };
 PhlegmAnalysisBody body;
 Phlegm( const unsigned char *data ): BCAR() {
  MemoryManager.alloc( size() );
  ::memcpy( &body, data + sizeof( header ), sizeof( body ) );
 }
 . . .
};
```

Где тут расположен «виртуальный» конструктор. А вот он:

```
BCAR::BCAR( const unsigned char *dataStream ) {
 ::memcpy( &header, dataStream, sizeof( header ) );
 if( CRC_OK( dataStream ) ) {
  // определяем тип конкретного результата
  // и строим соответствующий объект прямо на месте себя
  switch( AnalysisOf( dataStream ) ) {
case PHLEGM:
 ::new( this ) Phlegm( dataStream );
 break;
case BLOOD:
 ::new( this ) Blood( dataStream );
 break;
case ...:
 . . .
  }
 . . .
}
```

Теперь, чтобы не вынуждать вас носиться по всему материалу в целях построения целостной картины, объясним происходящее по шагам.

Менеджер памяти создан, инициализирован. Пул памяти существует (хотя бы от обычного **malloc**, а хоть и с потолка — ваше дело). Есть некоторый поток байт (пусть он зовётся **stream**), в котором то, с чем мы и боремся. Объект создаётся следующим образом:

```
BCAR *analysis = new BCAR( stream );
```

Обратите внимание — мы создаём объект класса **BCAR**. В первую очередь вызывается **BCAR::new**, который в действительности завуалированный **MemoryManager.largest()**. Мы имеем адрес в свободной памяти, где и создаётся объект **BCAR** и запускается его конструктор **BCAR::BCAR(const unsigned char *)**. В конструкторе по информации из заголовка (полученного из потока **stream**) выясняется точный тип анализа и через глобальный **new** (который не делает ничего) создаётся на месте объекта **BCAR** объект уточнённого типа. Начинает исполняться его конструктор, который в свою очередь вызывает конструктор **BCAR::BCAR()**. Надеемся, стало понятно почему **BCAR::BCAR()** определяется с пустым телом. Потом в конструкторе конкретного объекта вызывается **MemoryManager.alloc(int)**, благодаря чему менеджер памяти получает информацию о точном размере объекта и соответствующим образом правит свои структуры. Уничтожение объектов примитивно, ибо всей необходимой информацией **MemoryManager** располагает:

```
void BCAR::operator delete( void *p ) {
```



```
 MemoryManager.free( (BCAR *)p, ((BCAR *)p)->size() );
}
```

Переносимость этой конструкции очень высока, хотя может понадобиться некоторая правка для весьма экзотических машин. Факты же таковы, что она используется в трёх очень крупных мировых центрах на четырёх аппаратных платформах и пяти операционках.

Но это ещё не всё. Как особо дотошные могли заметить — здесь присутствует виртуальность конструктора, но в любом случае объект конкретного класса всё равно имеет фиксированный размер. А вот объектов одного класса, но разного размера нет. До относительно недавнего времени нас это вполне устраивало, пока не появились некоторые требования, в результате которых нам пришлось сделать и это. Для этого у нас есть два (по меньшей мере) способа. Один — переносимый, но неэстетичный, а второй — непереносимый, но из **common practice**. Эта самая **common practice** состоит в помещении последним членом класса конструкции вида **unsigned char storage[1]** в расчёте на то, что это будет действительно последним байтом во внутреннем представлении объекта и туда можно записать не байт, а сколько надо. Стандарт этого вовсе не гарантирует, но практика распространения нашего детища показала, что для применяемых нами компиляторов оно именно так и есть. И оно работает. Чуть-чуть поправим наши объекты:

```
class Blood: public BCAR {

friend BCAR;

private:
 int bodySize;
 int size() { return sizeof( Blood ) + bodySize; }
 int getSize( const char * );
 sturct BloodAnalysisBody {
  // тут его поля
 } *body;
 Blood( const unsigned char *data ): BCAR() {
  body = (BloodAnalysisBody *) bodyStorage;
  bodySize = getSize( data );
  ::memcpy( bodyStorage, data + sizeof( header ), bodySize );
  MemoryManager.alloc( size() );
 }
 unsigned char bodyStorage[ 1 ];
}
```

Бороться с данными далее придётся через **body->**, но сейчас мы не об этом...

Однако вспомним, что менеджер памяти у нас «свой в доску», и мы можем обойтись действительно переносимой конструкцией. Тело анализа достаточно разместить сразу за самим объектом, статический размер которого нам всегда известен. Ещё чуть-чуть правим:

```
class Blood: public BCAR {

friend BCAR;

private:
 int bodySize;
 int size() { return sizeof( Blood ) + bodySize; }
 int getSize( const unsigned char * );
 struct BloodAnalysisBody {
  // тут его поля
 } *body;
 Blood( const unsigned char *data ): BCAR() {
  body = (BloodAnalysisBody *) ((unsigned char *)this
 + sizeof( Blood ));
  bodySize = getSize( data );
  ::memcpy( body, data + sizeof( header ), bodySize );
  MemoryManager.alloc( size() );
 }
}
```

Данные гарантированно ложатся сразу за объектом в памяти, которую нам дал **MemoryManager** (а он, напомним, даёт нам всегда максимум из того, что имеет), а затем **alloc** соответствующим образом всё подправит.

Глава 11. Чтение исходных текстов

Хочется сразу же дать некоторые определения. Существует программирование для какой-то операционной системы. Программист, который пишет «под Unix», «под Windows», «под DOS», это такой человек, который знает (или догадывается) зачем нужен какой-либо системный вызов, умеет написать драйвер устройства и пытался дизассемблировать код ядра (это не относится к Unix — программист под Unix обычно пытается внести свои изменения в исходный текст ядра и посмотреть что получится).

Существует программирование на каком-то языке программирования. Программист, претендующий на то, что он является программистом «на Бейсике», «на Паскале», «на Си» или «на C++» может вообще ничего не знать о существовании конкретных операционных систем и архитектур, но обязан при этом знать тонкости своего языка программирования, знать о том, какие конструкции языка для чего эффективнее и т.д. и т.п.

Понятно, что «программирование на языке» и «программирование под архитектуру» могут вполне пересекаться. Программист на C под Unix, например. Тем не менее, в особенности среди новичков, часто встречается подмена этих понятий. Стандартный вопрос новичка: «я только начал изучать C++, подскажите пожалуйста, как создать окно произвольной формы?». В общем, нельзя, наверное, изучать сразу же операционную систему и язык программирования. Вопрос же «про окно» правомерен, наверное, только для Java, где работа с окнами находится в стандартной библиотеке языка. Но у C++ своя специфика, потому что даже задав вопрос: «Как напечатать на экране строку Hello, world!?» можно напороться на раздраженное «Покажи где у C++ экран?».

Тем не менее, возвратимся к исходным текстам. Начинающие программисты обычно любят устраивать у себя разнообразные коллекции исходных текстов. Т.е., с десяток эмуляторов терминалов, пятнадцать библиотек пользовательского интерфейса, полсотни DirectX и OpenGL программ «на Си». Кстати сказать, программирование с использованием OpenGL тоже можно отнести к отдельному классу, который ортогонален классам «операционная система» и «язык программирования». Почему-то люди упорно считают, что набрав большое количество различных программ малой, средней и большой тяжести, они решат себе много проблем, связанных с программированием. Это не так — совершенно не понятно, чем на стадии обучения программированию может помочь исходник Quake.

Этому есть простое объяснение. Читать исходные тексты чужих программ (да и своих, в принципе, тоже) очень нелегко. Умение «читать» программы само по себе признак высокого мастерства. Смотреть же на текст программ как на примеры использования чего-либо тоже нельзя, потому что в реальных программах есть очень много конкретных деталей, связанных со спецификой проекта, авторским подходом к решению некоторых задач и просто стилем программирования, а это очень сильно загораживает действительно существенные идеи.

Кроме того, при чтении «исходников», очень часто «программирование на языке» незаметно заменяется «программированием под ОС». Ведь всегда хочется сделать что-то красивое, чем можно удивить родителей или знакомых? А еще хочется сделать так, чтобы «как в Explorer» — трудно забыть тот бум на компоненты flat buttons для Delphi/C++ Builder,

когда только появился Internet Explorer 3.0. Это было что-то страшное, таких компонент появилось просто дикое количество, а сколько программ сразу же появилось с их присутствием в интерфейсе...

Изменять текст существующей программы тоже очень сложно. Дополнить ее какой-то новой возможностью, которая не ставилась в расчет первоначально, сложно вдвойне. Читать текст, который подвергался таким «изменениям» уже не просто сложно — практически невозможно. Именно для этого люди придумали модульное программирование — для того, чтобы сузить, как только это возможно, степень зависимости между собой частей программы, которые пишутся различными программистами, или могут потребоваться в дальнейшем.

Чтение исходных текстов полезно, но уже потом, когда проблем с языком не будет (а до этого момента можно только перенять чужие ошибки), их коллекционирование никак не может помочь в начальном изучении языка программирования. Приемы, которые применяются разработчиками, значительно лучше воспринимаются когда они расписаны без излишних деталей и с большим количеством комментариев, для этого можно посоветовать книгу Джеффа Элджера «C++».

Глава 12. Функция gets()

Функция **gets()**, входящая в состав стандартной библиотеки Си, имеет следующий прототип:

```
char* gets(char* s);
```

Это определение содержится в **stdio.h**. Функция предназначена для ввода строки символов из файла **stdin**. Она возвращает **s** если чтение прошло успешно и **NULL** в обратном случае.

При всей простоте и понятности, эта функция уникальна. Все дело в том, что более опасного вызова, чем этот, в стандартной библиотеке нет... почему это так, а также чем грозит использование **gets()**, мы как раз и попытаемся объяснить далее.

Вообще говоря, для тех, кто не знает, почему использование функции **gets()** так опасно, будет полезно посмотреть еще раз на ее прототип, и подумать. Все дело в том, что для **gets()** нельзя, т.е. совершенно невозможно, задать ограничение на размер читаемой строки, во всяком случае, в пределах стандартной библиотеки. Это крайне опасно, потому что тогда при работе с вашей программой могут возникать различные сбои при обычном вводе строк пользователями. Т.е., например:


```
char name[10];

// ...

puts("Enter you name:");
gets(name);
```

Если у пользователя будет имя больше, чем 9 символов, например, 10, то по адресу (name + 10) будет записан 0. Что там на самом деле находится, другие данные или просто незанятое место (возникшее, например, из-за того, что компилятор соответствующим образом выровнял данные), или этот адрес для программы недоступен, неизвестно.

Все эти ситуации ничего хорошего не сулят. Порча собственных данных означает то, что программа выдаст неверные результаты, а почему это происходит понять будет крайне трудно — первым делом программист будет проверять ошибки в алгоритме и только в конце заметит, что произошло переполнение внутреннего буфера. Надеемся, все знают как это происходит — несколько часов непрерывных «бдений» с отладчиком, а потом через день, «на свежую голову», выясняется что где-то был пропущен один символ...

Опять же, для программиста самым удобным будет моментальное аварийное прекращение работы программы в этом месте — тогда он сможет заменить **gets()** на что-нибудь более «порядочное».

У кого-то может возникнуть предложение просто взять и увеличить размер буфера. Но не надо забывать, что всегда можно ввести строку длиной, превышающий выделенный размер; если кто-то хочет возразить, что случаи имен длиной более чем, например, 1024 байта все еще редки, то перейдем к другому, несколько более интересному примеру возникающей проблемы при использовании **gets()**.

Для это просто подчеркнем контекст, в котором происходит чтение строки.

```
void foo()
{
 char name[10];

 // ...

 puts("Enter you name:");
 gets(name);

 // ...
}
```

Имеется в виду, что теперь **name** расположен в стеке. Надеемся, что читающие эти строки люди имеют представление о том, как обычно выполняется вызов функции. Грубо говоря, сначала в стек помещается адрес возврата, а потом в нем же размещается память под массив **name**. Так что теперь, когда функция **gets()** будет писать за пределами массива, она будет портить адрес возврата из функции **foo()**.

На самом деле, это значит, что кто-то может задать вашей программе любой адрес, по которому она начнет выполнять инструкции.

Немного отвлечемся, потому что это достаточно интересно. В операционной системе Unix есть возможность запускать программы, которые будут иметь привилегии пользователя, отличного от того, кто этот запуск произвел. Самый распространенный пример, это, конечно же, суперпользователь. Например, команды **ps** или **passwd** при запуске любым пользователем получают полномочия root'a. Сделано это потому, что копаться в чужой памяти (для **ps**) может только суперпользователь, так же как и вносить изменения в **/etc/passwd**. Понятно, что такие программы тщательнейшим образом проверяются на отсутствие ошибок — через них могут «утечь» полномочия к нехорошим «хакерам» (существуют и хорошие!). Размещение буфера в стеке некоторой функции, чей код выполняется с привилегиями другого пользователя, позволяет при переполнении этого буфера изменить на что-то осмысленное адрес возврата из функции. Как поместить по переданному адресу то, что требуется выполнить (например, запуск командного интерпретатора), это уже другой разговор и он не имеет прямого отношения к программированию на C или C++.

Принципиально иное: отсутствие проверки на переполнение внутренних буферов очень серьезная проблема. Зачастую программисты ее игнорируют, считая что некоторого заданного размера хватит на все, но это не так. Лучше с самого начала позаботиться о необходимых проверках, чтобы потом не мучиться с решением внезапно возникающих проблем. Даже если вы не будете писать программ, к которым выдвигаются повышенные требования по устойчивости к взлому, все равно будет приятно осознавать, что некоторых неприятностей возможно удалось избежать. А от **gets()** избавиться совсем просто:

```
fgets(name, 10, stdin);
```

Использование функции **gets()** дает лишнюю возможность сбоя вашей программы, поэтому ее использование крайне не рекомендовано. Обычно вызов **gets()** с успехом заменяется **fgets()**.

Глава 13. Свойства

Когда только появилась Delphi (первой версии) и ее мало кто видел, а еще меньше людей пробовали использовать, то в основном сведения об этом продукте фирмы Borland были похожи на сплетни. Помнится такое высказывание: «Delphi расширяет концепции объектно-ориентированного программирования за счет использования свойств».

Те, кто работал с C++ Builder, представляет себе, что такое его свойства. Кроме того, кто хоть как-то знаком с объектно-ориентированным анализом и проектированием, понимает — наличие в языке той или иной конструкции никак не влияет на используемые подходы. Мало того, префикс «**OO**» обозначает не использование ключевых слов **class** или **property** в программах, а именно подход к решению задачи в целом (в смысле ее архитектурного решения). С этой точки зрения, будут использоваться свойства или методы **set** и **get**, совершенно без разницы — главное разграничить доступ.

Тем не менее, свойства позволяют использовать следующую конструкцию: описать функции для чтения и записи значения и связать их одним именем. Если обратиться к этому имени, то в разных контекстах будет использоваться соответственно либо функция для чтения значения, либо функция для его установки.

Перед тем, как мы перейдем к описанию реализации, хотелось бы высказаться по поводу удобства использования. Прямо скажем, программисты — люди. Очень разные. Одним нравится одно, другое это ненавидят... в частности, возможность переопределения операций в C++ часто подвергается нападкам, при этом одним из основных аргументов (в принципе, не лишенный смысла) является то, что при виде такого выражения:

```
a = b;
```

нельзя сразу же сказать, что произойдет. Потому что на оператор присваивания можно «повесить» все что угодно. Самый распространенный пример «неправильного» (в смысле, изменение исходных целей) использования операций являются потоки ввода-вывода, в которых операторы побитового сдвига выполняют роль **printf**. Примерно то же нарекание можно отнести и к использованию свойств.

Тем не менее, перейдем к описанию примера. Итак, нужно оформить такой объект, при помощи которого можно было бы делать примерно следующее:

```
class Test
```

```
{
protected:
 int p_a;
 int& setA(const int& x);
 int getA();
public:
 /* ... */ a;
};

// ...

Test t;
t.a = 10;  // Вызов Test::setA()
int r = t.a; // Вызов Test::getA()
```

Естественный способ — использовать шаблоны. Например, вот так:

```
template<class A, class T,
 T& (A::*setter)(const T&),
 T (A::*getter)()
 >
class property
{
 // ...
};
```

Параметр **A** — класс, к которому принадлежат функции установки и чтения значения свойств (они передаются через аргументы шаблона **setter** и **getter**). Параметр **T** — тип самого свойства (например, **int** для предыдущего примера).

На запись, которая используется для описания указателей на функции, стоит обратить внимание — известно, что многие программисты на C++ и не догадываются о том, что можно получить и использовать адрес функции-члена класса. Строго говоря, функция-член ничем не отличается от обычной, за исключением того, что ей требуется один неявный аргумент, который передается ей для определения того объекта класса, для которого она применяется (это указатель **this**). Таким образом, получение адреса для нее происходит аналогично, а вот использование указателя требует указать, какой конкретно объект используется. Запись **A::*foo** говорит о том, что это указатель на член класса **A** и для его использования потребуется объект этого класса.

Теперь непосредственно весь класс целиком:

```
template<class A, class T,
 T& (A::*setter)(const T&),
 T (A::*getter)()
 >
class property
{
protected:
 A * ptr;
public:
 property(A* p) : ptr(p) { }

 const T& operator= (const T& set) const
 {
 assert(setter != 0);
 return (ptr->*setter)(set);
 }
 operator T() const
 {
 assert(getter != 0);
 return (ptr->*getter)();
 }
};
```

Мы внесли тела функций внутрь класса для краткости (на самом деле, общая рекомендация никогда не захламлять определения классов реализациями функций — если нужен **inline**, то лучше его явно указать у тела подпрограммы, чем делать невозможным для чтения исходный текст. Приходится иногда видеть исходные тексты, в которых определения классов занимают по тысяче строк из-за того, что все методы были описаны внутри класса (как в Java). Это очень неудобно читать.

Думаем, что идея предельно ясна. Использование указателей на функцию-член заключается как раз в строках вида:

```
(ptr->*f)();
```

Указатель внутри свойства — это, конечно же, нехорошо. Тем не менее, без него не обойтись — указатель на объект нельзя будет передать в объявлении класса, только при создании объекта. Т.е., в параметры шаблона его никак не затолкать.

Использовать класс **property** надо следующим образом:

```
class Test
{
 // ...
```

```
 property<Test, int, &Test::setA, &Test::getA> a;

 Test() : a(this) { }
};
```

Компиляторы g++, msvc и bcc32 последних версий спокойно восприняли такое издевательство над собой. Тем не менее, более старые варианты этих же компиляторов могут ругаться на то, что используется незаконченный класс в параметрах шаблона, не понимать того, что берется адрес функций и т.д. Честно говоря, кажется что стандарту это не противоречит.

Мы подошли к главному — зачем все это нужно. А вообще не нужно. Тяжело представить программиста, который решится на использование подобного шаблона — это же просто нонсенс. Опять же, не видно никаких преимуществ по сравнению с обычным вызовом нужных функций и видно только один недостаток — лишний указатель. Так что все, что здесь изложено, стоит рассматривать только как попытку продемонстрировать то, что можно получить при использовании шаблонов.

На самом деле, теоретики «расширения концепций» очень часто забывают то, за что иногда действительно стоит уважать свойства. Это возможность сохранить через них объект и восстановить его (т.е., создать некоторое подобие **persistent object**). В принципе, добавить такую функциональность можно и в шаблон выше. Как? Это уже другой вопрос...

Таким образом, свойства как расширение языка ничего принципиально нового в него не добавляют. В принципе, возможна их реализация средствами самого языка, но использовать такую реализацию бессмысленно. Догадываться же, что такой подход возможен — полезно.

Глава 14. Комментарии

Плохое комментирование исходных текстов является одним из самых тяжелых заболеваний программ. Причем программисты зачастую путают «хорошее» комментирование и «многословное». Согласитесь, комментарии вида:

```
i = 10; // Присваиваем значение 10 переменной i
```

выглядят диковато. Тем не менее, их очень просто расставлять и, поэтому, этим часто злоупотребляют. Хороший комментарий не должен находится внутри основного текста подпрограммы. Комментарий должен располагаться перед заголовком функции; пояснять что и как делает подпрограмма, какие условия накладываются на входные данные и что от

нее можно ожидать; визуально отделять тела функций друг от друга. Потому что при просмотре текста программы зачастую незаметно, где заканчивается одна и начинается другая подпрограмма. Желательно оформлять такие комментарии подобным образом:

```
/*********************************************
 * function
 */
void function()
{
}
```

Этот комментарий можно использовать в любом случае, даже если из названия подпрограммы понятно, что она делает — продублировать ее название не тяжелый труд. Единственное, из опыта следует, что не надо переводить название функции на русский язык; если уж писать комментарий, то он должен что-то добавлять к имеющейся информации. А так видно, что комментарий выполняет декоративную роль.

Чем короче функция, тем лучше. Законченный кусочек программы, который и оформлен в виде независимого кода, значительно легче воспринимается, чем если бы он был внутри другой функции. Кроме того, всегда есть такая возможность, что этот коротенький кусочек потребуется в другом месте, а когда это требуется, программисты обычно поступают методом cut&paste, результаты которого очень трудно поддаются изменениям.

Комментарии внутри тела функции должны быть только в тех местах, на которые обязательно надо обратить внимание. Это не значит, что надо расписывать алгоритм, реализуемый функцией, по строкам функции. Не надо считать того, кто будет читать ваш текст, за идиота, который не сможет самостоятельно сопоставить ваши действия с имеющимся алгоритмом. Алгоритм должен описываться перед заголовком в том самом большом комментарии, или должна быть дана ссылка на книгу, в которой этот алгоритм расписан.

Комментарии внутри тела подпрограммы могут появляться только в том случае, если ее тело все-таки стало длинным. Такое случается, когда, например, пишется подпрограмма для разбора выражений, там от этого никуда особенно не уйдешь. Комментарии внутри таких подпрограмм, поясняющие действие какого-либо блока, не должны состоять из одной строки, а обязательно занимать несколько строчек для того, чтобы на них обращали внимание. Обычно это выглядит следующим образом:

```
{
 /*
 * Комментарий
```

```
 */
}
```

Тогда он смотрится не как обычный текст, а именно как нужное пояснение.

Остальной текст, который желательно не комментировать, должен быть понятным. Названия переменных должны отображать их сущность и, по возможности, быть выполнены в едином стиле. Не надо считать, что короткое имя лучше чем длинное; если из названия короткого не следует сразу его смысл, то это имя следует изменить на более длинное. Кроме того, для C++ обычно используют области видимости для создания более понятных имен. Например, **dictionary::Creator** и **index::Creator**: внутри области видимости можно использовать просто **Creator** (что тоже достаточно удобно, потому что в коде, который имеет отношение к словарю и так ясно, какой **Creator** может быть без префикса), а снаружи используется нужный префикс, по которому смысл имени становится понятным.

Кроме того, должны быть очень подробно прокомментированы интерфейсы. Иерархии классов должны быть широкими, а не глубокими. Все дело в подходе: вы описываете интерфейс, которому должны удовлетворять объекты, а после этого реализуете конкретные виды этих объектов. Обычно все, что видит пользователь — это только определение базового класса такой «широкой» иерархии классов, поэтому оно должно быть максимально понятно для него. Кстати, именно для возврата объектов, удовлетворяющих определенным интерфейсам, используют «умные» указатели.

Еще хорошо бы снабдить каждый заголовочный файл кратким комментарием, поясняющим то, что в нем находится. Файлы реализации обычно смотрятся потом, когда по заголовочным файлам становится понятно, что и где находится, поэтому там такие комментарии возникают по мере необходимости.

Таким образом, комментарии не должны затрагивать конкретно исходный текст программы, они все являются декларативными и должны давать возможность читателю понять суть работы программы не вдаваясь в детали. Все необходимые детали становятся понятными при внимательном изучении кода, поэтому комментарии рядом с кодом будут только отвлекать.

Следовательно, надо предоставить читателю возможность отвлеченного ознакомления с кодом. Под этим подразумевается возможность удобного листания комментариев или распечаток программной документации. Подобную возможность обеспечивают программы автоматизации создания программной документации. Таких программ достаточно много, для Java, например, существует JavaDoc, для C++ — doc++ и doxygen. Все они позволяют сделать по специального вида комментариям качественную документацию с большим количеством перекрестных ссылок и индексов.

Вообще, хотелось бы немного отклониться от основной темы и пофилософствовать. Хороший комментарий сам по себе не появляется. Он является плодом тщательнейшей проработки алгоритма подпрограммы, анализа ситуации и прочее. Поэтому когда становится тяжело комментировать то, что вы хотите сделать, то это означает, скорее всего, то, что вы сами еще плохо сознаете, что хотите сделать. Из этого логичным образом вытекает то, что комментарии должны быть готовы до того, как вы начали программировать.

Это предполагает, что вы сначала пишите документацию, а потом по ней строите исходный текст. Такой подход называется «литературным программированием» и автором данной концепции является сам Дональд Кнут (тот самый, который написал «Искусство программирования» и сделал TeX). У него даже есть программа, которая автоматизирует этот процесс, она называется Web. Изначально она была разработана для Pascal'я, но потом появились варианты для других языков. Например, CWeb — это Web для языка Си.

Используя Web вы описываете работу вашей программы, сначала самым общим образом. Затем описываете какие-то более специфические вещи и т.д. Кусочки кода появляются на самых глубоких уровнях вложенности этих описаний, что позволяет говорить о том, что и читатель, и вы, дойдете до реализации только после того, как поймете действие программы.

Сам Web состоит из двух программ: для создания программной документации (получаются очень красивые отчеты) и создания исходного текста на целевом языке программирования. Кстати сказать, TeX написан на Web'e, а документацию Web делает с использованием TeX'a... как вы думаете, что из них раньше появилось?

Web в основном применятся программистами, которые пишут сложные в алгоритмическом отношении программы. Сам факт присутствия документации, полученной из Web-исходника, упрощает ручное доказательство программ (если такое проводится). Тем не менее, общий подход к программированию должен быть именно такой: сначала думать, потом делать. При этом не надо мерить работу программиста по количеству строк им написанных.

Несмотря на то, что использование Web'а для большинства «совсем» прикладных задач несколько неразумно (хотя вполне возможно, кто мешает хорошо программировать?), существуют более простые реализации той же идеи.

Рассмотрим **doxygen** в качестве примера.

```
/**
 * \bref Краткое описание.
 *
 * Этот класс служит для демонстрации
 * возможностей автодокументации.
 */
class AutoDoc
{
public:
 int foo() const; //!< Просто функция. Возвращает ноль.

 /**
  * \brief Другая просто функция.
  *
  * Назначение непонятно: возвращает ноль. foo() -
  * более безопасная реализация возврата нуля.
  *
  * \param ignored - игнорируется.
  *
  * \warning может отформатировать жесткий диск.
  *
  * \note форматирование происходит только по пятницам.
  *
  * \see foo().
  */
 int bar(int ignored);
};
```

Комментарии для **doxygen** записываются в специальном формате. Они начинаются с «**/****», «**/*!**» или «**//!<**». Весь подобный текст **doxygen** будет использовать в создаваемой документации. Он автоматически распознает имена функций или классов и генерирует ссылки на них в документации (если они присутствуют в исходном тексте). Внутри комментариев существует возможность использовать различные стилевые команды (пример достаточно наглядно демонстрирует некоторые из них), которые позволяют более тщательным образом структурировать документацию.

doxygen создает документацию в форматах:

- html;
- LaTeX;
- RTF;
- man.

Кроме того, из документации в этих форматах, можно (используя сторонние утилиты) получить документацию в виде MS HTML Help (наподобие MSDN), PDF (через LaTeX).

В общем использовать его просто, а результат получается очень хороший.

Кроме того (раз уж зашла об этом речь), существует такая программа, называется pcGRASP, которая позволяет «разрисовать» исходный текст. В чем это заключается: все видели красивые и очень бесполезные блок-схемы. pcGRASP делает кое-что в этом духе: к исходному тексту он добавляет в левое поле разные линии (характеризующие уровень вложенности), ромбики (соответствующие операторам выбора), стрелочки (при переходе с одного уровня вложенности на другой, например, **return**) и т.д. Самое приятное, так это распечатки, полученные таким образом. Учитывая то, что **indent** (отступы) pcGRASP делает сам (не ориентируясь на то, как оно было), это делает подобные распечатки исключительно ценными.

В заключении, еще раз отметим, что комментарии надо писать так, чтобы потом самому было бы приятно их читать. Никакого распускания соплей в исходном тексте — тот, кто будет читать его, прекрасно знает что делает операция присваивания и нечего ему это объяснять.

Красиво оформленная программная документация является большим плюсом, по крайней мере потому, что люди, принимающие исходный текст, будут рады увидеть текст с гиперссылками. Тем более, что не составляет труда подготовить исходный текст к обработке утилитой наподобие **doxygen**.

Глава 15. Веб-программирование

Популярный нынче термин «веб-программирование» обычно подразумевает под собой программирование, в лучшем случае, на perl, в худшем — на PHP, в совсем тяжелом — на JavaScript. Ничуть не пытаем-

ся обидеть людей, которые этим занимаются, просто смешно выделять «программирование на Perl» в отдельную нишу, прежде всего потому что специфика его использования отнюдь не в «веб-программировании».

Кроме того, вообще тяжело понять, чем принципиально отличается создание CGI-программ от «просто программ», кроме использования специализированного интерфейса для получения данных.

Тем не менее, «веб-программирование» действительно существует. Заключается оно не в генерации на лету HTML-страничек по шаблонам на основании информации из базы данных, потому что это относится именно к использованию БД. «Веб-программирование» — это работа с сетевыми протоколами передачи данных, да и сетями вообще. Более точно, это «программирование для сетей, основанных на TCP/IP». А подобное программирование подразумевает прежде всего передачу данных, а не их обработку — скажите, что и куда передает CGI-программа по сети? Данные передает веб-сервер, а CGI используется как метод расширения возможностей сервера.

Настоящее веб-программирование — это программирование веб-серверов и веб-клиентов. Т.е., написать Apache, Internet Explorer или lynx — это веб-программирование.

Некоторые могут сказать, что мы слишком строго подошли к программированию CGI-приложений и, если копать дальше, то веб-программирование в том смысле, в котором мы его определили только что, является всего-навсего обработкой устройств ввода-вывода (к коим относится в одинаковой степени и сетевая карта, и клавиатура). Ну... да, это будет законный упрек. Только мы не собираемся так далеко заходить, просто хочется точнее определить термин «веб-программирование». Все дело в том, что генерация страниц «на лету» подразумевает то, что они будут отдаваться по сети, но это совершенно не обязательно. Неужели что-то принципиальным образом изменится в CGI-приложении, если его результаты будут сохраняться на жесткий диск? А внутри того, что подсовывается на вход PHP-интерпретатору? Ничего не изменится. Вообще. Для них главным является корректная установка нужных переменных среды окружения, а тот факт, подключен компьютер к сети, или нет, их не волнует.

Проблему передачи или получения данных через TCP, конечно же, тоже можно аналогичным образом развернуть и сказать, что с появлением интерфейса сокетов (BSD sockets) передача данных на расстояние ничем принципиально не отличается от работы с файлами на локальном диске.

В принципе это так. Потому что, если откинуть использование функций, связанных с инициализацией сокетов, в остальном с ними можно общаться как с традиционными файловыми дескрипторами.

Все это хорошо, но до некоторой поры. Все дело в том, что два компьютера могут находится рядом и быть соединены всего-лишь одним кабелем, а могут отстоять друг от друга на тысячи километров. И хотя скорость передачи данных в пределах одного кабеля очень большая, но при использовании различных устройств для соединения кабелей друг с другом, происходит замедление передачи данных и чем «умнее» будет устройство, тем медленнее через него будут передаваться данные.

Это первое отличие. При работе с файловой системой программист никогда не думает о том, с какой скоростью у него считается файл или запишется (точнее, думает, но реже), буферизация обычно уже реализована на уровне операционной системы или библиотеки языка программирования, а при работе с сетью задержки могут быть очень велики и в это время программа будет ожидать прихода новых данных, вместо того, чтобы сделать что-либо полезное. Таким образом приходит необходимость разбивать программу на совокупность потоков и программирование превращается в ад, потому что ничего хорошего это не принесет, только лишние проблемы. Связано это с тем, что разделение программы на несколько одновременно выполняющихся потоков требует очень большой внимательности и поэтому чревато серьезными ошибками. А перевод существующей однопоточной программы в многопоточную, вообще занятие неблагодарное.

Второе отличие, в принципе, вытекает из первого, но стоит несколько отдельно, потому что это требование более жесткое. Все дело в том, что передача данных обычно не ограничивается одним файлом или одним соединением. Обычно сервер обслуживает одновременно несколько клиентов или клиент одновременно пытается получить доступ одновременно к нескольким ресурсам (они так выкачиваются быстрее, чем по одиночке). Тем самым приходиться открывать одновременно несколько сокетов и пытаться одновременно обработать их все.

Для подобной работы, в принципе, не требуется реальная многопоточность, существует такой вызов (или подобный ему в других операционных системах), называемый **select()**, который переводит программу в режим ожидания до тех пор, пока один (или несколько) файловых дескрипторов не станет доступным для чтения или записи. Это позволяет без введения нескольких потоков обрабатывать данные из нескольких соединений по мере их прихода.

Третье отличие, по сути, относится к серверным приложениям и выражается в повышенных требованиях к производительности. Все дело

в том, что когда один и тот же интернет-ресурс запрашивается большим количеством пользователей одновременно, то становится очень важна скорость реакции на него. В принципе, тут, если использовать CGI-приложения, на них тоже будет распространяться это требование, но в таких ситуациях обычно вставляют нужные обработчики непосредственно в сервер для повышения производительности.

Для облегчения написания веб-приложений (именно веб-приложений!) на языках C и C++ была написана библиотека **libwww**. Она реализует псевдопотоковую событийную модель программы.

Под псевдопотоками понимается как раз использование **select()** для обработки большого количества запросов, а не введение для каждого из них настоящего потока операционной системы. Событийная модель предполагает то, что приложение запускает цикл обработки событий, поступающих от **libwww**, и в дальнейшем работа программы основывается на выполнении предоставленных обработчиков событий из цикла.

Приложение в этом случае управляется поступающими или инициированными запросами. Кроме этого псевдопараллельного «фетчера» (от слова fetch), в **libwww** присутствует большое количество готовых обработчиков, которые помогают решать типовые задачи, появляющиеся вместе с «веб-программированием». Например, существует набор «переводчиков», которые позволяют, например, из потока «**text/html**» сделать поток «**text/present**» (это в терминологии **libwww**, «**present**» означает вид, который будет представлен пользователю). Для этого используется собственный html-парсер (основанный на sgml-парсере с html dtd).

Таким образом, типичное веб-приложение, написанное при помощи **libwww**, выглядит как набор подпрограмм, обрабатывающие различные сообщения. Это позволяет написать как клиентские, так и серверные программы.

В качестве примера использования **libwww**, можно вспомнить текстовый браузер **lynx**, который написан при помощи этой библиотеки.

Впрочем, не будем распространяться долго об архитектуре **libwww**, желающие могут посмотреть документацию. Теперь нам хотелось бы пройтись по недостаткам.

libwww со временем становится все сложнее и сложнее по причине увеличивающегося набора стандартных обработчиков. Зачастую, достаточно простая задача, но которая не предусмотрена изначально как типовая, может потребовать нескольких часов изучения документации в поисках того, какие обработчики и где должны для этого присутствовать.

Документация на библиотеку не отличается подробностью и, как следствие, понятностью; она сводится к двум-трем строкам комментариев к программным вызовам, при этом местами документация просто устарела. Совет: если вы используете **libwww**, то обязательно смотрите исходный текст, по нему станет многое понятно. В качестве примера, можно привести такую вещь. Для того, чтобы отследить контекст разбора HTML, вводится специальный объект типа **HText**, который используется в обработчиках событий от парсера в качестве «внешней памяти». Это как указатель **this** в C++, т.е. реализация объектов на языке Си. Тип **HText** целиком предоставляется пользователем, вместе с callback-функциями создания и удаления. Прежде чем его использовать, надо зарегистрировать эти функции (причем обязательно парой, т.е. надо обязательно указать и функцию создания объекта, и функцию разрушения), которые, если следовать документации, будут вызваны, соответственно, для создания и удаления объекта при начале и конце разбора соответственно.

Это все пока что «хорошо». «Плохо» — функция разрушения не вызывается никогда. В частности, в документации этого нет, но видно по исходному тексту, где в том месте, где должен был находится вызов удаляющей функции, находится комментарий, в котором написано, что забота об удалении ложится на плечи приложения, а не библиотеки **libwww**. Кто при этом мешает вызвать переданную функцию, не понятно. Кстати сказать, примеры использования объекта **HText** в поставляемых с **libwww** приложениях из каталога **Examples**, рассчитаны на то, что функция удаления будет вызвана.

И это не единственное забавное место в библиотеке. Все это решаемо, конечно, но лучше если вы будете сразу же смотреть исходный текст **libwww**, по нему значительно больше можно понять, чем по предоставленной документации.

Итак, использование **libwww** позволяет быстро написать типичное web-приложение, например, «робота», который выкачивает определенные документы и каким-то образом их анализирует. Тем не менее, использовать **libwww** в своих программах не всегда просто из-за проблем с документацией на нее.

Глава 16. Ошибки работы с памятью

Когда программа становится внушительной по своему содержанию (то есть, не по количеству строчек, а по непонятности внутренних

связей), то ее поведение становится похожим на поведение настоящего живого существа. Такое же непредсказуемое... впрочем, кое что все-таки предсказать можно: работать оно не будет. Во всяком случае, сразу.

Программирование на C и C++ дает возможность допускать такие ошибки, поиск которых озадачил бы самого Шерлока Холмса. Вообще говоря, чем загадочнее ведет себя программа, тем проще в ней допущена ошибка. А искать простые ошибки сложнее всего, как это ни странно; все потому, что сложная ошибка обычно приводит к каким-то принципиальным неточностям в работе программы, а ошибка простая либо превращает всю работу в «бред пьяного программиста», либо всегда приводит к одному и тому же: *segmentation fault*.

И зря говорят, что если ваша программа выдала фразу **core dumped**, то ошибку найти очень просто: это, мол, всего лишь обращение по неверному указателю, например, нулевому. Обращение-то, конечно же, есть, но вот почему в указателе появилось неверное значение? Откуда оно взялось? Зачастую на этот вопрос не так просто ответить.

В Java исключены указатели именно потому, что работа с ними является основным источником ошибок программистов. При этом отсутствие инициализации является одним из самых простых и легко отлавливаемых вариантов ошибок.

Самые трудные ошибки появляются, как правило, тогда, когда в программе постоянно идут процессы выделения и удаления памяти. То есть, в короткие промежутки времени появляются объекты и уничтожаются. В этом случае, если где-нибудь что-нибудь некорректно «указать», то «core dumped», вполне вероятно, появится не сразу, а лишь через некоторое время. Все дело в том, что ошибки с указателями проявляются обычно в двух случаях: работа с несуществующим указателем и выход за пределы массива (тоже в конечном итоге сводится к несуществующему указателю, но несколько чаще встречается).

Загадки, возникающие при удалении незанятой памяти, одни из самых трудных. Выход за границы массива, пожалуй, еще сложнее.

Представьте себе: вы выделили некоторый буфер и в него что-то записываете, какие-то промежуточные данные. Это критическое по времени место, поэтому тут быть не может никаких проверок и, ко всему прочему, вы уверены в том, что исходного размера буфера хватит на все, что в него будут писать. Не хотелось бы торопиться с подобными утверждениями: а почему, собственно, вы так в этом уверены? И вообще, а вы уверены в том, что правильно вычислили этот самый размер буфера?

Ответы на эти вопросы должны у вас быть. Мало того, они должны находиться в комментариях рядом с вычислением размера буфера и его заполнением, чтобы потом не гадать, чем руководствовался автор, когда написал:

```
char buf[100];
```

Что он хотел сказать? Откуда взялось число 100? Совершенно непонятно.

Теперь о том, почему важно не ошибиться с размерами. Представьте себе, что вы вышли за пределы массива. Там может «ничего не быть», т.е. этот адрес не принадлежит программе и тогда в нормальной операционной системе вы получите соответствующее «матерное» выражение. А если там что-то было?

Самый простой случай — если там были просто данные. Например, какое-нибудь число. Тогда ошибка, по крайней мере, будет видна почти сразу... а если там находился другой указатель? Тогда у вас получается наведенная ошибка очень высокой сложности. Потому что вы будете очень долго искать то место, где вы забыли нужным образом проинициализировать этот указатель...

Мало того, подобные «наведенные» ошибки вполне могут вести себя по-разному не только на разных тестах, но и на одинаковых.

А если еще программа «кормится» данными, которые поступают непрерывно... и еще она сделана таким образом, что реагирует на события, которые каким-то образом распределяются циклом обработки событий... тогда все будет совсем плохо. Отлаживать подобные программы очень сложно, тем более что зачастую, для того, чтобы получить замеченную ошибку повторно, может потребоваться несколько часов выполнения программы. И что делать в этих случаях?

Поиск таких ошибок более всего напоминает шаманские пляски с бубном около костра, не зря этот образ появился в программистском жаргоне. Потому что программист, измученный бдениями, начинает просто случайным образом «удалять» (закомментировав некоторую область, или набрав **#if 0 ... #endif**) блоки своей программы, чтобы посмотреть, в каком случае оно будет работать, а в каком — нет.

Это действительно напоминает шаманство, потому что иногда программист уже не верит в то, что, например, «от перестановки мест сумма слагаемых не меняется» и запросто может попытаться переставить и проверить результат... авось?

А вот теперь мы подошли к тому, что в шаманстве тоже можно выделить систему. Для этого достаточно осознать, что большинство зага-

дочных ошибок происходят именно из-за манипуляций с указателями. Поэтому, вместо того чтобы переставлять местами строчки программы, можно просто попытаться для начала закомментировать в некоторых особенно опасных местах удаление выделенной памяти и посмотреть что получится.

Кстати сказать, отладка таких моментов требует (именно требует) наличия отладочной информации во всех используемых библиотеках, так будет легче работать. Так что, если есть возможность скомпилировать библиотеку с отладочной информацией, то так и надо делать — от лишнего можно будет избавиться потом.

Если загадки остались, то надо двинуться дальше и проверить индексацию массивов на корректность. В идеале, перед каждым обращением к массиву должна находиться проверка инварианта относительно того, что индекс находиться в допустимых пределах. Такие проверки надо делать отключаемыми при помощи макросов **DEBUG/RELEASE** с тем, чтобы в окончательной версии эти дополнительные проверки не мешались бы (этим, в конце-концов, C отличается от Java: хотим — проверяем, не хотим — не проверяем). В этом случае вы значительно быстрее сможете найти глупую ошибку (а ошибки вообще не бывают умными; но найденные — глупее оставшихся)).

На самом деле, в C++ очень удобно использовать для подобных проверок шаблонные типы данных. То есть, сделать тип «массив», в котором переопределить необходимые операции, снабдив каждую из них нужными проверками. Операции необходимо реализовать как **inline**, это позволит не потерять эффективность работы программы. В то же самое время, очень легко будет удалить все отладочные проверки или вставить новые. В общем, реализация своего собственного типа данных **Buffer** является очень полезной.

Кстати, раз уж зашла об этом речь, то абзац выше является еще одним свидетельством того, что C++ надо использовать «полностью» и никогда не писать на нем как на «усовершенствованном Си». Если вы предпочитаете писать на Си, то именно его и надо использовать. При помощи C++ те же задачи решаются совсем по другому.

Глава 17.
Создание графиков с помощью ploticus

Есть такая программа, предназначенная для создания графиков различных видов из командной строки, называется **ploticus**. Программа сама по себе достаточно удобная — потому что иногда очень полезно автоматизировать генерацию различных графических отчетов, а тут без командной строки и вызова программ из скриптов не обойтись.

Нет, таких программ великое множество, но **ploticus** отличается от них очень удобным преимуществом: он «глупый». То есть, его можно, например, заставить разместить надпись на рисунке с точностью до пиксела... иногда это нужно.

Но разговор не об удобстве этой программы. Просто иногда требуется использовать **ploticus**, но при этом немного доработанный напильником.

Сначала немного лирики. **Ploticus**, для того, чтобы построить график, читает некоторый файл, в котором находится определение этого самого графика (скрипт, так сказать). Этот файл обладает очень простой грамматикой. Мало того, **ploticus** умеет организовывать программный канал (хотя, кто этого не умеет?) и читать данные оттуда, как результат выполнения другой программы.

Так вот о чтении этого файла мы и хотим немного рассказать. Итак, подпрограммы чтения данных в **ploticus** разбиты на некоторые логические блоки, исходя из структуры самого файла с данными. Ну это понятно и логично. Не особенно понятно другое: каждая подпрограмма (парсер) на вход воспринимает название файла, в котором находится содержимое.

В итоге, основная подпрограмма сначала разбивает файл на блоки, содержимое этих файлов копирует (!) во временные файлы (!!), которые подсовывает на вход другим подпрограммам. Это, конечно, уже достаточно оригинально, хотя задумка автора ясна — он хотел сделать так, чтобы в этих местах на вход подпрограммам чтения данных можно было бы подсунуть имя программы, которая эти данные бы сгенерировала. Тем не менее, можно было бы сделать значительно красивее, чем создавать кучу временных текстовых файлов.

Эти «подпарсеры» реализованы... аналогичным образом. Т.е., автор не смущаясь разбивает подсунутые файлы еще на кусочки и записывает их в другие временные файлы, которые потом читает.

В принципе, все эти места как раз и требовали вмешательства напильника, потому что хотелось иметь программный интерфейс ко всему этому хозяйству и, желательно, чтобы данные не покидали оперативной памяти.

Все написанное выше уже смешно. Но кусочек кода, который приведен, может довести программиста до истерического смеха.

Вот он (с купюрами):

```
/*
 * ...
 */
else if( stricmp( attr, "data" )==0 ) {
  FILE *tfp;
  sprintf( datafile, "%s_D", Tmpname );
  getmultiline( "data", lineval, fp, MAXBIGBUF, Bigbuf );
  tfp = fopen( datafile, "w" );
  if( tfp == NULL ) return( Eerr( 294, "Cannot open tmp data
file", datafile ));
  fprintf( tfp, "%s", Bigbuf );
  fclose( tfp );
  }
/*
 * ...
 */
```

Это как раз и есть выделение секции с данными и запись ее во временный файл. Вообще, использование **fprintf** с шаблоном **"%s"** уже смотрится очень оригинально, но то, что идет 80 строками ниже еще более необычно:

```
/*
 * ...
 */
if( standardinput || strcmp( datafile, "-" ) ==0 ) { /* a file of
"-" means read from stdin */
  dfp = stdin;
  goto PT1;
  }
if( strlen( datafile ) > 0 ) sprintf( command, "cat %s", datafile
);
if( strlen( command ) > 0 ) {
  dfp = popen( command, "r" );
  if( dfp == NULL ) {
 Skipout = 1;
 return( Eerr( 401, "Cannot open", command ) );
 }

  PT1:
/*
 * ...
 */
```


Обратите внимание на строчку:

```
if( strlen( datafile ) > 0 ) sprintf( command, "cat %s", datafile
);
```

и следующую за ней:

```
dfp = popen( command, "r" );
```

Честно говоря, это впечатляет. Очень впечатляет... при этом, совершено не понятно что мешало использовать обычный **fopen()** для этого (раз уж так хочется), раз уж есть строки вида:

```
dfp = stdin;
goto PT1;
```

В общем, дикость. Если кто-то не понял, то объясним то, что происходит, на пальцах: читается секция «**data**» и ее содержимое записывается в файл, название которого содержится в **datafile**. Потом, проверяется название этого файла, если оно равно «**-**», то это значит, что данные ожидаются со стандартного файла ввода, **stdin**. Если же нет, то проверяется длина строки, на которую указывает **datafile**. Если она ненулевая, то считается, что команда, результаты работы которой будут считаться за входные данные, это «**cat datafile**». Если же ненулевая длина у другого параметра, **command**, то его значение принимается за выполняемую команду. После всех этих манипуляций, открывается программный канал (**pipe**) при помощи **popen()**, результатом которой является обычный указатель на структуру **FILE** (при его помощи можно использовать обычные средства ввода-вывода).

Вам это не смешно?

Глава 18. Автоматизация и моторизация приложения

Программирование давно стало сплавом творчества и строительства, оставляя в прошлом сугубо научно-шаманскую окраску ремесла. И если такой переход уже сделан, то сейчас можно обозначить новый виток – ломание барьеров API и переход к более обобщенному подходу в проектировании, выход на новый уровень абстракции. Немало этому способствовал Интернет и его грандиозное творение – XML. Сегодня ключ к успеху приложения сплавляется из способности его создателей обеспечить максимальную совместимость со стандартами и в то же время масштабируемость. Придумано такое количество различных технологий для связи приложений и повторного использования кода, что сегодня прикладные программы не могут жить без такой «поддержки». Под термином «автоматизация» понимается настоящее оживление приложений,

придание им способности взаимодействовать с внешней средой, предоставление пользователю максимального эффекта в работе с приложениями. Не равняясь на такие гранды технической документации, как MSDN, тем не менее, хотим указать на путь, по которому сегодня проектируются современные приложения.

Автоматизация как есть

Автоматизация (Automation) была изначально создана как способ для приложений (таких как Word или Excel) предоставлять свою функциональность другим приложениям, включая скрипт-языки. Основная идея заключалась в том, чтобы обеспечить наиболее удобный режим доступа к внутренним объектам, свойствам и методам приложения, не нуждаясь при этом в многочисленных «хедерах» и библиотеках.

Вообще, можно выделить два вида автоматизации – внешнюю и внутреннюю. Внешняя автоматизация – это работа сторонних приложений с объектной моделью вашей программы, а внутренняя – это когда сама программа предоставляет пользователю возможность работы со своей объектной структурой через скрипты. Комбинирование первого и второго вида представляется наиболее масштабируемым решением и именно о нем пойдет речь.

Для начала обратим внимание на самое дно – интерфейсы COM. Если термин «интерфейс» в этом контексте вам ничего не говорит, то представьте себе абстрактный класс без реализации – это и есть интерфейс. Реальные объекты наследуются от интерфейсов. Компоненты, наследующиеся от интерфейса IUnknown, называются COM-объектами. Этот интерфейс содержит методы подсчета ссылок и получения других интерфейсов объекта.

Автоматизация базируется на интерфейсе IDispatch, наследующегося от IUnknown. IDispatch позволяет запускать методы и обращаться к свойствам вашего объекта через их символьные имена. Интерфейс имеет немного методов, которые являются тем не менее довольно сложными в реализации. К счастью, существует множество шаблонных классов, предлагающих функциональность интерфейса IDispatch, поэтому для создания объекта, готового к автоматизации, необходимо всего лишь несколько раз щелкнуть мышкой в ClassWizard Visual C++.

Что касается способа доступа и динамического создания ваших внутренних **dispatch** объектов, то тут тут тоже все довольно просто – данные об объекте хранятся в реестре под специальным кодовым именем, которое называется **ProgId**.

Например, **progid** программы Excel – **Excel.Application**. Создать в любой процедуре на VBScript достаточно легко – надо только вызвать функцию **CreateObject**, в которую передать нужный **ProgID**. Функция вернет указатель на созданный объект.

MFC

В MFC существует специальный класс, под названием **CCmdTarget**. Наследуя свои классы от **CCmdTarget**, вы можете обеспечить для них необходимую функциональность в **dispatch** виде – как раз как ее понимают скрипты. При создании нового класса в **ClassWizard** (**View ⇨ ClassWizard ⇨ Add Class ⇨ New**), наследуемого от **CCmdTarget**, просто щелкните на кнопке **Automation** или **Creatable by ID**, чтобы обеспечить возможность создания экземпляра объекта по его **ProgID**. Заметим, что для программ, реализующих внутреннюю автоматизацию, это не нужно. Для приложений, реализующих внешнюю и смешанную автоматизацию, это необходимо для «корневых» объектов.

COM-объекты можно создавать только динамически. Это связано с тем, что объект может использоваться несколькими приложениями одновременно, а значит, удаление объекта из памяти не может выполнить ни одно из них. Разумно предположить, что объект сам должен отвечать за свое удаление. Такой механизм реализован при помощи механизма ссылок (reference count). Когда приложение получает указатель на объект, он увеличивает свой внутренний счетчик ссылок, а когда приложение освобождает объект – счетчик ссылок уменьшается. При достижении счетчиком нуля, объект удаляет сам себя. Если наш объект был создан по **ProgID** другим приложением, то программа **CTestApp** (другими словами, Automation-Server) не завершится до тех пор, пока счетчик ссылок **CTestAutomatedClass** не станет равным нулю.

Создаваемые через **ProgID** COM-объекты, обычно являются Proxy-компонентами. Реально они не содержат никакой функциональности, но имеют доступ к приложению и его внутренним, не доступным извне, функциям. Хотя можно организовать все таким образом, чтобы всегда создавался только один COM-объект, а все остальные вызовы на создание возвращали указатели на него.

Метод интерфейса **CCmdTarget GetIDispatch()**, позволяет получить указатель на реализованный интерфейс **IDispatch**. В параметрах можно указать, нужно ли увеличивать счетчик ссылок или нет.

Оболочка из классов для COM

Программировать с использованием COM настолько трудно, что вы не должны даже пробовать это без MFC. Правильно или неправиль-

но? Абсолютная чушь! Рекламируемые OLE и его преемник COM имеют элегантность гиппопотама, занимающегося фигурным катанием. Но размещение MFC на вершине COM подобно одеванию гиппопотама в клоунский костюм еще больших размеров.

Итак, что делать программисту, когда он столкнется с потребностью использовать возможности оболочки Windows, которые являются доступными только через интерфейсы COM?

Для начала, всякий раз, когда вы планируете использовать COM, вы должны сообщить системе, чтобы она инициализировала COM подсистему. Точно так же всякий раз, когда вы заканчиваете работу, вы должны сообщить системе, чтобы она выгрузила COM. Самый простой способ это сделать заключается в определении объекта, конструктор которого инициализирует COM, а деструктор выгружает ее. Самое лучшее место, для внедрения данного механизма — это объект **Controller**.

```
class Controller
{
public:
 Controller (HWND hwnd, CREATESTRUCT * pCreate);
 ~Controller ();
 // ...
private:
 UseCom _comUser; // I'm a COM user

 Model  _model;
 View  _view;
 HINSTANCE _hInst;
};
```

Этот способ гарантирует, что COM подсистема будет проинициализирована прежде, чем к ней будут сделаны любые обращения и что она будет освобождена после того, как программа осуществит свои разрушения (то есть, после того, как «Вид» и «Модель» будут разрушены).

Класс **UseCom** очень прост.

```
class UseCom
{
public:
 UseCom ()
 {
  HRESULT err = CoInitialize (0);
  if (err != S_OK)
throw "Couldn't initialize COM";
 }
 ~UseCom ()
 {
  CoUninitialize ();
 }
};
```


Пока не было слишком трудно, не так ли? Дело в том, что мы не коснулись главной мерзости COM программирования — подсчета ссылок. Вам должно быть известно, что каждый раз, когда вы получаете интерфейс, его счетчик ссылок увеличивается. И вам необходимо явно уменьшать его. И это становится более чем ужасным тогда, когда вы начинаете запрашивать интерфейсы, копировать их, передавать другим и т.д. Но ловите момент: мы знаем, как управляться с такими проблемами! Это называется управлением ресурсами. Мы никогда не должны касаться интерфейсов COM без инкапсуляции их в интеллектуальных указателях на интерфейсы. Ниже показано, как это работает.

```
template <class T>class SIfacePtr
{
public:
 ~SIfacePtr ()
 {
  Free ();
 }
 T * operator->() { return _p; }
 T const * operator->() const { return _p; }
 operator T const * () const { return _p; }
 T const & GetAccess () const { return *_p; }

protected:
 SIfacePtr () : _p (0) {}
 void Free ()
 {
  if (_p != 0)
_p->Release ();
  _p = 0;
 }

 T * _p;
private:
 SIfacePtr (SIfacePtr const & p) {}
 void operator = (SIfacePtr const & p) {}
};
```

Не волнуйте, что этот класс выглядит непригодным (потому что имеет защищенный конструктор). Мы никогда не будем использовать

его непосредственно. Мы наследуем от него. Между прочим, это удобный прием: создайте класс с защищенным пустым конструктором, и осуществите наследование от него. Все наследующие классы должны обеспечивать их внутреннюю реализацию своими открытыми конструкторами. Поскольку вы можете иметь различные классы, наследующие от **SIfacePtr**, они будут отличаться по способу получения, по их конструкторам, рассматриваемым интерфейсам.

Закрытый фиктивный копирующий конструктор и оператор «=» не всегда необходимы, но они сохранят вам время, потраченное на отладку, если по ошибке вы передадите интеллектуальный интерфейс по значению вместо передачи по ссылке. Это больше невозможно. Вы можете освободить один и тот же интерфейс, дважды и это будет круто отражаться на подсчете ссылок COM. Верьте, это случается. Как это часто бывает, компилятор откажется передавать по значению объект, который имеет закрытую копию конструктора. Он также выдаст ошибку, когда вы пробуете присвоить объект, который имеет закрытый оператор присваивания.

Оболочки API часто распределяет память, используя свои собственные специальные программы распределения. Это не было бы настолько плохо, если бы не предположение, что они ожидают от вас освобождения памяти с использованием той же самой программы распределения. Так, всякий раз, когда оболочка вручает нам такой сомнительный пакет, мы оборачиваем его в специальный интеллектуальный указатель.

```
template <class T>
class SShellPtr
{
public:
 ~SShellPtr ()
 {
  Free ();
  _malloc->Release ();
 }
 T * weak operator->() { return _p; }
 T const * operator->() const { return _p; }
 operator T const * () const { return _p; }
 T const & GetAccess () const { return *_p; }

protected:
 SShellPtr () : _p (0)
 {
  // Obtain malloc here, rather than
```



```
  // in the destructor.
  // Destructor must be fail-proof.
  // Revisit: Would static IMalloc * _shellMalloc work?
  if (SHGetMalloc (& _malloc) == E_FAIL)
throw Exception "Couldn't obtain Shell Malloc";
 }
 void Free ()
 {
  if (_p != 0)
_malloc->Free (_p);
  _p = 0;
 }
 T * _p;
 IMalloc * _malloc;
private:
 SShellPtr (SShellPtr const & p) {}
 void operator = (SShellPtr const & p) {}
};
```

Обратите внимание на использование приема: класс **SShellPtr** непосредственно не пригоден для использования. Вы должны наследовать от него подкласс и реализовать в нем соответствующий конструктор.

Обратите также внимание, нет уверенности, может ли **_shellMalloc** быть статическим элементом **SShellPtr**. Проблема состоит в том, что статические элементы инициализируются перед **WinMain**. Из-за этого вся COM система может оказаться неустойчивой. С другой стороны, документация говорит, что вы можете безопасно вызывать из другой API функции **CoGetMalloc** перед обращением к **CoInitialize**. Это не говорит о том, может ли **SHGetMalloc**, который делает почти то же самое, также вызываться в любое время в вашей программе. Подобно многим другим случаям, когда система ужасно разработана или задокументирована, только эксперимент может ответить на такие вопросы.

Между прочим, если вы нуждаетесь в интеллектуальном указателе, который использует специфическое распределение памяти для COM, то получите его, вызывая **CoGetMalloc**. Вы можете без опаски сделать этот **_malloc** статическим элементом и инициализировать его только один раз в вашей программе (ниже **SComMalloc:: GetMalloc** тоже статический):

```
IMalloc * SComMalloc::_malloc = SComMalloc::GetMalloc ();
IMalloc * SComMalloc::GetMalloc ()
{
 IMalloc * malloc = 0;
 if (CoGetMalloc (1, & malloc) == S_OK)
```

```
  return malloc;
 else
  return 0;
}
```

Это — все, что надо знать, чтобы начать использовать оболочку Windows и ее COM интерфейсы. Ниже приводится пример. Оболочка Windows имеет понятие Рабочего стола, являющегося корнем «файловой» системы. Вы обращали внимание, как Windows приложения допускают пользователя, просматривают файловую систему, начинающуюся на рабочем столе? Этим способом вы можете, например, создавать файлы непосредственно на вашем рабочем столе, двигаться между дисководами, просматривать сетевой дисковод, и т.д. Это, в действительности, Распределенная Файловая система (PMDFS — poor man's Distributed File System). Как ваше приложение может получить доступ к PMDFS? Просто. В качестве примера напишем код, который позволит пользователю выбирать папку, просматривая PMDFS. Все, что мы должны сделать — это овладеть рабочим столом, позиционироваться относительно его, запустить встроенное окно просмотра и сформировать путь, который выбрал пользователь.

```
char path [MAX_PATH];
path [0] = '\0';
Desktop desktop;
ShPath browseRoot (desktop, unicodePath);
if (browseRoot.IsOK ())
{
 FolderBrowser browser (hwnd,
  browseRoot,
  BIF_RETURNONLYFSDIRS,
  "Select folder of your choice");
 if (folder.IsOK ())
 {
  strcpy (path, browser.GetPath ());
 }
}
```

Давайте, запустим объект **desktop**. Он использует интерфейс по имени **IShellFolder**. Обратите внимание, как мы приходим к Первому Правилу Захвата. Мы распределяем ресурсы в конструкторе, вызывая функцию API **SHGetDesktopFolder**. Интеллектуальный указатель интерфейса будет заботиться об управлении ресурсами (подсчет ссылок).

```
class Desktop: public SIfacePtr<IShellFolder>
{
public:
```



```
 Desktop ()
 {
  if (SHGetDesktopFolder (& _p) != NOERROR)
throw "SHGetDesktopFolder failed";
 }
};
```

Как только мы получили рабочий стол, мы должны создать специальный вид пути, который используется PMDFS. Класс **ShPath** инкапсулирует этот «путь». Он создан из правильного **Unicode** пути (используйте **mbstowcs**, чтобы преобразовать путь ASCII в Unicode: **int mbstowcs(wchar_t *wchar, const char *mbchar, size_t count)**). Результат преобразования — обобщенный путь относительно рабочего стола. Обратите внимание, что память для нового пути распределена оболочкой — мы инкапсулируем это в **SShellPtr**, чтобы быть уверенными в правильном освобождении.

```
class ShPath: public SShellPtr<ITEMIDLIST>
{
public:
 ShPath (SIfacePtr<IShellFolder> & folder, wchar_t * path)
 {
  ULONG lenParsed = 0;
  _hresult =
folder->ParseDisplayName (0, 0, path, & lenParsed, & _p, 0);
 }
 bool IsOK () const { return SUCCEEDED (_hresult); }
private:
 HRESULT _hresult;
};
```

Этот путь оболочки станет корнем, из которого окно просмотра начнет его взаимодействие с пользователем.

С точки зрения клиентского кода, окно просмотра — путь, выбранный пользователем. Именно поэтому он наследуется от **SShellPtr<ITEMIDLIST>**.

Между прочим, **ITEMIDLIST** — официальное имя для этого обобщенного пути.

```
class FolderBrowser: public SShellPtr<ITEMIDLIST>
{
public:
 FolderBrowser (
  HWND hwndOwner,
  SShellPtr<ITEMIDLIST> & root,
```

```
  UINT browseForWhat,
  char const *title);
 char const * GetDisplayName () { return _displayName; }
 char const * GetPath ()  { return _fullPath; }
 bool IsOK() const { return _p != 0; };

private:
 char  _displayName [MAX_PATH];
 char  _fullPath [MAX_PATH];
 BROWSEINFO _browseInfo;
};

FolderBrowser::FolderBrowser (
 HWND hwndOwner,
 SShellPtr<ITEMIDLIST> & root,
 UINT browseForWhat,
 char const *title)
{
 _displayName [0] = '\0';
 _fullPath [0] = '\0';
 _browseInfo.hwndOwner = hwndOwner;
 _browseInfo.pidlRoot = root;
 _browseInfo.pszDisplayName = _displayName;
 _browseInfo.lpszTitle = title;
 _browseInfo.ulFlags = browseForWhat;
 _browseInfo.lpfn = 0;
 _browseInfo.lParam = 0;
 _browseInfo.iImage = 0;
 // Let the user do the browsing
 _p = SHBrowseForFolder (& _browseInfo);

 if (_p != 0)
  SHGetPathFromIDList (_p, _fullPath);
}
```

Вот так! Разве это не просто?

Как функции, не являющиеся методами, улучшают инкапсуляцию

Когда приходится инкапсулировать, то иногда лучше меньше, чем больше.

Если вы пишете функцию, которая может быть выполнена или как метод класса, или быть внешней по отношению к классу, вы должны

предпочесть ее реализацию без использования метода. Такое решение увеличивает инкапсуляцию класса. Когда вы думаете об использовании инкапсуляции, вы должны думать том, чтобы не использовать методы.

При изучении проблемы определения функций, связанных с классом для заданного класса **C** и функции **f**, связанной с **C**, рассмотрим следующий алгоритм:

```
if (f необходимо быть виртуальной)
 сделайте f функцией-членом C;
else if (f – это operator>> или operator<<)
{
 сделайте f функцией – не членом;
 if (f необходим доступ к непубличным членам C)
  сделайте f другом C;
}
else if (в f надо преобразовывать тип его крайнего левого аргу-
мента)
{
 сделайте f функцией – не членом;
 if (f необходимо иметь доступ к непубличным членам C)
  сделайте f другом C;
}
else
 сделайте f функцией-членом C;
```

Этот алгоритм показывает, что функции должны быть методами даже тогда, когда они могли бы быть реализованы как не члены, которые использовали только открытый интерфейс класса C. Другими словами, если **f** могла бы быть реализована как функция-член (метод) или как функция не являющаяся не другом, не членом, действительно ее надо реализовать как метод класса? Это не то, то, что подразумевалось. Поэтому алгоритм был изменен:

```
if (f необходимо быть виртуальной)
 сделайте f функцией-членом C;
else if (f – это operator>> или operator<<)
{
 сделайте f функцией – не членом;
 if (f необходим доступ к непубличным членам C)
  сделайте f другом C;
}
else if (f необходимо преобразовывать тип его крайнего левого ар-
гумента)
{
 сделайте f функцией – не членом;
```

```
 if (f необходимо иметь доступ к непубличным членам C)
  сделайте f другом C;
}
else if (f может быть реализована через доступный интерфейс клас-
са)
 сделайте f функцией – не членом;
else
 сделайте f функцией-членом C;
```

Инкапсуляция не определяет вершину мира. Нет ничего такого, что могло бы возвысить инкапсуляцию. Она полезна только потому, что влияет на другие аспекты нашей программы, о которых мы заботимся. В частности, она обеспечивает гибкость программы и ее устойчивость к ошибкам. Посмотрите на эту структуру, чья реализация не является инкапсулированной:

```
struct Point {
 int x, y;
};
```

Слабостью этой структуры является то, что она не обладает гибкостью при ее изменении. Как только клиенты начнут использовать эту структуру, будет очень тяжело изменить ее. Придется изменять слишком много клиентского кода. Если бы мы позднее решили, что хотели бы вычислять **x** и **y** вместо того, чтобы хранить эти значения, мы были бы обречены на неудачу. У нас возникли бы аналогичные проблемы при запоздалом озарении, что программа должна хранить **x** и **y** в базе данных. Это реальная проблема при недостаточной инкапсуляции: имеется препятствие для будущих изменений реализации. Неинкапсулированное программное обеспечение негибко, и, в результате, оно не очень устойчиво. При изменении внешних условий программное обеспечение неспособно элегантно измениться вместе с ними. Не забывайте, что мы говорим здесь о практической стороне, а не о том, что является потенциально возможным. Понятно, что можно изменить структуру **Point**. Но, если большой объем кода зависит от этой структуры, то такие изменения не являются практичными.

Перейдем к рассмотрению класса с интерфейсом, предлагающим клиентам возможности, подобные тем, которые предоставляет выше описанная структура, но с инкапсулированной реализацией:

```
class Point {
public:
 int getXValue() const;
 int getYValue() const;
 void setXValue(int newXValue);
 void setYValue(int newYValue);
```



```
private:
 ...  // прочее...
};
```

Этот интерфейс поддерживает реализацию, используемую структурой (сохраняющей **x** и **y** как целые), но он также предоставляет альтернативные реализации, основанные, например, на вычислении или просмотре базы данных. Это более гибкий замысел, и гибкость делает возникающее в результате программное обеспечение более устойчивым. Если реализация класса найдена недостаточной, она может быть изменена без изменения клиентского кода. Принятые объявления доступных методов остаются, неизменными, что ведет к неизменности клиентского исходного текста.

Инкапсулированное программное обеспечение более гибко, чем неинкапсулированное, и, при прочих равных условиях, эта гибкость делает его предпочтительнее при выборе метода проектирования.

Степень инкапсуляции

Класс, рассмотренный выше, не полностью инкапсулирует свою реализацию. Если реализация изменяется, то еще имеется код, который может быть изменен. В частности, методы класса могут оказаться нарушенными. По всей видимости, они зависят от особенностей данных класса. Однако ясно видно, что класс более инкапсулирован, чем структура, и хотелось бы иметь способ установить это более формально.

Это легко сделать. Причина, по которой класс является более инкапсулированным, чем структура, заключается в том, что при изменении открытых данных структуры может оказаться разрушенным больше кода, чем при изменением закрытых данных класса. Это ведет к следующему подходу в оценке двух реализаций инкапсуляции: если изменение для одной реализации может привести к большему разрушению кода, чем это разрушение будет при другой реализации, то соответствующее изменение для первой реализации, будет менее инкапсулировано. Это определение совместимо с нашей интуицией, которая подсказывает нам, что вносить изменения следует таким образом, чтобы разрушать как можно меньше кода. Имеется прямая связь между инкапсуляцией (сколько кода могут разрушить вносимые изменения) и практической гибкостью (вероятность, что мы будем делать специфические изменения).

Простой способ измерить, сколько кода может быть разрушено, состоит в том, чтобы считать функции, на которые пришлось бы воздействовать. То есть, если изменение одной реализации ведет потенциально к большему числу разрушаемых функций, чем изменения в другой реа-

лизации, то первая реализация менее инкапсулирована, чем вторая. Если мы применим эти рассуждения к описанной выше структуре, то увидим, что изменение ее элементов может разрушить неопределенно большое количество функций, а именно: каждую функцию, использующую эту структуру. В общем случае мы не можем рассчитать количество таких функций, потому что не имеется никакого способа выявить весь код, который использует специфику структуры. Это особенно видно, если изменения касаются кода библиотек. Однако число функций, которые могли бы быть разрушены, если изменить данные, являющиеся элементами класса, подсчитать просто: это все функции, которые имеют доступ к закрытой части класса. В данном случае, изменятся только четыре функции (не включая объявлений в закрытой части класса). И мы знаем об этом, потому что все они удобно перечислены при определении класса. Так как они — единственные функции, которые имеют доступ к закрытым частям класса, они также — единственные функции, на которые можно воздействовать, если эти части изменяются.

Инкапсуляция и функции — не члены

Приемлемый способом оценки инкапсуляции является количество функций, которые могли бы быть разрушены, если изменяется реализация класса. В этом случае становится ясно, что класс с **n** методами более инкапсулирован, чем класс с **n+1** методами. И это наблюдение поясняет предпочтение в выборе функций, не являющихся ни друзьями, ни методами: если функция **f** может быть выполнена как метод или как функция, не являющаяся другом, то создание ее в виде метода уменьшило бы инкапсуляцию, тогда как создание ее в виде «недруга» инкапсуляцию не уменьшит. Так как функциональность здесь не обсуждается (функциональные возможности **f** доступны классам клиентов независимо от того, где эта **f** размещена), мы естественно предпочитаем более инкапсулированный проект.

Важно, что мы пытаемся выбрать между методами класса и внешними функциями, не являющимися друзьями. Точно так же, как и методы, функции-друзья могут быть подвержены разрушениям при изменении реализации класса. Поэтому, выбор между методами и функциями-друзьями можно правильно сделать только на основе анализа поведения. Кроме того, общее мнение о том, что «функции-друзья нарушают инкапсуляцию» — не совсем истина. Друзья не нарушают инкапсуляцию, они только уменьшают ее точно таким же способом, что и методы класса.

Этот анализ применяется к любому виду методов, включая и статические. Добавление статического метода к классу, когда его функциональные возможности могут быть реализованы как не члены и не друзья

уменьшают инкапсуляцию точно так же, как это делает добавление нестатического метода. Перемещение свободной функции в класс с оформлением ее в виде статического метода, только для того, чтобы показать, что она соприкасается с этим классом, является плохой идеей. Например, если имеется абстрактный класс для виджетов (Widgets) и затем используется функция фабрики классов, чтобы дать возможность клиентам создавать виджеты, можно использовать следующий общий, но худший способ организовать это:

```
// a design less encapsulated than it could be
class Widget {
 ... // внутренее наполнение Widget; может быть:
  // public, private, или protected
public:
 // может быть также «недругом» и не членом
 static Widget* make(/* params */);
};
```

Лучшей идеей является создание вне Widget, что увеличивает совокупную инкапсуляцию системы. Чтобы показать, что Widget и его создание (make) все-таки связаны, используется соответствующее пространство имен (namespace):

```
// более инкапсулированный проект
namespace WidgetStuff {
 class Widget { ... };
 Widget* make( /* params */ );
};
```

Увы, у этой идеи имеется своя слабость, когда используются шаблоны.

Синтаксические проблемы

Возможно что вы, как и многие люди, имеете представление относительно синтаксического смысла утверждения, что не методы и не друзья предпочтительнее методов. Возможно, что вы даже «купились» на аргументы относительно инкапсуляции. Теперь, предположим, что класс **Wombat** поддерживает функциональные возможности *поедания* и *засыпания*. Далее предположим, что функциональные возможности, связанные с поеданием, должны быть выполнены как метод, а засыпание может быть выполнено как член или как не член и не друг. Если вы следуете советам, описанным выше, вы создадите описания подобные этим:

```
class Wombat {
public:
 void eat(double tonsToEat);
```

```
 ...
};
void sleep(Wombat& w, double hoursToSnooze);
```

Это привело бы к синтаксическому противоречию для клиентов класса, потому что для

```
Wombat w;
```

они напишут:

```
w.eat(.564);
```

при вызове **eat**. Но они написали бы:

```
sleep(w, 2.57);
```

для выполнения **sleep**. При использовании только методов класса можно было бы иметь более опрятный код:

```
class Wombat {
public:
 void eat(double tonsToEat);
 void sleep(double hoursToSnooze);
 ...
};
w.eat(.564);
w.sleep(2.57);
```

Ах, эта всеобщая однородность! Но эта однородность вводит в заблуждение, потому что в мире имеется огромное количество функций, которые мечтают о вашей философии.

Чтобы непосредственно ее использовать, нужны функции — не методы. Позвольте нам продолжить пример Wombat. Предположим, что вы пишете программу моделирования этих прожорливых созданий, и воображаете, что одним из методов, в котором вы часто нуждаетесь при использовании вомбатов, является засыпание на полчаса. Ясно, что вы можете засорить ваш код обращениями **w.sleep (.5)**, но этих **(.5)** будет так много, что их будет трудно напечатать. А что произойдет, если это волшебное значение должно будет измениться? Имеется ряд способов решить эту проблему, но возможно самый простой заключается в определении функции, которая инкапсулирует детали того, что вы хотите сделать. Понятно, что если вы не являетесь автором **Wombat**, функция будет обязательно внешней, и вы будете вызвать ее таким образом:

```
// может быть inline, но это не меняет сути
void nap(Wombat& w) { w.sleep(.5); }
Wombat w;
...
```



```
nap(w);
```

И там, где вы используете это, появится синтаксическая несогласованность, которой вы так боитесь. Когда вы хотите кормить ваши желудки (wombats), вы обращаетесь к методу класса, но когда вы хотите их усыпить, вы обращаетесь к внешней функции.

Если вы самокритичны и честны сами с собой, вы увидите, что имеете эту предполагаемую несогласованность со всеми нетривиальными классами, вы используете ее потому, что класс не может имеет любую функцию, которую пожелает какой-то клиент. Каждый клиент добавляет, по крайней мере, несколько своих функций для собственного удобства, и эти функции всегда не являются методами классов. Программисты на C++ используют это, и они не думают ничего об этом. Некоторые вызовы используют синтаксис методов, а некоторые синтаксис внешних вызовов. Клиенты только ищут, какой из синтаксисов является соответствующим для тех функций, которые они хотят вызвать, затем они вызывают их. Жизнь продолжается. Это происходит особенно часто в STL (Стандартной библиотеки C++), где некоторые алгоритмы являются методами (например, **size**), некоторые — не методами (например, **unique**), а некоторые — тем и другим (например, **find**). Никто и глазом не моргает.

Интерфейсы и упаковка

«Интерфейс» класса (подразумевая, функциональные возможности, обеспечиваемые классом) включает также внешние функции, связанные с классом. Им также показано, что правила области видимости имен в C++ поддерживают эти изменения понятия «интерфейса». Это означает, что решение сделать функцию, зависимую от класса, в виде не друга — не члена вместо члена даже не изменяет интерфейс этого класса! Кроме того, вывод функций интерфейса класса за границы определения класса ведет к некоторой замечательной гибкости упаковки, которая была бы иначе недоступна. В частности, это означает, что интерфейс класса может быть разбит на множество заголовочных файлов.

Предположим, что автор класса **Wombat** обнаружил, что клиенты **Wombat** часто нуждаются в ряде дополнительных функций, связанных, с едой, сном и размножением. Такие функции по определению не строго необходимы. Те же самые функциональные возможности могли бы быть получены через вызов других (хотя и более громоздких) методов. В результате, каждая дополнительная функция должна быть не другом и не методом.

Но предположим, что клиенты дополнительных функций, используемых для еды, редко нуждаются в дополнительных функциях для сна или размножения. И предположим, что клиенту, использующему до-

полнительные функции для сна и размножения, также редко нужны дополнительные функции для еды. То же самое можно развить на функции размножения и сна.

Вместо размещения всех Wombat-зависимых функций в одном заголовочном файле, предпочтительнее было бы разместить элементы интерфейса **Wombat** в четырех отдельных заголовках: один для функций ядра **Wombat** (описания функций, связанных с определением класса), и по одному для каждой дополнительной функции, определяющей еду, сон, и размножение. Клиенты включают в свои программы только те заголовки, в которых они нуждаются. Возникающее в результате программное обеспечение не только более ясное, оно также содержит меньшее количество зависимостей, пустых для трансляции. Этот подход, использующий множество заголовков, был принят для стандартной библиотеки (STL). Содержание **namespace std** размещено в 50 различных заголовочных файлах. Клиенты включают заголовки, объявляющие только части библиотеки, необходимые им, и они игнорирует все остальное.

Кроме этого, такой подход расширяем. Когда объявления функций, составляющих интерфейс класса, распределены по многим заголовочным файлам, это становится естественным для клиентов, которые размещают свои наборы дополнительных функций, специфические для приложения, в новых заголовочных файлах, подключаемых дополнительно. Другими словами, чтобы обработать специфические для приложения дополнительные функции, они поступают точно так же, как и авторы класса. Это то, что и должно быть. В конце концов, это только дополнительные функции.

Минимальность и инкапсуляция

Существуют интерфейсы класса, которые являются полными и минимальными. Такие интерфейсы позволяют клиентам класса делать что-либо, что они могли бы предположительно хотеть делать, но классы содержат методов не больше, чем абсолютно необходимо. Добавление функций вне минимума необходимого для того, чтобы клиент мог сделать его работу, уменьшает возможности повторного использования класса. Кроме того, увеличивается время трансляции для программы клиента. Добавление методов сверх этих требований нарушает принцип открытости-закрытости, производит жирные интерфейсы класса, и в конечном счете ведет к загниванию программного обеспечения. Возможно увеличение числа параметров, чтобы уменьшить число методов в классе, но теперь мы имеем дополнительную причину, чтобы отказаться от этого: уменьшается инкапсуляция класса.

Конечно, минимальный интерфейс класса — не обязательно самый лучший интерфейс. Добавление функций сверх необходимости может быть оправданным, если это значительно увеличивает эффективность класса, делает класс более легким в использовании или предотвращает вероятные клиентские ошибки. Основываясь на работе с различными строковыми классами, можно отметить, что для некоторых функций трудно ощутить, когда их делать не членами, даже если они могли быть не друзьями и не методами. «Наилучший» интерфейс для класса может быть найден только после балансировки между многими конкурирующими параметрами, среди которых степень инкапсуляции является лишь одним.

Стандартная мудрость, несмотря на использование «недругов» и не методов улучшает инкапсуляцию класса, и предпочтительнее для таких функций над методами, потому что делает решение проще, когда надо проектировать и разрабатывать классы с интерфейсами, которые являются полными и минимальными (или близкими к минимальному). Возражения, связанные с неестественностью, возникающей в результате изменения синтаксиса вызова, являются совершенно необоснованными, а склонность к «недругам» и не методам ведет к пакующим стратегиям для организации интерфейсов класса, которые минимизируют клиентские зависимости при трансляции, сохраняя доступ к максимальному числу дополнительных функций.

Пришло время отказаться от традиционной, но неточной идеи, связанной с тем, что означает «быть объектно-ориентированным». Действительно ли вы — истинный сторонник инкапсуляции? Если так, то вы ухватитесь за «недружественные» внешние функции с пылом, которого они заслуживают.

Все изложенное выше показывает, что не все так гладко в чистых методах объектно-ориентированного проектирования, если приходится прибегать к ухищрениям, присущим чисто процедурному программированию. Конечно, эффект от использования будет очевиден лишь при разработке достаточно больших программных систем, когда программу приходится развивать и модифицировать, а не создавать заново. Отсюда следует, что чисто объектные языки и методы могут оказаться в этом случае весьма неудобными. А значит: прощай Java и Си? Или их ждет ревизия?

С другой стороны оказывается, что инкапсуляция лучше всего удается процедурным языкам!? Ведь любую структуру данных можно окружить внешними функциями и через них осуществлять доступ. А методы в классе вообще не нужны!?

Глава 19.
Обзор C/C++ компиляторов EMX и Watcom

Watcom C/C++

Watcom — звезда прошлого. Основные черты — многоплатформенность и качество кода. В лучшие времена генерировал код для DOS real mode, DOS protected mode (DOS/4G, DOS/4GW, Phar Lap), Win16, Win32, Win32s, QNX, OS/2 (16- и 32-bit), Netware NLM. Причем, работая под любой системой, можно было генерировать код для всех остальных (к примеру, программу под Win32 можно было скомпилировать и слинковать из-под OS/2 и т.д.). Watcom стал весьма популярен во времена DOS-игр, работающих в защищенном режиме (DOOM и прочие).

К моменту появления версии 11.0 (1997 г.) фирма, разрабатывавшая Watcom, была куплена Sybase Inc., и это, к сожалению, возвестило о кончине компилятора. Дальнейшая разработка была практически заморожена, а в 1999 г. Sybase Inc. объявила о прекращении продаж и установила крайний срок, после которого будет прекращена и техническая поддержка для тех, кто еще успел купить компилятор (это было в середине 2000 г.). Дальнейшая судьба продукта пока неизвестна.

Последняя версия — 11.0B. C++ компилятор в ней не поддерживает **namespaces** и не содержит STL. Впрочем, существуют многие реализации STL, поддерживающие Watcom C++ (к примеру, STLPort).

Под любую поддерживаемую систему есть набор стандартных утилит: компиляторы, линкер, отладчик(и), **make**, **lib**, **strip** и другие. В системах с GUI (OS/2, Windows) есть также IDE (хотя и не очень удобная).

Кодогенерация застыла на уровне 1997 г., и теперь даже MS Visual C++ обгоняет Watcom (естественно, сравнения проводились под Windows, но некоторое представление это дать может).

При работе с Watcom C++ под OS/2 нужно знать следующее:

- В версиях 11.0* в линкере есть досадная ошибка, и вызовы 16-разрядных функций OS/2 (Vio*, Kbd*, Mou* и др.) будут давать трапы. Для борьбы с этим предназначена утилита **LXFix**, которая запускается после линкера и исправляет **fixups**.
- В комплект входит весьма древний OS/2 Toolkit (от OS/2 2.x). Поэтому крайне рекомендуется установить Toolkit из последних (4.0, 4.5).

Кроме разработки «родных» OS/2-программ, Watcom C/C++ можно рекомендовать для компиляции кода, слабо привязанного к ОС. Наличие в стандартной библиотеке функций вроде **_dos_setdrive()**, поддерживаемых под всеми системами (ну, или как минимум под OS/2, Win32 и DOS) позволяет писать в этом смысле платформонезависимо (для пользовательского интерфейса в данном случае можно использовать Turbo Vision).

И напоследок стоит еще раз напомнить про то, что компилятор более не развивается и не поддерживается. Имеющиеся проблемы никуда не денутся и не будут теперь решены.

EMX (GNU C/C++)

EMX — представительство Unix в OS/2 и одно из представительств Unix в DOS. Это целый комплект из компиляторов, сопутствующих утилит и библиотек поддержки. В первую очередь предназначен для портирования программ из среды Unix в OS/2, для чего эмулирует множество функций в «первозданном» виде, включая даже и **fork()**. Основывается на одном из наибольших достижений мира бесплатных программ — системе компиляторов GCC (gcc означает «GNU Compiler Collection»). GCC состоит из собственно трансляторов с языков программирования (в настоящее время это C, C++, Objective C, Fortran 77, Chill и Java, хотя ничто не мешает встроить в систему свой язык), превращающих исходный код в программу на внутреннем языке компилятора (он называется RTL — Register Transfer Language) и стартующих уже от представления на RTL генераторов машинного кода для различных платформ. В частности, поддерживается платформа i386.

Сам EMX является портом GCC под OS/2/DOS и содержит измененные версии компиляторов, линкера, отладчика **gdb** и многих других программ; стандартную библиотеку C, содержащую множество функций из мира Unix; DLL поддержки и многое другое. Кроме того, с помощью EMX под OS/2 были скомпилированы многие другие Unix-программы, к примеру GNU Make, который обязательно понадобится при мало-мальски серьезной разработке.

Кроме всего прочего, EMX позволяет создавать «родные» программы для OS/2, используя OS/2 API. Можно также использовать в программах одновременно и «родные», и «заимствованные» функции.

Программы же, не использующие OS/2 API и некоторых функций Unix, будут «контрабандой» работать и из-под голого DOS во flat mode (в комплекте с EMX поставляется DOS-расширитель). К тому же, и под Windows есть расширитель **rsx.exe**, позволяющий запускать файлы в формате **a.out**, сгенерированные EMX!

Но сам GCC родом из мира Unix, и поэтому EMX также привносит с собой кое-что оттуда. Вот основные моменты:

- Прямой слэш ('/'). Как известно, в Unix для разделения каталогов в файловом пути вместо обратных слэшей используются прямые. Нет, все стандартные функции (**open()**, **fopen()** и др.) понимают оба варианта, но вот при указании файлов и путей компилятору придется использовать прямые. (Не пугайтесь, **c:/aaa/bbb/ccc —** это нормально.)
- Нестандартные форматы файлов. Да, объектные файлы имеют расширение **.o** и формат a**.out**, отличный от привычных **.obj**-файлов. То же самое верно и для файлов объектных библиотек (**.a** в сравнении с **.lib**). И даже исполняемые файлы фактически являются файлами формата **a.out**, содержащими пришпиленный в начале LX-загрузчик.

Но это еще не все. Существует возможность делать и **.obj** файлы, и нормальные **LX .exe** (для этого вызываются всяческие конверторы и на финальном этапе link386). Все эти многочисленные варианты (еще отметим широкие возможности по созданию различных типов DLL) разнятся предоставляемыми возможностями. К примеру, если работать с **.obj** и **LX .exe**, то программа не будет запускаться под DOS и ее нельзя будет отлаживать. Если к тому же выбрать статическую линковку, то еще и список поддерживаемых функций уменьшится. В общем, есть простор для экспериментирования (хотя наиболее часто используемый вариант — **a.out** формат исполняемого файла плюс динамическая линковка с EMX runtime).

- Расширения C, C++. Не будем их здесь перечислять, отметим лишь, что они есть и что их применение делает программу непереносимой.
- Компиляция всегда идет через ассемблер. Т.е. кодогенераторы генерируют лишь ассемблерный текст и переваливают проблему на плечи ассемблера. Не стоит пугаться, с ней он справляется весьма быстро. К тому же, существуют возможности:
- всем частям компилятора общаться друг между другом с помощью pipes (выход препроцессора поступает сразу на вход компилятору, а выход последнего ассемблеру) и обойтись без временных файлов.

- всем частям компилятора висеть некоторое время в памяти после последнего обращения и тем самым экономить время на их запуске.
- Нестандартная библиотека. Работавшие с какими-либо другими компиляторами могут не найти привычных функций, зато могут найти множество других, доселе неизвестных.

Но основное отличие EMX от остальных — это объединение хендлов файлов и сокетов в одну группу. К примеру, используя EMX, не нужно вызывать **sock_init()**, можно использовать **read()** и **write()**, а задачу **soclose()** выполняет обычный **close()**. Кроме этого, функция **select()**, работающая в IBM TCP/IP только для сокетов, в EMX расширена до поддержки любых хендлов, как и полагается в Unix'e.

Как уже отмечалось выше, GCC распространяется под лицензией GNU. Разработка GCC, инициированная где-то в конце 80-х гг. — начале 90-х гг., поначалу велась командой разработчиков, возглавлявшейся идеологом GNU Ричардом М. Столлменом (rms); в 1996 г. ими была выпущена версия 2.7.2.1 и затем экспериментальная версия 2.8.1. Если поддержка C в последней была на уровне ANSI C + расширения, то ситуация с C++ была тяжелой; к тому же, разработка фактически остановилась. Но еще до выпуска 2.8.1 за развитие GCC взялась фирма Cygnus, особенно направив свои усилия на выправление ситуации с C++ (к тому времени до принятия стандарта C++ оставалось не так уж и много). Эта фирма выпустила несколько версий EGCS (Enhanced GNU Compiler Suite), после чего Столлмен и компания решили и вовсе их благословить. Развитие версии 2.8.1, содержавшей кучу ошибок в реализации C++, было заброшено, последняя к тому времени версия EGCS автоматически превратилась в последнюю версию GCC (2.95), а развитие GCC фактически продолжилось командой из Cygnus. Последняя выпущенная ими версия — 2.95.2, это случилось 27 октября 1999 г. (А сама Cygnus не так давно была приобретена небезызвестной компанией Red Hat Inc.)

Последняя версия GCC довольно близка к стандарту, поддерживает все последние добавления к C++ (вроде namespace) и включает в себя также реализацию STL от SGI (она включена в libstdc++, последняя версия 2.90.8). STL из libstdc++ близка к стандарту, но iostreams там все еще не template-based, а взяты из совсем старой libg++. Впрочем, можно опять же обратиться к STLport, она поддерживает и GCC.

Таково состояние GCC на сегодняшний момент. Однако, использовать GCC под OS/2 означает использовать EMX, последняя версия которого (v0.9d) включает в себя старый GCC 2.8.1. Но все не так плохо. Ибо есть еще проект под названием PGCC, суть Pentium-optimized GCC.

Сам GCC хоть и содержит различные оптимизации для базовой платформы, но про особенности конкретных процессоров современности (а это кроме различных вариантов Pentium еще и Cyrix, AMD, все сильно отличающиеся друг от друга по тому, как надо для них оптимизировать) знает крайне мало. Цель проекта PGCC — научить GCC генерировать программы, выжимающие максимум из процессора. (PGCC — это набор «патчей» к GCC). Последний PGCC — 2.95.3, основан на GCC 2.95.2. Оптимизация для конкретного процессора производится при указании определенного ключа в командной строке, так что если его не указывать, то мы получаем «честный» GCC 2.95.2, со всеми его прелестями.

А теперь о прелестях применительно к OS/2. Сам компилятор версии 2.95.2 уже вполне неплох. Он параноидален в духе последнего стандарта (предупреждений об ошибках в сравнении с версией EGCS 1.1.2 стало раза в два больше), не падает, генерирует приемлемый код. Смелые могут даже поставить ключ **-O6** и попробовать оптимизацию под Pentium (здесь имеется в виду PGCC). Но про нормальную отладку PM-приложений можно сразу же забыть. Нацеленный на это PMGDB, входящий в состав EMX, крайне примитивен, да и порой просто не работает. То же самое с profiling (поддержка заявлена, но виснет намертво, до reset). Проблемы могут явиться сами собой. Короче говоря, будьте готовы к возникновению странных проблем и к дубовой отладке.

Компиляторы GCC (C и C++), как уже говорилось выше, можно рекомендовать для переноса программ из Unix под OS/2. Впрочем, как раз в этой области весьма мало вариантов, если не сказать, что как раз один. Можно наоборот, с помощью EMX разрабатывать программы, которые потом будут работать под Unix. Правда, к сожалению, многие функции не поддерживаются EMX. Как минимум, нет очередей сообщений, семафоров, shared memory (ни BSD, ни POSIX). Здесь стоит также заметить, что порты GCC существуют и под win32, и под DOS (а еще вспомним про возможность запуска a.out-программ, сделанных EMX, под DOS и win32!), так что теоретически с помощью EMX можно писать программы, которые будут компилироваться и работать под OS/2, Unix, DOS и Windows.

Главное же достоинство EMX — он абсолютно бесплатен и доступен в исходных текстах. А если вы не верите, что он работает, вот доказательство: такая большая вещь, как XFree86, компилируется с помощью EMX и работает под OS/2! Не говоря о многих других программах меньшего размера.

Глава 20. Использование директивы #import

Как осуществить на VC создание документа и написать туда пару слов?

Возникла следующая проблема — необходимо загрузить документ Excel или Word (вместе с программами — т.е. запускается Word и загружается в него документ) и запустить в нем функцию или макрос на VBA.

Имеется файл БД. Необходимо читать и писать (добавлять и изменять) в файл. Как это лучше сделать?

Как работать с OLE?

Подобные вопросы часто можно встретить в конференциях Fidonet, посвящённых программированию на Visual C++. Как правило, после некоторого обсуждения, фидошная общественность приходит к мнению, что лучшее решение — использование директивы **#import**.

Ниже мы попытаемся объяснить то, как работает эта директива и привести несколько примеров её использования. Надеемся, после этого вы тоже найдёте её полезной.

Директива **#import** введена в Visual C++, начиная с версии 5.0. Её основное назначение облегчить подключение и использование интерфейсов COM, описание которых реализовано в библиотеках типов.

Библиотека типов представляет собой файл или компонент внутри другого файла, который содержит информацию о типе и свойствах COM объектов. Эти объекты представляют собой, как правило, объекты OLE автоматизации. Программисты, которые пишут на Visual Basic, используют такие объекты, зачастую сами того не замечая. Это связано с тем, что поддержка OLE автоматизации является неотъемлемой частью VB и при этом создаётся иллюзия того, что эти объекты также являются частью VB.

Добиться такого же эффекта при работе на C++ невозможно (да и нужно ли?), но можно упростить себе жизнь, используя классы, представляющие обёртки (wrappers) интерфейса **IDispatch**. Таких классов в библиотеках VC имеется несколько.

Первый из них — **COleDispatchDriver**, входит в состав библиотеки MFC. Для него имеется поддержка со стороны MFC ClassWizard'a, диалоговое окно которого содержит кнопку **Add Class** и далее **From a type library**. После выбора библиотеки типов и указания интерфейсов, которые мы хотим использовать, будет сгенерирован набор классов, пред-

ставляющих собой обёртки выбранных нами интерфейсов. К сожалению, **ClassWizard** не генерирует константы, перечисленные в библиотеке типов, игнорирует некоторые интерфейсы, добавляет к именам свойств префиксы **Put** и **Get** и не отслеживает ссылок на другие библиотеки типов.

Второй — **CComDispatchDriver** является частью библиотеки ATL. В VC нет средств, которые могли бы облегчить работу с этим классом, но у него есть одна особенность — с его помощью можно вызывать методы и свойства объекта не только по ID, но и по их именам, то есть использовать позднее связывание в полном объёме.

Третий набор классов — это результат работы директивы **#import**.

Последний способ доступа к объектам OLE Automation является наиболее предпочтительным, так как предоставляет достаточно полный и довольно удобный набор классов.

Рассмотрим пример.

Создадим IDL-файл, описывающий библиотеку типов. Наш пример будет содержать описание одного перечисляемого типа **SamplType** и описание одного объекта **ISamplObject**, который в свою очередь будет содержать одно свойство **Prop** и один метод **Method**.

```
import "oaidl.idl";
import "ocidl.idl";

[
 uuid(37A3AD11-F9CC-11D3-8D3C-0000E8D9FD76),
 version(1.0),
 helpstring("Sampl 1.0 Type Library")
]
library SAMPLLib
{
 importlib("stdole32.tlb");
 importlib("stdole2.tlb");

 typedef enum {
  SamplType1 = 1,
  SamplType2 = 2
 } SamplType;

 [
  object,
  uuid(37A3AD1D-F9CC-11D3-8D3C-0000E8D9FD76),
```



```
  dual,
  helpstring("ISamplObject Interface"),
  pointer_default(unique)
 ]
 interface ISamplObject : IDispatch
 {
  [propget, id(1)] HRESULT Prop([out, retval] SamplType *pVal);
  [propput, id(1)] HRESULT Prop([in] SamplType newVal);
  [id(2)] HRESULT Method([in] VARIANT Var,[in] BSTR Str,[out,
retval] ISamplObject** Obj);
 };

 [
  uuid(37A3AD1E-F9CC-11D3-8D3C-0000E8D9FD76),
  helpstring("SamplObject Class")
 ]
 coclass SamplObject
 {
  [default] interface ISamplObject;
 };
};
```

После подключения соответствующей библиотеки типов с помощью директивы **#import** будут созданы два файла, которые генерируются в выходном каталоге проекта. Это файл **sampl.tlh**, содержащий описание классов, и файл **sampl.tli**, который содержит реализацию членов классов. Эти файлы будут включены в проект автоматически. Ниже приведено содержимое этих файлов.

```
#pragma once
#pragma pack(push, 8)

#include <comdef.h>

namespace SAMPLLib {

// Forward references and typedefs struct __declspec
(uuid("37a3ad1d-f9cc-11d3-8d3c-0000e8d9fd76"))
/* dual interface */ ISamplObject;
struct /* coclass */ SamplObject;

// Smart pointer typedef declarations _COM_SMARTPTR_TYPEDEF
(ISamplObject, __uuidof(ISamplObject));

// Type library items
```

```
enum SamplType
{
 SamplType1 = 1,
 SamplType2 = 2
};

struct __declspec(uuid("37a3ad1d-f9cc-11d3-8d3c-0000e8d9fd76"))
ISamplObject : IDispatch
{
 // Property data
 __declspec(property(get=GetProp,put=PutProp)) enum SamplType
Prop;

 // Wrapper methods for error-handling
 enum SamplType GetProp ( );
 void PutProp (enum SamplType pVal );
 ISamplObjectPtr Method (const _variant_t & Var,_bstr_t Str );

 // Raw methods provided by interface
 virtual HRESULT __stdcall get_Prop (enum SamplType * pVal) = 0 ;
 virtual HRESULT __stdcall put_Prop (enum SamplType pVal) = 0 ;
 virtual HRESULT __stdcall raw_Method (VARIANT Var,BSTR
Str,struct ISamplObject** Obj) = 0 ;
};

struct __declspec(uuid("37a3ad1e-f9cc-11d3-8d3c-0000e8d9fd76"))
SamplObject;

#include "debug\sampl.tli"

} // namespace SAMPLLib

#pragma pack(pop)
-----------------------------------------------------------------
#pragma once

// interface ISamplObject wrapper method implementations

inline enum SamplType ISamplObject::GetProp ( ) {
 enum SamplType _result;
 HRESULT _hr = get_Prop(&_result);
 if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
 return _result;
```



```
}

inline void ISamplObject::PutProp ( enum SamplType pVal ) {
 HRESULT _hr = put_Prop(pVal);
 if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
}
inline ISamplObjectPtr ISamplObject::Method ( const _variant_t &
Var, _bstr_t Str ) {
 struct ISamplObject * _result;
 HRESULT _hr = raw_Method(Var, Str, &_result);
 if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
 return ISamplObjectPtr(_result, false);
}
```

Первое на что следует обратить внимание — это на строчку файла **sampl.tlh**:

```
namespace SAMPLLib {
```

Это означает, что компилятор помещает описание классов в отдельное пространство имён, соответствующее имени библиотеки типов. Это является необходимым при использовании нескольких библиотек типов с одинаковыми именами классов, такими, например, как **IDocument**. При желании, имя пространства имён можно изменить или запретить его генерацию совсем:

```
#import "sampl.dll" rename_namespace("NewNameSAMPLLib")
#import "sampl.dll" no_namespace
```

Теперь рассмотрим объявление метода **Method**:

```
ISamplObjectPtr Method (const _variant_t & Var,_bstr_t Str);
```

Здесь мы видим использование компилятором классов поддержки COM. К таким классам относятся следующие.

- **_com_error**. Этот класс используется для обработки исключительных ситуаций, генерируемых библиотекой типов или каким либо другим классом поддержки (например, класс **_variant_t** будет генерировать это исключение, если не сможет произвести преобразование типов).
- **_com_ptr_t**. Этот класс определяет гибкий указатель для использования с интерфейсами COM и применяется при создании и уничтожении объектов.

- **_variant_t**. Инкапсулирует тип данных **VARIANT** и может значительно упростить код приложения, поскольку работа с данными **VARIANT** напрямую является несколько трудоёмкой.
- **_bstr_t**. Инкапсулирует тип данных **BSTR**. Этот класс обеспечивает встроенную обработку процедур распределения и освобождения ресурсов, а также других операций.

Нам осталось уточнить природу класса **ISamplObjectPtr**. Мы уже говорили о классе **_com_ptr_t**. Он используется для реализации smart-указателей на интерфейсы COM. Мы будем часто использовать этот класс, но не будем делать этого напрямую. Директива **#import** самостоятельно генерирует определение smart-указателей. В нашем примере это сделано следующим образом.

```
// Smart pointer typedef declarations
_COM_SMARTPTR_TYPEDEF(ISamplObject, __uuidof(ISamplObject));
```

Это объявление эквивалентно следующему:

```
typedef _com_ptr_t<ISamplObject, &__uuidof(ISamplObject)>
ISamplObjectPtr
```

Использование smart-указателей позволяет не думать о счётчиках ссылок на объекты COM, т.к. методы **AddRef** и **Release** интерфейса **IUnknown** вызываются автоматически в перегруженных операторах класса **_com_ptr_t**.

Помимо прочих, этот класс имеет следующий перегруженный оператор:

```
Interface* operator->() const throw(_com_error);
```

где **Interface** — тип интерфейса, в нашем случае — это **ISamplObject**. Таким образом, мы сможем обращаться к свойствам и методам нашего COM объекта. Вот как будет выглядеть пример использования директивы **#import** для нашего примера:

```
#import "sampl.dll"

void SamplFunc ()
{
 SAMPLLib::ISamplObjectPtr obj;
 obj.CreateInstance(L"SAMPLLib.SamplObject");
SAMPLLib::ISamplObjectPtr obj2 = obj->Method(1l,L"12345");
 obj->Prop = SAMPLLib::SamplType2;
 obj2->Prop = obj->Prop;
}
```


Как видно из примера создавать объекты COM с использованием классов, сгенерированных директивой **#import**, достаточно просто. Во-первых, необходимо объявить smart-указатель на тип создаваемого объекта. После этого для создания экземпляра нужно вызвать метод **CreateInstance** класса **_com_ptr_t**, как показано в следующих примерах:

```
SAMPLLib::ISamplObjectPtr obj;
 obj.CreateInstance(L"SAMPLLib.SamplObject");
```

или

```
 obj.CreateInstance(__uuidof(SamplObject));
```

Можно упростить этот процесс, передавая идентификатор класса в конструктор указателя:

```
SAMPLLib::ISamplObjectPtr obj(L"SAMPLLib.SamplObject");
```

или

```
SAMPLLib::ISamplObjectPtr obj(__uuidof(SamplObject));
```

Прежде чем перейти к примерам, нам необходимо рассмотреть обработку исключительных ситуаций. Как говорилось ранее, директива **#import** использует для генерации исключительных ситуаций класс **_com_error**. Этот класс инкапсулирует генерируемые значения **HRESULT**, а также поддерживает работу с интерфейсом **IErrorInfo** для получения более подробной информации об ошибке. Внесём соответствующие изменения в наш пример:

```
#import "sampl.dll"

void SamplFunc ()
{
 try {
  using namespace SAMPLLib;
  ISamplObjectPtr obj(L"SAMPLLib.SamplObject");
  ISamplObjectPtr obj2 = obj->Metod(1l,L"12345");
  obj->Prop = SAMPLLib::SamplType2;
  obj2->Prop = obj->Prop;
 } catch (_com_error& er) {
  printf("_com_error:\n"
 "Error   : %08lX\n"
 "ErrorMessage: %s\n"
 "Description : %s\n"
 "Source  : %s\n",
 er.Error(),
 (LPCTSTR)_bstr_t(er.ErrorMessage()),
 (LPCTSTR)_bstr_t(er.Description()),
 (LPCTSTR)_bstr_t(er.Source()));
```

```
 }
}
```

При изучении файла **sampl.tli** хорошо видно как директива **#import** генерирует исключения. Это происходит всегда при выполнении следующего условия:

```
if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
```

Этот способ, безусловно, является универсальным, но могут возникнуть некоторые неудобства. Например, метод **MoveNext** объекта **Recordset ADO** возвращает код, который не является ошибкой, а лишь индицирует о достижении конца набора записей. Тем не менее, мы получим исключение. В подобных случаях придётся использовать либо вложенные операторы **try {} catch**, либо корректировать **wrapper**, внося обработку исключений непосредственно в тело сгенерированных процедур. В последнем случае, правда, придется подключать файлы ***.tlh** уже обычным способом, через **#include**. Но делать это никто не запрещает.

Наконец, настало время рассмотреть несколько практических примеров. Приведем четыре примера работы с MS Word, MS Excel, ADO DB и ActiveX Control. Первые три примера будут обычными консольными программами, в последнем примере покажем, как можно заменить класс **COleDispatchDriver** сгенерированный MFC Class Wizard'ом на классы полученные директивой **#import**.

Для первых двух примеров нам понадобится файл следующего содержания:

```
// Office.h
#define Uses_MSO2000
#ifdef Uses_MSO2000
// for MS Office 2000
#import "C:\Program Files\Microsoft Office\Office\MSO9.DLL"
#import "C:\Program Files\Common Files\Microsoft
Shared\VBA\VBA6\VBE6EXT.OLB"
#import "C:\Program Files\Microsoft Office\Office\MSWORD9.OLB" \
  rename("ExitWindows","_ExitWindows")
#import "C:\Program Files\Microsoft Office\Office\EXCEL9.OLB" \
  rename("DialogBox","_DialogBox") \
  rename("RGB","_RGB") \
  exclude("IFont","IPicture")
#import "C:\Program Files\Common Files\Microsoft
Shared\DAO\DAO360.DLL" \
  rename("EOF","EndOfFile") rename("BOF","BegOfFile")
#import "C:\Program Files\Microsoft Office\Office\MSACC9.OLB"
#else
```



```
// for MS Office 97
#import "C:\Program Files\Microsoft Office\Office\MSO97.DLL"
#import "C:\Program Files\Common Files\Microsoft Shared\VBA\VBE-
EXT1.OLB"
#import "C:\Program Files\Microsoft Office\Office\MSWORD8.OLB" \
  rename("ExitWindows","_ExitWindows")
#import "C:\Program Files\Microsoft Office\Office\EXCEL8.OLB" \
  rename("DialogBox","_DialogBox") \
  rename("RGB","_RGB") \
  exclude("IFont","IPicture")
#import "C:\Program Files\Common Files\Microsoft
Shared\DAO\DAO350.DLL" \
  rename("EOF","EndOfFile")
  rename("BOF","BegOfFile")
#import "C:\Program Files\Microsoft Office\Office\MSACC8.OLB"
#endif
```

Этот файл содержит подключение библиотек типов MS Word, MS Excel и MS Access. По умолчанию подключаются библиотеки для MS Office 2000, если на вашем компьютере установлен MS Office 97, то следует закомментировать строчку:

```
#define Uses_MSO2000
```

Если MS Office установлен в каталог, отличный от «**C:\Program Files\Microsoft Office\Office**», то пути к библиотекам также следует подкорректировать. Обратите внимание на атрибут **rename**, его необходимо использовать, когда возникают конфликты имён свойств и методов библиотеки типов с препроцессором. Например, функция **ExitWindows** объявлена в файле **winuser.h** как макрос:

```
#define ExitWindows(dwReserved,Code)
ExitWindowsEx(EWX_LOGOFF,0xFFFFFFFF)
```

В результате, там, где препроцессор встретит имя **ExitWindows**, он будет пытаться подставлять определение макроса. Этого можно избежать при использовании атрибута **rename**, заменив такое имя на любое другое.

MS Word

```
// console.cpp : Defines the entry point for the console
application.

#include "stdafx.h"
#include <stdio.h>
#include "Office.h"

void main()
```

```
{
 ::CoInitialize(NULL);
 try {
 using namespace Word;
 _ApplicationPtr word(L"Word.Application");
 word->Visible = true;
 word->Activate();

 // создаём новый документ
 _DocumentPtr wdoc1 = word->Documents->Add();

 // пишем пару слов
 RangePtr range = wdoc1->Content;
 range->LanguageID = wdRussian;
 range->InsertAfter("Пара слов");

 // сохраняем как HTML
 wdoc1->SaveAs(&_variant_t("C:\\MyDoc\\test.htm"),
  &_variant_t(long(wdFormatHTML)));
// иногда придется прибегать к явному преобразованию типов,
// т.к. оператор преобразования char* в VARIANT* не определён

 // открывает документ test.doc
 _DocumentPtr wdoc2 = word->Documents->Open
(&_variant_t("C:\\MyDoc\\test.doc"));
 // вызываем макрос
 word->Run("Macro1");

 } catch (_com_error& er) {
 char buf[1024];
 sprintf(buf,"_com_error:\n"
 "Error  : %08lX\n"
 "ErrorMessage: %s\n"
 "Description : %s\n"
 "Source  : %s\n",
 er.Error(),
 (LPCTSTR)_bstr_t(er.ErrorMessage()),
 (LPCTSTR)_bstr_t(er.Description()),
 (LPCTSTR)_bstr_t(er.Source()));

 CharToOem(buf,buf); // только для консольных приложений
 printf(buf);
 }
```



```
 ::CoUninitialize();
}
```

MS Excel

```
// console.cpp : Defines the entry point for the console applica-
tion.

#include "stdafx.h"
#include <stdio.h>
#include "Office.h"

void main()
{
 ::CoInitialize(NULL);
 try {
 using namespace Excel;
 _ApplicationPtr excel("Excel.Application");
 excel->Visible[0] = true;

 // создаём новую книгу
 _WorkbookPtr book = excel->Workbooks->Add();
 // получаем первый лист (в VBA нумерация с единицы)
 _WorksheetPtr sheet = book->Worksheets->Item[1L];
// Аналогичная конструкция на VBA выглядит так:
// book.Worksheets[1]
// В библиотеке типов Item объявляется как метод или
// свойство по умолчанию (id[0]), поэтому в VB его
// можно опускать. На C++ такое, естественно, не пройдёт.
 // заполняем ячейки
 sheet->Range["B2"]->FormulaR1C1 = "Строка 1";
 sheet->Range["C2"]->FormulaR1C1 = 12345L;
 sheet->Range["B3"]->FormulaR1C1 = "Строка 2";
 sheet->Range["C3"]->FormulaR1C1 = 54321L;
 // заполняем и активизируем итоговую строку
 sheet->Range["B4"]->FormulaR1C1 = "Итого:";
 sheet->Range["C4"]->FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)";
 sheet->Range["C4"]->Activate();
 // делаем красиво
 sheet->Range["A4:D4"]->Font->ColorIndex = 27L;
 sheet->Range["A4:D4"]->Interior->ColorIndex = 5L;
// Постфикс L говорит, что константа является числом типа //
long.
```

```
// Вы всегда должны приводить числа к типу long или short // при
преобразовании их к _variant_t, т.к. преобразование // типа int к
_variant_t не реализовано. Это вызвано не // желанием разработчи-
ков компилятора усложнить нам жизнь, // а спецификой самого типа
int.

 } catch (_com_error& er) {
 char buf[1024];
 sprintf(buf,"_com_error:\n"
 "Error   : %08lX\n"
 "ErrorMessage: %s\n"
 "Description : %s\n"
 "Source  : %s\n",
 er.Error(),
 (LPCTSTR)_bstr_t(er.ErrorMessage()),
 (LPCTSTR)_bstr_t(er.Description()),
 (LPCTSTR)_bstr_t(er.Source()));


 CharToOem(buf,buf); // только для консольных приложений
 printf(buf);
 }
 ::CoUninitialize();
}
```

ADO DB

```
// console.cpp : Defines the entry point for the console applica-
tion.
#include "stdafx.h"
#include <stdio.h>

#import "C:\Program Files\Common Files\System\ado\msado20.tlb" \
  rename("EOF","ADOEOF") rename("BOF","ADOBOF")
 // оператор rename необходим, т.к. EOF определён как макрос
 // в файле stdio.h
using namespace ADODB;

void main()
{
 ::CoInitialize(NULL);
 try {
 // открываем соединение с БД
 _ConnectionPtr con("ADODB.Connection");
```



```
con->Open(L"Provider=Microsoft.Jet.OLEDB.3.51;"
L"Data Source=Elections.mdb","","",0);

// открываем таблицу
_RecordsetPtr rset("ADODB.Recordset");
rset->Open(L"ElectTbl",(IDispatch*)con,
adOpenDynamic,adLockOptimistic,adCmdTable);

FieldsPtr flds = rset->Fields;

// добавляем
rset->AddNew();
flds->Item[L"Фамилия"] ->Value = L"Пупкин";
flds->Item[L"Имя"] ->Value = L"Василий";
flds->Item[L"Отчество"] ->Value = L"Карлович";
flds->Item[L"Голосовал ли"] ->Value = false;
flds->Item[L"За кого проголосовал"] ->Value = L"Против всех";
rset->Update();

// подменяем
flds->Item[L"Голосовал ли"] ->Value = true;
flds->Item[L"За кого проголосовал"] ->Value = L"За наших";
rset->Update();
// просмотр
rset->MoveFirst();
while (!rset->ADOEOF) {
 char buf[1024];
 sprintf(buf,"%s %s %s: %s - %s\n",
 (LPCTSTR)_bstr_t(flds->Item[L"Фамилия"]->Value),
 (LPCTSTR)_bstr_t(flds->Item[L"Имя"]->Value),
 (LPCTSTR)_bstr_t(flds->Item[L"Отчество"]->Value),
 (bool)flds->Item[L"Голосовал ли"]->Value? "Да": "Нет",
 (LPCTSTR)_bstr_t(flds->Item[L"За кого проголосовал"] ->Value));

 CharToOem(buf,buf);
 printf(buf);
 rset->MoveNext();
}
} catch (_com_error& er) {
char buf[1024];
sprintf(buf,"_com_error:\n"
"Error   : %08lX\n"
"ErrorMessage: %s\n"
```

```
 "Description : %s\n"
 "Source  : %s\n",
 er.Error(),
 (LPCTSTR)_bstr_t(er.ErrorMessage()),
 (LPCTSTR)_bstr_t(er.Description()),
 (LPCTSTR)_bstr_t(er.Source()));

 CharToOem(buf,buf); // только для консольных приложений
 printf(buf);
 }
 ::CoUninitialize();
}
```

AciveX Control

Для этого примера нам понадобится любое оконное приложение.

ActiveX Control'ы вставляются в диалог обычно через **Components and Controls Gallery**: **Menu ⇨ Project ⇨ Add To Project ⇨ Components and Controls-Registered ActiveX Controls**.

Нам в качестве примера вполне подойдёт **Microsoft FlexGrid Control**. Нажмите кнопку **Insert** для добавления его в проект, в появившемся окне **Confirm Classes** оставьте галочку только возле элемента **CMSFlexGrid** и смело жмите **OK**. В результате будут сформированы два файла **msflexgrid.h** и **msflexgrid.cpp**, большую часть содержимого которых нам придётся удалить. После всех изменений эти файлы будут иметь следующий вид:

msflexgrid.h

```
// msflexgrid.h

#ifndef __MSFLEXGRID_H__
#define __MSFLEXGRID_H__

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#pragma warning(disable:4146)
#import <MSFLXGRD.OCX>

class CMSFlexGrid : public CWnd
{
protected:
```



```
 DECLARE_DYNCREATE(CMSFlexGrid)
public:

 MSFlexGridLib::IMSFlexGridPtr I; // доступ к интерфейсу
 void PreSubclassWindow ();  // инициализация I
};

//{{AFX_INSERT_LOCATION}}

#endif
```

msflexgrid.cpp

```
// msflexgrid.cpp

#include "stdafx.h"
#include "msflexgrid.h"

IMPLEMENT_DYNCREATE(CMSFlexGrid, CWnd)
void CMSFlexGrid::PreSubclassWindow ()
{
 CWnd::PreSubclassWindow();

 MSFlexGridLib::IMSFlexGrid *pInterface = NULL;

 if (SUCCEEDED(GetControlUnknown()->QueryInterface(I.GetIID(),
  (void**)&pInterface))) {
 ASSERT(pInterface != NULL);
 I.Attach(pInterface);
 }
}
```

Теперь вставим элемент в любой диалог, например **CAboutDlg**. В диалог добавим переменную связанную с классом **CMSFlexGrid** и метод **OnInitDialog**, текст которого приведён ниже. При вызове диалога в наш **FlexGrid** будут добавлены два элемента:

```
BOOL CAboutDlg::OnInitDialog()
{
 CDialog::OnInitDialog();

 m_grid.I->AddItem("12345");
 m_grid.I->AddItem("54321");

 return TRUE;
}
```

В заключении, позволим ещё несколько замечаний.

Всегда внимательно изучайте файлы ***.tlh**. Отчасти они могут заменить документацию, а если её нет, то это единственный источник информации (кроме, конечно, OLE/COM Object Viewer).

Избегайте повторяющихся сложных конструкций. Например, можно написать так:

```
book->Worksheets->Item[1L]->Range["B2"]->FormulaR1C1 = "Строка 1";
book->Worksheets->Item[1L]->Range["C2"]->FormulaR1C1 = 12345L;
```

Но в данном случае вы получите неоправданное замедление из-за лишнего межзадачного взаимодействия, а в случае DCOM — сетевого взаимодействия. Лучше написать так:

```
_WorksheetPtr sheet = book->Worksheets->Item[1L];
sheet->Range["B2"]->FormulaR1C1 = "Строка 1";
sheet->Range["C2"]->FormulaR1C1 = 12345;
```

При работе с MS Office максимально используйте возможности VBA для подготовки и тестирования вашего кода.

Будьте внимательны с версиями библиотек типов. К примеру, в MS Word 2000 появилась новая версия метода **Run**. Старая тоже осталась, но она имеет теперь название **RunOld**. Если вы используете MS Word 2000 и вызываете метод **Run**, то забудьте о совместимости с MS Word 97 — метода с таким ID в MS Word 97 просто нет. Используйте вызов **RunOld** и проблем не будет, хотя если очень хочется можно всегда проверить номер версии MS Word.

Бывают глюки. Сразу заметим, что это не связано с самой директивой **#import**. Например, при использовании класса **COleDispatchDriver** с MSADODC.OCX всё прекрасно работало, а после того как стали использовать директиву **#import**, свойство **ConnectionString** отказалось возвращать значение. Дело в том, что директива **#import** генерирует обёртку, используя dual-интерфейс объекта, а класс **COleDispatchDriver** вызывает **ConnectionString** через **IDispatch::Invoke**. Ошибка, видимо, в реализации самого MSADODC.OCX. После изменения кода вызова свойства всё заработало:

```
inline _bstr_t IAdodc::GetConnectionString () {
 BSTR _result;
 HRESULT _hr = _com_dispatch_propget(this,0x01,VT_BSTR,&_result);
// HRESULT _hr = get_ConnectionString(&_result);
 if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
 return _bstr_t(_result, false);
}
```


В результате раскрутки библиотек типов MS Office, компилятор нагенерирует вам в выходной каталог проекта около 12 Mb исходников. Всё это он потом, естественно, будет компилировать. Если вы не являетесь счастливым обладателем PIII, то наверняка заметите некоторые тормоза. В таких случаях надо стараться выносить в отдельный файл всю работу, связанную с подобными библиотеками типов. Кроме того, компилятор может генерировать обёртки классов каждый раз после внесения изменений в файл, в который включена директива **#import**. Представьте, что будет, если после каждого нажатия клавиши будут заново генерироваться все 12 Mb? Лучше вынести объявление директивы **#import** в отдельный файл и подключать его через **#include**.

Удачи в бою.

Глава 21. Создание системных ловушек Windows на Borland C++ Builder

Для начала определим, что именно мы хотим сделать.

Цель: написать программу, которая будет вызывающую хранитель экрана при перемещении курсора мыши в правый верхний угол и выдавать звуковой сигнал через встроенный динамик при переключении языка с клавиатуры.

Предполагается, что такая программа должна иметь небольшой размер. Поэтому будем писать её с использованием только WIN API.

Понятие ловушки

Ловушка (hook) — это механизм, который позволяет производить мониторинг сообщений системы и обрабатывать их до того как они достигнут целевой оконной процедуры.

Для обработки сообщений пишется специальная функция (**Hook Procedure**). Для начала срабатывания ловушки эту функцию следует специальным образом «подключить» к системе.

Если надо отслеживать сообщения всех потоков, а не только текущего, то ловушка должна быть глобальной. В этом случае функция ловушки должна находиться в DLL. Таким образом, задача разбивается на две части:

- Написание DLL с функциями ловушки (их будет две: одна для клавиатуры, другая для мыши).

- Написание приложения, которое установит ловушку.
- Написание DLL.
- Создание пустой библиотеки.

C++ Builder имеет встроенный мастер по созданию DLL. Используем его, чтобы создать пустую библиотеку. Для этого надо выбрать пункт меню **File ⇨ New**: В появившемся окне надо выбрать «**DLL Wizard**» и нажать кнопку «**OK**». В новом диалоге в разделе «**Source Type**» следует оставить значение по умолчанию — «**C++**». Во втором разделе надо снять все флажки. После нажатия кнопки «**Ок**» пустая библиотека будет создана.

Глобальные переменные и функция входа (DllEntryPoint)

Надо определить некоторые глобальные переменные, которые понадобятся в дальнейшем.

```
#define UP 1// Состояния клавиш
#define DOWN 2
#define RESET 3

int iAltKey; // Здесь хранится состояние клавиш
int iCtrlKey;
int iShiftKey;

int KEYBLAY;// Тип переключения языка
bool bSCRSAVEACTIVE;// Установлен ли ScreenSaver
MOUSEHOOKSTRUCT* psMouseHook;// Для анализа сообщений от мыши
```

В функции **DllEntryPoint** надо написать код, подобный нижеприведённому:

```
if(reason==DLL_PROCESS_ATTACH)// Проецируем на адр. простр.
{
HKEY pOpenKey;
 char* cResult=""; // Узнаём как перекл. раскладка
long lSize=2;
KEYBLAY=3;

if(RegOpenKey(HKEY_USERS,".Default\\keyboard layout\\toggle",
&pOpenKey)==ERROR_SUCCESS)
{
 RegQueryValue(pOpenKey,"",cResult,&lSize);
```



```
if(strcmp(cResult,"1")==0)
  KEYBLAY=1;   // Alt+Shift
if(strcmp(cResult,"2")==0)
  KEYBLAY=2;   // Ctrl+Shift

RegCloseKey(pOpenKey);
}
 else
MessageBox(0,"Не могу получить данные о способе"
   "переключения раскладки клавиатуры",
  "Внимание!",MB_ICONERROR);
//------------- Есть ли активный хранитель экрана
if(!SystemParametersInfo(SPI_GETSCREENSAVEACTIVE,0,&bSCRSAVEACTIVE
,0))
MessageBox(0,"Не могу получить данные об установленном"
   "хранителе экрана", "Внимание!",MB_ICONERROR);
 }
return 1;
```

Этот код позволяет узнать способ переключения языка и установить факт наличия активного хранителя экрана. Обратите внимание на то, что этот код выполняется только когда библиотека проецируется на адресное пространство процесса — проверяется условие (**reason==DLL_PROCESS_ATTACH**).

Функция ловушки клавиатуры

Функция ловушки в общем виде имеет следующий синтаксис:

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM
lParam),
```

где:

- **HookProc** — имя функции;
- **nCode** — код ловушки, его конкретные значения определяются типом ловушки;
- **wParam, lParam** — параметры с информацией о сообщении.

В случае нашей задачи функция должна определять состояние клавиш **Alt**, **Ctrl** и **Shift** (нажаты или отпущены). Информация об этом берётся из параметров **wParam** и **lParam**. После определения состояния клавиш надо сравнить его со способом переключения языка (определяется в функции входа). Если текущая комбинация клавиш способна переключить язык, то надо выдать звуковой сигнал.

Всё это реализует примерно такой код:

```
LRESULT CALLBACK KeyboardHook(int nCode,WPARAM wParam,LPARAM
lParam)
{  // Ловушка клав. – биканье при перекл. раскладки
if((lParam>>31)&1)   // Если клавиша нажата...
switch(wParam)
  {// Определяем какая именно
  case VK_SHIFT: {iShiftKey=UP; break};
  case VK_CONTROL: {iCtrlKey=UP; break};
  case VK_MENU: {iAltKey=UP; break};
}
else// Если была отпущена...
switch(wParam)
{// Определяем какая именно
  case VK_SHIFT: {iShiftKey=DOWN; break};
  case VK_CONTROL: {iCtrlKey=DOWN; break};
  case VK_MENU: {iAltKey=DOWN; break};
}
//--------------

switch(KEYBLAY)   // В зависимости от способа переключения рас-
кладки
{
 case 1: // Alt+Shift
{
if(iAltKey==DOWN && iShiftKey==UP)
 {
vfBeep();
  iShiftKey=RESET;
 }
if(iAltKey==UP && iShiftKey==DOWN)
 {
vfBeep();
  iAltKey=RESET;
 }
((iAltKey==UP && iShiftKey==RESET)||(iAltKey==RESET &&
iShiftKey==UP))
 {
  iAltKey=RESET;
  iShiftKey=RESET;
 }
  break;
}
```



```
//-----------------------------------
case 2: // Ctrl+Shift
  {
if(iCtrlKey==DOWN && iShiftKey==UP)
 {
vfBeep();
  iShiftKey=RESET;
 }
if(iCtrlKey==UP && iShiftKey==DOWN)
 {
vfBeep();
  iCtrlKey=RESET;
 }
if((iCtrlKey==UP && iShiftKey==RESET)||(iCtrlKey==RESET &&
iShiftKey==UP))
 {
  iCtrlKey=RESET;
  iShiftKey=RESET;
 }
}
}

return 0;
}
```

Звуковой сигнал выдаётся такой небольшой функцией:

```
void vfBeep()
{// Биканье
MessageBeep(-1);
MessageBeep(-1);// Два раза – для отчётливости
}
```

Функция ловушки мыши

Эта функция отслеживает движение курсора мыши, получает его координаты и сравнивает их с координатами правого верхнего угла экрана (0,0). Если эти координаты совпадают, то вызывается хранитель экрана. Для отслеживания движения анализируется значение параметра **wParam**, а для отслеживания координат значение, находящееся в структуре типа **MOUSEHOOKSTRUCT**, на которую указывает **lParam**. Код, реализующий вышесказанное, примерно такой:

```
LRESULT CALLBACK MouseHook(int nCode,WPARAM wParam,LPARAM lParam)
{  // Ловушка мыши – включает хранитель когда в углу
if(wParam==WM_MOUSEMOVE || wParam==WM_NCMOUSEMOVE)
```

```
{
 psMouseHook=(MOUSEHOOKSTRUCT*)(lParam);
if(psMouseHook->pt.x==0 && psMouseHook->pt.y==0)
if(bSCRSAVEACTIVE)
PostMessage(psMouseHook->hwnd,WM_SYSCOMMAND,
SC_SCREENSAVE,0);
}
return 0;
}
```

Обратите внимание, что команда на активизацию хранителя посылается в окно, получающее сообщения от мыши:

```
PostMessage(psMouseHook->hwnd,WM_SYSCOMMAND, SC_SCREENSAVE ,0).
```

Теперь, когда функции ловушек написаны, надо сделать так, чтобы они были доступны из процессов, подключающих эту библиотеку. Для этого перед функцией входа следует добавить такой код:

```
extern "C" __declspec(dllexport) LRESULT CALLBACK
KeyboardHook(int,WPARAM,LPARAM);
extern "C" __declspec(dllexport) LRESULT CALLBACK
MouseHook(int,WPARAM,LPARAM);
```

Написание приложения, устанавливающего ловушку

Создание пустого приложения

Для создания пустого приложения воспользоваться встроенным мастером. Для этого надо использовать пункт меню **File ⇨ New**: В появившемся окне необходимо выбрать «**Console Wizard**» и нажать кнопку «**Ok**». В новом диалоге в разделе «**Source Type**» следует оставить значение по умолчанию — «**C++**». Во втором разделе надо снять все флажки. По нажатию «**Ок**» приложение создаётся.

Создание главного окна

Следующий этап — это создание главного окна приложения. Сначала надо зарегистрировать класс окна. После этого создать окно. Всё это делает следующий код (описатель окна **MainWnd** определён глобально):

```
BOOL InitApplication(HINSTANCE hinstance,int nCmdShow)
{ // Создание главного окна
WNDCLASS wcx; // Класс окна
wcx.style=NULL;
wcx.lpfnWndProc=MainWndProc;
wcx.cbClsExtra=0;
wcx.cbWndExtra=0;
```



```
wcx.hInstance=hinstance;
wcx.hIcon=LoadIcon(hinstance,"MAINICON");
wcx.hCursor=LoadCursor(NULL,IDC_ARROW);
wcx.hbrBackground=(HBRUSH)(COLOR_APPWORKSPACE);
wcx.lpszMenuName=NULL;
wcx.lpszClassName="HookWndClass";

if(RegisterClass(&wcx)) // Регистрируем класс
{

MainWnd=CreateWindow("HookWndClass","SSHook", /* Создаём окно */
  WS_OVERLAPPEDWINDOW,
CW_USEDEFAULT,CW_USEDEFAULT,
CW_USEDEFAULT,CW_USEDEFAULT,
NULL,NULL,hinstance,NULL);
if(!MainWnd)
return FALSE;

return TRUE;
}
 return false;
}
```

Обратите внимание на то, каким образом был получен значок класса:

```
wcx.hIcon=LoadIcon(hinstance,"MAINICON");
```

Для того, чтобы это получилось надо включить в проект файл ресурсов (***.res**), в котором должен находиться значок с именем «**MAINICON**».

Это окно никогда не появится на экране, поэтому оно имеет размеры и координаты, устанавливаемые по умолчанию. Оконная процедура такого окна необычайно проста:

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT uMsg,WPARAM wParam,
  LPARAM lParam)
{// Оконная процедура
switch (uMsg)
{
case WM_DESTROY:{PostQuitMessage(0); break;}
case MYWM_NOTIFY:
{
if(lParam==WM_RBUTTONUP)
PostQuitMessage(0);
break; // Правый щелчок на значке – завершаем
```

```
}
default:
return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
return 0;
}
```

Размещение значка в системной области

Возникает естественный вопрос: если окно приложения никогда не появится на экране, то каким образом пользователь может управлять им (например, закрыть)? Для индикации работы приложения и для управления его работой поместим значок в системную область панели задач. Делается это следующей функцией:

```
void vfSetTrayIcon(HINSTANCE hInst)
{ // Значок в Tray
char* pszTip="Хранитель экрана и раскладка";
// Это просто Hint
NotIconD.cbSize=sizeof(NOTIFYICONDATA);
NotIconD.hWnd=MainWnd;
NotIconD.uID=IDC_MYICON;
NotIconD.uFlags=NIF_MESSAGE|NIF_ICON|NIF_TIP;
NotIconD.uCallbackMessage=MYWM_NOTIFY;
 NotIconD.hIcon=LoadIcon(hInst,"MAINICON");
lstrcpyn(NotIconD.szTip,pszTip,sizeof(NotIconD.szTip));
Shell_NotifyIcon(NIM_ADD,&NotIconD);
}
```

Для корректной работы функции предварительно нужно определить уникальный номер значка (параметр **NotIconD.uID**) и его сообщение (параметр **NotIconD.uCallbackMessage**). Делаем это в области определения глобальных переменных:

```
#define MYWM_NOTIFY (WM_APP+100)
#define IDC_MYICON 1006
```

Сообщение значка будет обрабатываться в оконной процедуре главного окна (**NotIconD.hWnd=MainWnd**):

```
case MYWM_NOTIFY:
{
if(lParam==WM_RBUTTONUP)
PostQuitMessage(0);
break; // Правый щелчок на значке - завершаем
}
```

Этот код просто завершает работу приложения по щелчку правой кнопкой мыши на значке.

При завершении работы значок надо удалить:

```
void vfResetTrayIcon()
{// Удаляем значок
Shell_NotifyIcon(NIM_DELETE,&NotIconD);
}
```

Установка и снятие ловушек

Для получения доступа в функциям ловушки надо определить указатели на эти функции:

```
LRESULT CALLBACK (__stdcall *pKeybHook)(int,WPARAM,LPARAM);
LRESULT CALLBACK (__stdcall *pMouseHook)(int,WPARAM,LPARAM);
```

После этого спроецируем написанную DLL на адресное пространство процесса:

```
hLib=LoadLibrary("SSHook.dll");
(hLib описан как HINSTANCE hLib)
```

После этого мы должны получить доступ к функциям ловушек:

```
(void*)pKeybHook=GetProcAddress(hLib,"KeyboardHook");
(void*)pMouseHook=GetProcAddress(hLib,"MouseHook");
```

Теперь всё готово к постановке ловушек. Устанавливаются они с помощью функции **SetWindowsHookEx**:

```
hKeybHook=SetWindowsHookEx(WH_KEYBOARD,(HOOKPROC)(pKeybHook),hLib,
0);
hMouseHook=SetWindowsHookEx(WH_MOUSE,(HOOKPROC)(pMouseHook),
hLib,0);
(hKeybHook и hMouseHook описаны как HHOOK hKeybHook; HOOK
hMouseHook;)
```

Первый параметр — тип ловушки (в данном случае первая ловушка для клавиатуры, вторая — для мыши). Второй — адрес процедуры ловушки. Третий — описатель DLL-библиотеки. Последний параметр — идентификатор потока, для которого будет установлена ловушка. Если этот параметр равен нулю (как в нашем случае), то ловушка устанавливается для всех потоков.

После установки ловушек они начинают работать. При завершении работы приложения следует их снять и отключить DLL. Делается это так:

```
UnhookWindowsHookEx(hKeybHook);
UnhookWindowsHookEx(hMouseHook); // Завершаем
FreeLibrary(hLib);
```

Функция WinMain

Последний этап — написание функции WinMain в которой будет создаваться главное окно, устанавливаться значок в системную область панели задач, ставиться и сниматься ловушки. Код её должен быть примерно такой:

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPSTR
lpCmdLine,
 int nCmdShow)
{
MSG msg;
//----------------
hLib=LoadLibrary("SSHook.dll");
if(hLib)
{
(void*)pKeybHook=GetProcAddress(hLib,"KeyboardHook");
hKeybHook=SetWindowsHookEx(WH_KEYBOARD,(HOOKPROC)(pKeybHook),
 hLib,0);// Ставим ловушки
(void*)pMouseHook=GetProcAddress(hLib,"MouseHook");
hMouseHook=SetWindowsHookEx(WH_MOUSE,(HOOKPROC)(pMouseHook),
 hLib,0);
//-------------------------------
if (InitApplication(hInstance,nCmdShow))
// Если создали главное окно
{
vfSetTrayIcon(hInstance);// Установили значок
while (GetMessage(&msg,(HWND)(NULL),0,0))
{// Цикл обработки сообщений
TranslateMessage(&msg);
DispatchMessage(&msg);
}
//--------- Всё - финал
UnhookWindowsHookEx(hKeybHook); // Снимаем ловушки
UnhookWindowsHookEx(hMouseHook);
FreeLibrary(hLib);// Отключаем DLL
vfResetTrayIcon();// Удаляем значок
return 0;
}
}
return 1;
}
```

После написания этой функции можно смело запускать полностью готовое приложение.

Вопросы и ответы

Я наследовал из абстрактного класса A класс B и определил все pure-методы. А она при выполнении ругается, что в конструкторе A по прежнему зовётся абстрактный метод? Почему и что делать?

Так и должно быть — в C++ конструкторы предков вызываются только до конструктора потомка, а вызов методов не инициализированного потомка может окончиться катастрофически (это верно и для деструкторов).

Поэтому и была ограничена виртуальность при прямом или косвенном обращении в конструкторе (деструкторе) предка к виртуальным методам таким образом, что всегда будут вызваны методы предка, даже если они переопределены в потомке.

Замечание: это достижимо подменой **VMT**.

Практически принятое ограничение поначалу сбивает с толку, а проблемы с TV (созданным в Турбо Паскале, где конструктор сам зовёт нужные методы и конструкторы предков) доказывают незавершённость схемы конструкторов C++, в котором из-за автоматического вызова конструкторов подобъектов (предков) сложно описать конструирование объекта как целого в одном месте, поэтому конструкторы C++ правильнее называть **инициализаторами**.

Таким образом, логичнее было бы иметь два шага: автоматический вызов инициализаторов предков (например, с обнулением указателей) и последующий автоматический же вызов общего конструктора. И в C++ это реализуемо!

Для этого во всех классах нужно ввести инициализаторы (защищённые конструктор по умолчанию или конструкторы с фиктивным параметром) и в конструкторах потомка явно задавать именно их (чтобы подавить вызов конструкторов вместо инициализаторов предков). Если же код конструкторов выносить в отдельные (виртуальные) методы, то можно будет вызывать их в конструкторах потомков.

С деструкторами сложнее, поскольку в классе их не может быть более одного, поэтому можно ввести отдельные методы (типа **shutdown** и **destroy** в TV).

Теперь остаётся либо убрать деструкторы (хотя придётся явно вызывать методы деструкции), либо ввести общий признак, запрещающий

деструкторам предков разрушать, и при переопределении метода деструкции переопределять также деструктор. И не забывайте делать их виртуальными!

В качестве примера можно привести преобразование следующего фрагмента, содержащего типичную ошибку, в правильный:

```
class PrintFile
public:
PrintFile(char name[]) Печать(GetFileName(name, MyExt())); virtual
const char *MyExt() return "xxx";
;
class PrintAnotherTypeOfFile :public PrintFile
public:
PrintAnotherTypeOfFile(char name[]) :PrintFile(name) const char
*MyExt() return "yyy";
;
```

После преобразования получаем следующее:

```
class PrintFile
enum Init_ Init; // Тип фиктивного параметра protected:
```

Инициализатор; здесь можно заменить на дефолт конструктор **PrintFile(Init_)**.

Можно добавить несколько «конструкторов» с другими именами, или, если «конструкторы» не виртуальные, можно использовать полиморфизм:

```
bool construct(char name[])
return Печать(GetFileName(name,MyExt()));
public:
//... Код вынесен в отдельный метод для использования в потомках
PrintFile(char name[]) construct(name); virtual const char
*MyExt() return "xxx";
;
class PrintAnotherTypeOfFile :public PrintFile
//... Здесь инициализатор пропущен (никто не наследует)
public:
//... Конструктор; использует "конструктор" предка, с виртуально-
стью;
//... указание инициализатора обязательно
PrintAnotherTypeOfFile(char name[]) :PrintFile(Init)
construct(name);

const char *MyExt() return "yyy";
;
```


Что такое NAN?

Специальное значение вещественного числа, обозначающее не-число — **Non-a-Number**. Имеет характеристику (смещенный порядок) из всех единиц, любой знак и любую мантиссу за исключением **.00__00** (такая мантисса обозначает бесконечность). Имеются даже два типа **не_чисел**:

- **SNAN** — Signalling NAN (сигнализирующие не-числа) — старший бит мантиссы=0
- **QNAN** — Quiet NAN (тихие не-числа) — старший бит мантиссы = 1.

SNAN никогда не формируется **FPU** как результат операции, но может служить аргументом команд, вызывая при этом случай недействительной операции.

QNAN=11__11.100__00 (называется еще «вещественной неопределенностью»), формируется **FPU** при выполнении недействительной операции, делении **0** на **0**, умножении **0** на **бесконечность**, извлечении корня **FSQRT**, вычислении логарифма **FYL2X** отрицательного числа, и т.д. при условии, что обработчик таких особых случаев замаскирован (регистр **CW**, бит **IM=1**). В противном случае вызывается обработчик прерывания (**Int 10h**) и операнды остаются неизменными.

Остальные не-числа могут определяться и использоваться программистом для облегчения отладки (например, обработчик может сохранить для последующего анализа состояние задачи в момент возникновения особого случая).

Как выключить оптимизацию и как longjmp может привести к багу без этого?

Иногда бывает необходимо проверить механизм генерации кода, скорость работы с какой-нибудь переменной или просто использовать переменную в параллельных процедурах (например, обработчиках прерываний). Чтобы компилятор не изничтожал такую переменную и не делал её регистровой придумали ключевое слово **volatile**.

longjmp получает переменная типа **jmp_buf**, в которой **setjmp** сохраняет текущий контекст (все регистры), кроме значения переменных. То есть если между **setjmp** и **longjmp** переменная изменится, её значение восстановлено не будет.

Содержимое переменной типа **jmp_buf** никто никогда (кроме **setjmp**) не модифицирует — компилятор просто не знает про это, потому что все это не языковое средство.

Поэтому при **longjmp** в отличие от прочих регистровые переменные вернутся в исходное состояние (и избежать этого нельзя). Также компилятор обычно не знает, что вызов функции может привести к передаче управления в пределах данной функции. Поэтому в некоторых случаях он может не изменить значение переменной (например, полагая ее выходящей из области использования).

Модификатор **volatile** в данном случае поможет только тем переменным, к к которым он применён, поскольку он никак не влияет на оптимизацию работы с другими переменными...

Как получить адрес члена класса?

Поскольку указатель на член, в отличие от простого указателя, должен хранить также и контекстную информацию, поэтому его тип отличается от прочих указателей и не может быть приведён к **void***. Выглядит же он так:

```
int i; int f();
struct X int i; int f(); x, *px = &x;
int *pi = &i; i = *pi;
int (*pf)() = &f; i = (*pf)();
int X::*pxi       = &X::i; i = x.*pxi;
int (X::*pxf)() = &X::f; i = (px->*pxf)();
```

Зачем нужен for, если он практически идентичен while?

Уточним различие циклов **for** и **while**:

- **for** позволяет иметь локальные переменные с инициализацией;
- **continue** не «обходит стороной» выражение шага, поэтому
  `for(int i = 0; i < 10; i++) ... continue; ...`
  не идентично
  `int i = 0; while(i < 10) ... continue; ... i++`

Зачем нужен NULL?

Формально стандарты утверждают, что **NULL** идентичен **0** и для обоих гарантируется корректное преобразование к типу указателя.

Но разница есть для случая функций с переменным числом аргументов (например, **printf**) — не зная типа параметров компилятор не может преобразовать **0** к типу указателя (а на писюках **NULL** может быть равным **0L**).

С другой стороны, в нынешней редакции стандарта **NULL** не спасёт в случае полиморфности: когда параметр в одной функции **int**, а в другой указатель, при вызове и с **0**, и с **NULL** будет вызвана первая.

Безопасно ли delete NULL? Можно ли применять delete()var после new var()? А что будет при delete data; delete data?

- **delete NULL** (как и **free(NULL)**) по стандарту безопасны;
- **delete[]** после **new**, как и **delete** после **new[]** по стандарту применять нельзя.

Если какие-то реализации допускают это — это их проблемы;

- повторное применение **delete** к освобождённому (или просто не выделенному участку) обладает «неопределённым поведением» и может вызвать всё, что угодно — **core dump**, сообщение об ошибке, форматирование диска и прочее;
- последняя проблема может проявиться следующим образом:

```
new data1; delete data1;
new data2;
delete data1; delete data2;
```

Что за чехарда с конструкторами? Деструкторы явно вызываются чаще...

На это существует неписанное «Правило Большой четвёрки»: если вы сами не озаботитесь о дефолтном конструкторе, конструкторе копирования, операторе присваивания и виртуальном деструкторе, то либо Старший Брат озаботит вас этим по умолчанию (первые три), либо через указатель будет дестроиться некорректно (четвёртый).

Например:

```
struct String1 ... char *ptr; ...
String1 &operator = (String1&); ... ;
struct String2 ... char array[lala]; ... ;
```

В результате отсутствия оператора присваивания в **String2** происходило лишнее копирование **String2::array** в дефолтном операторе присваивания, поскольку **String1::operator =** и так уже дублировал строку **ptr**. Пришлось вставить.

Так же часто забывают про конструктор копирования, который вызывается для передачи по значению и для временных объектов. А ещё есть чехарда с тем, что считать конструктором копирования или операторами присваивания:

```
struct C0  C0 &operator = (C0 &src) puts("C0=");
return *this; ;
struct C1 :C0 C0 & operator = (C0 &src) puts("C1="); return
```

```
*this;
;
int main()
C1 c1, c2; c1 = c2;
```

Некоторые считают, что здесь должен быть вызван дефолтный оператор присваивания **`C1::operator=(C1&)'** (а не **`C1::operator=(C0&)'**), который, собственно, уже вызовет **C0::operator=(C0&)**.

Понадобилось написать метод объекта на ассемблере, а Ватком строит имена так, что это невозможно — стоит знак «:» в середине метки, типа xcvvxx:svvsvv. Какие ключи нужны чтобы он такого не делал?

```
class A;
extern "C" int ClassMetod_Call2Asm (A*, ...);
class A
int Call2Asm(...) return ClassMetod_Call2Asm(this, ...);
;
```

Сожрет любой **Cpp** компилятор. Для методов, которые вы хотите вызывать из **asm** — аналогично...

Так можно произвольно менять генерацию имён в **Watcom:**

```
#pragma aux var "_*";
```

И **var** будет всегда генериться как **var**. А ещё лучше в данном случае использовать **extern «C»** и для переменных также.

После имени функции говорит о том, что Ватком использует передачу параметров в регистрах. От этого помогает **cdecl** перед именем функции.

Пример:

```
extern "C" int cdecl my_func();
```

Скажите почему возникает stack overflow и как с ним бороться?

Причины:

1. Велика вложенность функций

2. Слишком много локальных переменных (или большие локальные массивы);

3. Велика глубина рекурсии (например, по ошибке рекурсия бесконечна)

4. Используется **call-back** от какого-то драйвера (например, мыши);

В пунктах с 1 по 3 — проверить на наличие ошибок, по возможности сделать массивы статическими или динамическими вместо локальных, увеличить стек через **stklen (**для C++**)**, проверять оставшийся стек самостоятельно путем сравнения **stklen** с регистром **SP**.

В пункте 4 — в функции, использующей **call-back**, не проверять стек; в связи с тем, что он может быть очень мал — организовать свой.

Любители Ваткома! А что, у него встроенного ассемблера нет что ли? Конструкции типа asm не проходят?

Встроенного **asm'a** у него на самом деле нет. Есть правда возможность писать **asm**-функции через **'#pragma aux ...'**.

Например:

```
#pragma aux DWordsMover = \
"mov esi, eax",    \
"mov edi, ebx",    \
"jcxz @@skipDwordsMover",  \
"rep movsd",       \
"@@skipDWordsMover:",      \
parm [ebx] [eax] [ecx] modify [esi edi ecx]
void DWordsMover (void* dst, void* src, size_t sz);
```

При создании 16-bit OS/2 executable Watcom требует либу DOSCALLS.LIB. Причем ее нет ни в поставке Ваткома ни в OS/2. Что это за либа и где ее можно достать?

Называют ее теперь по другому. В каталоге **LIB286** и **LIB386** есть такая **OS2286.LIB**. Это то, что вам нужно. Назовите ее **DOSCALLS.LIB** и все.

BC не хочет понимать метки в ассемблерной вставке — компилятор сказал, что не определена эта самая метка. Пришлось определить метку за пределами ASM-блока. Может быть есть более корректное решение?

Загляните в исходники **RTL** от C++ 3.1 и увидите там нечто красивое. Например:

```
#define I asm
//........
I or si,si
I jz m1
I mov dx,1 m1:
I int 21h
```

и т.д.

Есть — компилировать с ключом **'-B' (via Tasm) aka '#pragma inline'**. Правда, при этом могут возникнуть другие проблемы: если присутствуют имена **read** и **_read** (например), то компилятор в них запутается.

Было замечено, что Борланд (3.1, например) иногда генерит разный код в зависимости от ключа **-B**.

Как правило, при его наличии он становится «осторожнее» — начинает понимать, что не он один использует регистры.

Почему при выходе из программы под BC++ 3.1 выскакивает «Null pointer assignment»?

Это вы попытались что-то записать по нулевому адресу памяти, чего делать нельзя.

Типичные причины:

- используете указатель, не инициализировав его.

Например:

```
char *string; gets(string);
```

- запрашиваете указатель у функции, она вам возвращает **NULL** в качестве ошибки, а вы этого не проверяете.

Например:

```
FILE *f = fopen("gluck", "w"); putc('X', f);
```

Это сообщение выдаётся только в моделях памяти **Tiny**, **Small**, **Medium**.

Механизм его возникновения такой: в сегменте данных по нулевому адресу записан борландовский копирайт и его контрольная сумма. После выхода из **main** контрольная сумма проверяется и если не совпала — значит напорчено по нулевому адресу (или рядом) и выдаётся сообщение.

Как отловить смотрите в **HELPME!.DOC** — при отладке в **Watch** поставить выражения:

```
*(char*)0,4m
(char*)4
```

потом трассировать программу и ловить момент, когда значения изменятся.

Первое выражение — контрольная сумма, второе — проверяемая строка.

При запуске программы из BC (Ctrl-F9) все работает нормально, а если закрыть BC, то программа не запускается. Что делать?

Если вы используете **BWCC**, то эту либу надо грузить самому — просто среда загружает **BWCC** сама и делаёт её доступной для программы. Советуем вам пользоваться таким макросом:

```
#define _BEST EnableBWCC(TRUE); \
EnableCtl3d(TRUE); \
EnableCtl3dAutosubclass(TRUE)
```

Потом в **InitMainWindow** пишете:

```
_BEST;
```

и все будет хорошо.

Вообще-то правильнее **OWL**'евые экзепшены ловить и выдавать сообщение самостоятельно. Заодно и понятнее будет отчего оно произошло:

```
int OwlMain(int /*argc*/, char* /*argv*/[])
int res;
TRY res = App().Run();
CATCH((xmsg &s)
//Какие хочешь ексепшены
MessageBox(NULL, "Message", s.c_str());
return res;
```

Почему иногда пытаешься проинспектировать переменную в BC++ во время отладки, а он ругается на inactive scope?

Вот пример отлаживаемой программы. Компилим так:

```
bcc -v is.cpp
=== Cut ===
#include <iostream.h>
void a()
int b = 7;
cout << b << endl;
void main()
a();
=== Cut ===
```

Входим в **TD**. Нажимаем **F8**, оказываемся на строке с вызовом **a()**. Пытаемся **inspect b**. Естественно, не находим ничего. А теперь перемещаем курсор в окне исходника на строку с **cout**, но трассировкой в **a()** не входим и пробуем посмотреть **b**. И вот тут-то и получаем **inactive scope**.

Были у меня две структуры подобные, но вторая длиннее. Сначала в функции одна была, я на ней отлаживался, а потом поменял на вторую, да только в malloc'е, где sizeof(struct ...) старое оставил, и налезали у меня данные на следующий кусок хипа

Для избегания подобной баги можно в **C** сымитировать Сиплюсный **new**:

```
#define tmalloc(type)      ((type*)malloc(sizeof(type)))
#define amalloc(type, size) ((type*)malloc(sizeof(type) * (size)))
```

Более того, в последнем **define** можно поставить **(size) + 1**, чтобы гарантированно избежать проблем с завершающим нулём в строках.

Можно сделать иначе. Поскольку присвоение от **malloc()** как правило делают на стилизованную переменную, то нужно прямо так и писать:

```
body = malloc(sizeof(*body));
```

Теперь вы спокойно можете менять типы не заботясь о **malloc()**. Но это верно для Си, который не ругается на присвоение **void*** к **type*** (иначе пришлось бы кастить поинтер, и компилятор изменения типа просто не пережил бы).

Вообще в C нет смысла ставить преобразования от **void*** к указательному типу явно. Более того, этот код не переносим на C++ — в проекте стандарта C++ нет **malloc()** и **free()**, а в некоторых компиляторах их нет даже в **hosted c++** заголовках.

Проще будет:

```
#ifdef __cplusplus
# define tmalloc(type) (new type)
# define amalloc(type, size)
(new type[size])
#else
# define tmalloc(type) malloc(sizeof(type))
# define amalloc(type, size) malloc(sizeof(type) * (size))
#endif
```

Суммируя вышеперечисленное, можно отметить следующее. Необходимо скомбинировать все варианты:

```
#ifdef __cplusplus
# define tmalloc(type) (new type)
# define amalloc(type, size)
(new type[size])
# define del(var) delete(var)
#else
# define tmalloc(type) ((type*)malloc(sizeof(type)))
```



```
# define amalloc(type, size) ((type*)malloc(sizeof(type) *
(size)))
# define del(var) free(var)
# define vmalloc(var)
((var) = malloc(sizeof(*(var))))
#endif
```

Я не понимаю, почему выдаются все файлы, вроде указал, что мне нужны только с атрибутом директория? Можно, конечно, проверять ff_attrib, что нашли findfirst и findnext, но это мне кажется не выход. Может я что не дочитал или не понял?

```
done = findfirst("*.*", &onlydir, FA_DIREC);
while(!done)
  cout << onlydir.ff_name << endl;
  done = findnext(&onlydir);
```

Это не баг, это фича MS DOS. Если атрибут установлен, то находятся как файлы с установленным атрибутом, так и без него. Если не установлен, то находятся только файлы без него. И проверять **ff_attrib** вполне выход. Вы не дочитал хелп про **findfirst/findnext**.

Создается файл: fopen(FPtr, "w"). Как может случиться, что структура пишется на диск некорректно?

```
fopen (FPtr,"wb");
```

Режим не тот...

При печати функцией cprintf в позицию экрана x = 80, y = 25 происходит автоматический перевод строки (сдвиг всего экрана на строку вверх и очистка нижней строки) и это знакоместо так и остается пустым. Может кто знает, как вывести символ в это знакоместо?

Нажмите **Ctrl+F1** на слове **_wscroll** в Борландовском **IDE**. Правда, **printf** это не вылечит, так как его вывод идёт не через борландовскую библиотеку.

Как очистить текстовый экран в стандарте ANSI C?

Никак, в ANSI C нет понятия экрана и текстового режима. В Turbo C так:

```
#include <conio.h>
void main(void) clrscr();
```

Можно также попробовать выдавать ANSI **ESC**-коды или сделать следующее:

```
#include <stdio.h>
#define NROWS 2*25 /*
```

Чтобы обработать случай курсора в первой строке:

```
void main(void)
short i;
for(i = 0; i < NROWS; i++) puts("");
```

Но это совершенно негарантированные способы.

Используя прерывания VESA, пытаюсь подключить мышь и вот тут начинается сумасшедший дом... Что делать?

Мышиный драйвер не знает какой у вас на данный момент видеорежим и использует параметры предыдущего режима (у вас он наверное текстовый — там мышь скачет дискретно по 8).

Поэтому, рисовать мышь вы должны сами. А чтобы координаты мыши отслеживать, у **33h** прерывания есть функция, которая возвращает смещение мыши от последней ее позиции.

Можно обойтись без рисования своего курсора мыши если найти драйвер, понимающий **VESA**-режимы.

Например, в **Logitech MouseWare 6.3** входит некий оверлейчик для генерации курсора для режимов Везы, который соответствует какой-то там совместной спецификации Везы и Логитеча.

Как установить патчи на версию «Try & Bye»?

Для Win32 в реестре меняете ключ:

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\IBM VisualAge for C++ for Windows
Demo\demo
```

на

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\IBM VisualAge for C++ for
Windows\3.5
```

Для OS/2 редактируете файл **\os2\system\epfis.ini** при помощи любого редактора **INI** файлов и заменяете в нем:

- имя апликации

```
EPFINST_IBM VisualAge C++ for OS/2_TRIAL_COPY_0001
```

или что-то подобное на

```
EPFINST_IBM VisualAge C++ for OS/2_5622-679_0001
```

- содержимое ключа **ApplicationName** для данной апликации изменяете с

```
IBM VisualAge C++ for OS/2 TRIAL COPY
```

или опять что-то подобное на

```
IBM VisualAge C++ for OS/2
```


- файл **cppexit.dll** копируете в **exit.dll**.

После таких манипуляций можно спокойно ставить патчи.

Как сортировать записи в IVBContainerControl?

IVBContainerControl отвечает только за отображение. Капать надо в области **IVSequence**, на который есть ссылка в объекте **IVBContainerControl**.

Он ведь только то отображает, что в **IVSequence * IVBContainerControl::items** содержится. Так что берете этот **items** и сортируете.

Для создания невизуальных part лучше использовать VB или .VBE?

Настоятельно рекомендуется **.VBE**

Где находятся описания типов (не классов) для VB?

.VBE, использовать редактор **Part** для описания типов нельзя. Правильнее всего посмотреть **.\Samples\VisBuild\vbSample*.VBE** Там хорошо показано, как делать описание блоков функций, типов и перечислений.

Что можно использовать для выбора цвета?

Для выбора цвета лучше всего использовать **..\Sample\VisBuild\Doodle\ClrDlg.VBB.**

Можно ли использовать VAC++ без WPS и WF?

Можно. Надо инсталлировать его из под **WPS**, а потом заменить его на что-нибудь типа **FileBar**.

Будет работать все, кроме редактора. Это позволяет использовать **VB** на **16 MB**.

Есть некое окошко, которое должно делать нечто через каждые N секунд. Как это правильно изобразить в VisualBilder/PartEditor?

На **Ibm**'ком сервере в примерах по **VAC++** лежит как раз подобный пример. Файл **vbtimer.zip** размером ~30 K.

Я уже замучился загружать все .vbb модули в Visual Builder. Что делать?

Создайте файлик **VbLoad.Dat** со списком этих файлов с указанием пути и положите его либо в каталог, где живут файлы приложения, в случае если **Visual Builder** запускается оттуда, либо (что подходит только для одного проекта) в каталог в **VbBase.Vbb**, **VbDax.Vbb e.t.c** (он называется **IVB** для **Win** и **DDe4Vb** для **Os/2**).

Пути указывать не обязательно, если каталог, где они лежат «входит» в переменную окружения **VBPATH**.

Где взять документацию на Ватком?

В поставке. Все что есть в виде книжек включено в дистрибутив, кроме книги Страуструпа.

Как поставить Ватком версии 10 под пополамом, при установке в самом конце происходят странные вещи?

Лучше всего провести установку (копирование файлов и создание каталогов) в досовской сессии, а потом пополамным инсталлером просто откорректировать конфиги и создать все необходимые установки.

В русифицированной WIN9x криво, устанавливается WATCOM. Не создает папки со своими иконками. Что делать?

Нужно сделать каталоги **\Windows\Start Menu\Programs** и переустановить Ватком. Потом перекинуть **.lnk** куда вам нужно.

При отсутствии нужных англоязычных папок ссылки улетают в никуда.

Он занял очень много места на диске, от чего можно избавиться?

Если вы не предполагаете писать программы под какие-либо платформы, то не стоит устанавливать и библиотеки для них, если вы собираетесь работать под пополамом, можно смело прибить досовские и виндовозные хелпы, и программку для их просмотра.

Кроме того, надо решить какой средой вы будете пользоваться, компилировать в дос-боксе или нет.

Пополамный компилятор ресурсов под досом очень слаб и сваливается по нехватке памяти даже на простых файлах. Более того есть мнение, что при компиляции в осевой сессии, по крайней мере линкер работает примерно в 3 раза быстрее.

Вот я его поставил, ничего не понятно, с чего начать?

Прежде всего — почитать документацию, версия 10 поставляется с огромными файлами хелпа, если вы работаете под пополамом — используйте **VIEW** или иконки помощи в фолдере, если под **Windows** — соответственно программку **WHELP** для просмотра ***.HLP**, ну и под досом — аналогично, правда там вы не получите красивых окошек и приятной гипертекстовой среды.

Где у него IDE, я привык, чтобы нажал кнопку, а оно откомпилировалось?

IDE существует, но работает только под **Windows** или **OS/2**. Для работы в Досе используйте командную строку.

Если вы так привыкли к **IDE** — поддержка Ватком есть в **MultiEdit**, и комплект удобных макросов тоже.

С чего начать, чтобы сразу заработало?

Начните с простейшего:

```
#include <stdio.h>
main()
puts("Hello, world");
```

Для компиляции нужно использовать:

```
wcl hello.c – для DOS/16
wcl386 /l=dos4gw hello.c – для DOS4GW
```

Я написал простейшую программку, а она внезапно повисает, или генерирует сообщение о переполнении стека, что делать? В то же время когда я компилирую эту программку другим компилятором — все работает нормально?

Желательно сразу после установки поправить файлики **WLSYSTEM.LNK**, поставив требуемый размер стека, по умолчанию там стоит 1 или 2 Кб, чего явно недостаточно для программ, создающих пусть даже небольшие объекты на стеке.

Для большинства применений достаточно размера стека в 16 или 32 килобайта. Если вы работаете под экстендером, можно поставить туда хоть мегабайт.

Я столкнулся с тем, что Ватком ставит знак подчеркивания не в начало имени, а в конец, к чему бы это?

Положение знака подчеркивания говорит о способе передачи параметров в данную функцию, если его нет совсем, параметры передаются через регистры, если сзади — через стек.

Я написал подпрограмму на ассемблере, со знаком подчеркивания спереди, а Ватком ищет то же имя, но со знаком «_» сзади, как это поправить?

Можно написать:

```
#pragma aux ASMFUNC "_*";
```

и описывать все свои функции как:

```
#pragma aux (ASMFUNC) foo;
```

```
#pragma aux (ASMFUNC) bar;
```

Причем, есть специальное значение — символ «^», который сигнализирует, что имя надо преобразовать в верхний регистр, например: **#pragma aux myfunc «^»;** приведет к появлению в объектном файле ссылки на **«MYFUNC»**.

Есть библиотека, исходники которой отсутствуют, как заставить Ватком правильно понимать имена функций и ставить знак «_» спереди а не сзади?

Нужно в файле заголовка описать данные функции как **cdecl**, при этом параметры будут передаваться через стек и имя функции будет сформировано правильно.

Как сделать так, чтобы в некоторых случаях Watcom передавал параметры не через регистры, а через стек?

Использовать **cdecl**.

Например:

```
extern void cdecl dummy( int );
```

Как делать ассемблерные вставки в программу?

Примерно так:

```
unsigned short swap_bytes ( unsigned short word );
#pragma aux swap_bytes = "xchg ah, al" \
parm [ ax ] \
value [ ax ];
```

Слово **parm** определяет в каком регистре вы передаете значение, слово **value** — в каком возвращаете.

Можно использовать метки. Есть слово **modify** — можно указать что ваша вставка (или функция) не использует память, а трогает только те или иные регистры.

От этого оптимизатору лучше жить. Прототип не обязателен, но если есть, то компилер проверяет типы.

Надо слепить задачу под графику, но нужны окошки и мышь. Тащить ли ZINC 3.5, или в графике описать что-нибудь свое. Может, под Ватком что-то есть более мощное и готовое?

Ничего лучше **Зинки** пока нет. Тащите лучше **Зинку 4.0**, она вроде под Ватком лучше заточена.

При написании некоторых функций по видео-режимах вдруг захотелось мне сотворить динамические библиотеки. Есть мысля генерить exe-файл а затем грузить его. Что делать?

Использовать **DOS4GW/PRO**. Он вроде поддерживает **DLL**. Или пользоваться **PharLap TNT**, он тоже поддерживает.

Грузить экзешник тоже можно, но муторно. Через **DPMI** аллоцируете сегмент (сегменты) делаете из них код и данные, читаете экзешник и засовываете код и данные из него в эти сегменты. Лучше использовать **TNT**.

Графическая библиотека Ваткома отказывается переключать режимы/банки или делает это криво. Что делать?

В результате ковыряния в библиотеке выяснилось, что криворукие ваткомовцы совершенно не задумываются ни о какой переносимости и универсальности их библиотек.

В результате, если видео-карта имеет в биосе прошитое имя производителя или другую информацию о нем, то для нее будет вызываться вместо функции переключения банков через **VESA**, другая функция, работающая с картой напрямую (иногда даже через порты).

Единственная проблема, что у каждого производителя рано или поздно выходят новые и продвинутые карты, раскладка портов в которых может отличаться от той, которая использовалась в старых моделях.

В результате, все это свинство начинает глючить и иногда даже виснуть.

После того, как вы руками заткнете ему возможность использовать «родные» фишки для конкретной карты и пропишите пользоваться только **VESA** — все будет работать как из пушки.

Как затыкать — а просто, есть переменная: **_SVGAType**, которая описывается следующим образом:

```
"extern "C" int _SVGAType;",
```

и потом перед (важно!) вызовом **_setvideomode** нужно сказать:

```
"_SVGAType = 1;"
```

Как руками слинковать exe-файл?

Командой **WLINK**, указав параметры.

```
name ...
system ...
debug all
option ...
```

```
option ...
option ...
...
file ...
file ...
...
libpath ...
library ...
```

Например:

```
wlink name myprog system dos4gw debug all file myprog
```

Что такое ms2vlink и зачем она нужна?

Это для тех кто переходит с мелкософтовского **Си.** Преобразователь команд **LINK** в **WLINK**.

Что такое _wd_?

Это отладчик, бывший **WVIDEO**, но с более удобным интерфейсом.

Поставляется начиная с версии 10.

Нужно состряпать маленький NLM'чик. Что делать?

Вам нужен **WATCOM 10.0**. В него входит **NLM SDK** и вроде хелп к нему. Если **WC <= 9.5,** то нужен сам **NLM SDK** и документация.

```
// Линковать :
// (файл wclink.lnk например)
// system netware
// Debug all
// opt scr 'Hello, world'
// OPT VERSION=1.0
// OPT COPYR 'Copyright (C) by me, 1994'
#include <conio.h>
void main( void )
cprintf( "Hello, world!\n\r" );
ConsolePrintf( "Hello, World - just started!\n\r" );
RingTheBell();
```

Собираю программу под OS/2 16-бит, линкер не находит библиотеку DOSCALLS.LIB. Кто виноват и что делать?

Никто не виноват. В поставке Ваткома есть библиотека **os2286.lib**.

Это она и есть. Ее надо либо переименовать в **doscalls.lib**, либо явно прилинковывать.

Что такое удаленная отладка через pipe? Как ею пользоваться под OS/2?

В одной сессии запускается **vdmserv.exe**, потом запускается отладчик **wd /tr=vdm** и соединяется с **vdmserv** по пайпу, ну и рулит им. Как удаленная отладка через компорт работает знаете? Вот тут так же, только через пайп.

Собираю 32-битный экзешник под PM с отладочной информацией (/d2), но после того как осевым rc пришпиливаю к нему ресурсы, отладочной информации — как не бывало. Это лечится как-нибудь?

Откусываете дебугинфу **wstrip**'ом в **.sym** файл и потом присобачиваете ресурсы. Если имя экзешника и имя **.sym** совпадают, дебаггер сам его подхватит.

Отладочную информацию надо сбрасывать в **SYM**-файл:

```
wcl386 /d2 /"op symf" /l=os2v2_pm
```

WATCOM на 4 мб компилирует быстрей чем на 8 мб, а на 8 мб быстрее чем на 16 мб, почему?

Чем больше памяти, тем лучше работает оптимизатор. Можно дать ему фиксированный размер памяти — **SET WCGMEMORY=4096**, и тогда он не будет пользоваться лишней памятью.

Учтите, что для компиляции программ для Windows на C++ данного значения может не хватить.

Есть такая штука — pipe в gcc и bcc. А вот в Watcom'е как перехватить выхлоп программы?

В смысле забрать себе **stdout** и **stderr**? Да как обычно — сдупить их куда-нибудь. Функцию **dup()** еще никто не отменял.

А есть ли способ перехватить ошибку по нехватке памяти? То есть какой-нибудь callback, вызываемый диспетчером памяти при невозможности удовлетворить запрос?

В C++ есть стандартный: **set_new_handler()**.

Чем отличаются статические DLL от динамических?

Разница в том, что вы можете функи из **DLL** на этапе линковки в **EXE**'шник собрать (**static**). А можете по ходу работы проги **DLL** грузить и функи выполнять (**dynamic**).

Решил тут DLL под OS/2 создать — ничего не вышло. Что делать?

Вы динамически собираетесь линковать или статически? Если статически, тогда вам просто **declare func** сделать и включить **dll** в **test.lnk**.

Если динамически, то вы должны прогрузить **dll**, получить адрес функи и только после этого юзать. Можно делать это через **API OS/2: DosLoadModule DosQueryProcAddr DosFreeModule.**

Например:

```
exe.c:
#define INCL_DOSMODULEMGR
#include <os2.h>
#define DLLNAME "DLL"
PFN Dllfunc;
char FuncName[]="RegardFromDll_";
char LoadError[100];
void main()
HMODULE MHandle;
DosLoadModule( LoadError, sizeof( LoadError ), DLLNAME, &MHandle
); DosQueryProcAddr( MHandle, 0, FuncName, &Dllfunc );
(*Dllfunc)();
DosFreeModule( MHandle );
dll.c
#include <stdio.h>
void RegardFromDll( void )
printf( "This printed by function, loaded vs DLL\n" );
```

Когда компиляете свою **DLL**, то добавьте свич **-bd**, который создаст в **.obj** такое дело, как **DLLstart**. После этого все заработает:

```
wpp386 -bd -4s -ox dll.cpp
```

Как подавить варнинги о неиспользованных аргументах?

Если используете плюсовый компилер — просто опускайте имена параметров, например:

```
void foo( int bar, char* )
```

или

```
#pragma off(unreferenced)
```

Можно использовать макрос:

```
#ifndef NU
#ifdef __cplusplus
#ifdef __BORLANDC__
#define NU( ARG ) ARG
#else
```



```
#define NU( ARG ) (void)ARG
#endif
#else
#define NU( ARG ) ARG=ARG
#endif
#endif
```

Для многих, особенно юниксового происхождения, компайлеров работает **/*ARGSUSED*/** перед определением функции.

Подскажите, как в watcom'е увеличить число открытых файлов?

Смотрите **TFM**. **_grow_handles(int newcount)**.

Как заставить 16-ти битные OS/2 задачи видеть длинные имена файлов?

Опция **newfiles** для линкера.

Что-то у меня Dev.Toolkit for OS/2 Warp к Ваткому WC10.0 прикрутить не получается. Говорит definition of macro '_Far16' not identical previos definition. Что делать?

Воткните где-нибудь определение **IBMCPP** или **-D_IBMCPP** в командной строке или **#define** перед **#include <os2.h>**.

Для Ваткома 10.5а надо не просто **d_IBMCPP**, а **-d_IBMCPP_=1.**

Пишу: printf («*»);, а он сразу ничего не печатает. Что делать?

В стандарте **ansi**, чёткого определения как должны буферизоваться потоки **stdin/stdout/stderr** нет, нормальным является поведение со строчной буферизацией **stdout/stdin**.

Всякие другие дос-компилеры обычно не буферизуют **stdout** совсем, что тоже нормально. Признаком конца строки в потоке является **'\n'**, именно при получении этого символа происходит **flush** для **line buffered** потока.

Выходов два: отменить буферизацию или писать **'\n'** в нужных местах. Можно **fflush(stdout)** звать, тоже вариант.

Буферизация отменяется **setbuf(stdout,NULL)** или **setvbuf(stdout, NULL,_IONBF,0)**.

Можно ли сделать встроенный в exe-шник DOS4GW, как в D00M?

Легально — нет. Предыдущие версии позволяли просто скопировать:

```
copy /b dos4gw.exe + a.exe bound.exe
```

Но сейчас (начиная с версии 10.0a) это не работает и для этой цели нужно приобрести **dos4gw/pro** у фирмы **Tenberry Software**.

Нелегально — да. Существует утилита **dos4g/link** для автоматизированного выдирания и вклеивания экстендеров из/в **EXE**-файлов. Помещалась в **WATCOM.C** в **uuencode** и доступна от автора.

Нужно взять не тот **DOS4GW**, что в комплекте (**DOS4GW 1.97**), а **Pro**-версию (**DOS4GW Professional**). Выдрать можно из **DOOM**, **HERETIC**, **HEXEN**, **WARCRAFT2** и т.д., где он прибинден. Причем можно найти 2 разновидности **Pro 1.97** — одна поддерживает виртуальную память, другая нет и еще что-то по мелочи.

Различаются размерами (который с виртуалкой — толще). Прибиндить можно разными тулзами, например **PMWBIND** из комплекта **PMODE/W**.

Также можно отрезать у **dos4gw.exe** последние несколько байт с хвоста, содержащие строку **WATCOM patch level [...]**. Далее обычным бинарным копированием:

```
copy /b dos4gw.exe myexe.exe mynewexe.exe
```

Работоспособно вплоть до версии **DOS/4GW 1.95**. В версии **1.97** введена проверка на внедренность **linexe** в хвост экстендера.

Еще существует родной биндер для **DOS/4GW**. Он в какой-то мере может помочь **pmwbind.exe** от **PMODE/W** (однако версия **1.16** не понимала каскадный формат **DOS/4GW**, работоспособна для одномодульного **4GW/PRO**); решает проблему тулза **dos4g/link**, которая доступна у автора или у модератора.

Рекомендуется попробовать **4GWPRO** (выдрать из игрушек с помощью **pmwbind.exe** или **dos4g/link**), усеченный вариант **DOS/4GW** (в модулях **4grun.exe**, **wd.exe —** для ДОС), а также **PMODE/W**.

В поставке **DOS4G** есть **4GBIND.EXE** (но для этого надо купить или украсть **DOS4G**).

Как определить количество свободной памяти под dos4gw? Попытки использовать _memavl и _memmax не дают полную картину. Что делать?

Количество свободной памяти под экстендером — не имеет смысла, особенно если используется своппинг. Для определения наличия свободного **RAM** нужно использовать функции **DMPI**, пример использования есть в хелпе.

Как добраться до конкретного физического адреса под экстендером?

Вспомните про линейную адресацию в **dos4gw**. Он в этом плане очень правильно устроен — например, начало сегмента **0xC000** находится по линейному адресу **0x000C0000**.

Вот примерчик, который печатает сигнатуру **VGA** биоса.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
void main( void )

unsigned i;
for( i = 0; i < 256; i++)

char c;
c = *(char*)( 0x000C0000 + i );
putchar( isprint( c ) ? c: '.' );
```

Как отлаживать программы, работающие в Pharlap режиме?

```
wd /tr=pls file.exe,
```

для фарлапа или

```
wd /tr=rsi file.exe
```

для **dos4gw.**

Какая разница между dos4gw и Pharlap? Или это одно и то же?

Это разные экстендеры. Самая существенная для вас разница — **dos4gw** входит в поставку Ваткома, а фарлап — нет.

Что такое RUN386.EXE? Вроде с его помощью можно пускать фарлаповые exe-шники?

Это рантайм фарлапа. Но он денег стоит.

DOS4GW такой огромный (больше 200 к), что можно использовать вместо него, чтобы поменьше диска занимало?

Существует шароварный экстендер **PMODE**, у которого есть версия, рассчитанная на **Watcom — PMODE/W**, ее можно использовать вместо **DOS4GW**, она занимает всего 9 к, встраивается внутрь **exe**-файла.

Он не обрабатывает некоторые исключительные ситуации, поэтому отлаживать программу все-таки лучше с **dos4gw**, а встраивать **pmode/w** лишь в окончательный вариант.

100% совместимости с **dos4gw** никто, конечно, гарантировать не может, но говорят, что под ним удалось запустить даже **D00M.**

Более того, 100% совместимости просто нет — например, графические программы под **DOS/4G** в Цинке, которые определяют наличие **DOS/4GW** путем вызова **int 21h, ah = 0xFF**.

При этом **DOS/4GW** и **PMODE/W** возвращают различный (хотя и похожий) результат.

А также **DOS/4G** «подобные»:

- **WDOSX** (последняя версия 0.94, size ~12 Kb)
- **DOS32A** (последняя версия 4.30, size ~20 Kb)

В чем отличия между DOS4GW и DOS4GW PRO?

DOS4GW:

- используется в виде отдельного **.EXE** модуля, имеет ограничения по размеру виртуальной памяти (16 Мб), ограничение по общей используемой памяти (32 Мб)
- отсутствует поддержка некоторых **DPMI** вызовов (например **303h — allocate callback**)
- отсутствует возможность писать **TSR**'ы
- отсутствует поддержка **DLL**, freeware
- 4GWPRO — встраивается в исполняемую программу
- ограничений в размере виртуальной памяти нет
- полная поддержка **DPMI 1.0**
- поддержка **DLL**
- поддержка **TSR**
- стоит денег.

DOS4G:

- не привязан к конкретному компилятору
- возможен запуск нескольких **.EXE**'шников под одним экстендером
- поддержка **DLL** документирована
- обильная документация

- стоит больших денег.

В процессе экспериментов выяснилось, что поддержка виртуальной памяти (**VMM — virtual memory manager**) и поддержка полного набора **DPMI** вызовов присутствуют не во всех вариантах **4GWPRO**.

Можно ли поиметь 4GWPRO даром?

Да, можно. Для этого его надо «вырезать» из головы программы собранной с **4GWPRO**. Обычно такая программа при запуске сама об этом сообщает.

Однако не из любой программы можно получить полноценный экстендер.

Ниже приведен список программ подвергшихся обрезанию и результаты.

- **ACMAIN.EXE**,
- **DESCENT.EXE**,
- **HB5.EXE**,
- **HEROES.EXE**

дают версию **1.97** с полным набором прелестей. Размер: 217764 байта.

- **ABUSE.EXE**,
- **BK.EXE**,
- **HEXEN.EXE**,
- **ROTT.EXE**,
- **TV.EXE** (**Terminal Velocity**)

дают версию **1.97** без **VMM** и поддержки расширенного набора **DPMI**.

Размер: 157268 байт.

- **ACRODOS.EXE** (**Acrobat reader for DOS**)

дает версию **1.97** с **VMM,** но без расширенного набора **DPMI**.

Размер: 203700 байт.

- **D4GRUN.EXE** (из **Watcom 10.0a**)

дает версию **1.96** без **VMM,** но с расширенной поддержкой **DPMI** (но судя по надписям внутри — это **DOS4G,** а не **4GWPRO**). Размер: 154996 байт.

- **DOOM2.EXE**

дает версию **1.95** без поддержек **VMM** и расширенного набора **DPMI**. Размер: 152084 байт.

Как переделать программу, скомпилированную под DOS4GW для использования с полученным 4GWPRO?

```
COPY /B 4GWPRO + OLD.EXE NEW.EXE
```

Почему полученный 4GWPRO не дает использовать VMM, или не дает больше 16 Мб?

Простое шаманство поможет:

```
00000247  бит     0-3     ???
00000247: бит     4       1-VMM по умолчанию вкл., 0-выкл.
00000247: бит     5       ???
00000247: бит     6       1-подавлять заставку при старте
00000247: бит     7       ???
```

Для **1.97** размером 217764 байта.

0001BFF8: (4 байта) размер виртуальной памяти по умолчанию.

Можно ли использовать DLL с DOS4GW?

Можно, это обеспечивает утилита **DLLPOWER**.

Ищите в **SimTel**'овских архивах файлы **dllpr251.zip**, **dllpr254.zip** и может быть уже есть более поздние.

Я всю жизнь писал на Борланд-С, теперь решил перебраться на Ватком, как мне проще всего это сделать?

Перенос ваших программ возможен, но скорее всего вам придется править ваш код. Начать можно с изменения **int -> short.**

Ватком ругается на стандартные библиотечные функции, в то время как BC жует их нормально, что делать?

Большинство программ, которые нормально работали под **BC**, будут нормально компилироваться и Ваткомом, нужно лишь немного изменить код, использующий специфичные функции, реализованные в **BC**, как расширение стандартной библиотеки.

Для большинства таких функций есть аналогичная или подобная функция.

Например, вместо **gettime()** следует использовать **dos_gettime()**, вместо **findfirst — dos_find_first**, и так далее.

Обратитесь к хелпу по **BC**, вы наверняка найдете имена аналогичных функций, запросив помощь по тем именам, которые не устроили компилятор Ваткома.

Кроме того, следует помнить, что например **random(num)** не является стандартной функцией, а это просто макрос из **stdlib.h**, и вместо него можно использовать конструкцию типа (**rand() % num**).

Можно ли перенести под Ватком программы, написанные с применением OWL или TVision?

OWL — скорее всего нет, поскольку он построен на расширениях синтаксиса, которые не обрабатывается компилятором Ватком. Для переноса **TVision** есть несколько вариантов.

Существуют **diff**ы для преобразования **TV** под **GNU C++** (продукт называется **GVISION**. Это не решает разом все проблемы, естественно, но по крайней мере этот вариант **TV** заточен под 32 бита и флат модель.

Совсем недавно стали доступны два порта **TV** под **Watcom** C++.

Первый — это перенос под полуось и **dos4gw TV 1.0**.

Второй имеет внутри маленькую доку и собственно сырец+**makefile**. Доку рекомендуем внимательно прочесть — может пригодится при перекомпиляции ваших программ.

В моей программе используются inline ассемблер и псевдорегистры, смогу ли я заставить их работать под Ваткомом?

Нет. Придется переписать на встроенном ассемблере, либо вынести в отдельный **.asm** модуль.

С 11 версии поддерживает стиль a la Borland:

```
_asm ...
```

А нельзя ли как-нибудь на ваткоме реализовать _AX из Borland? А то переношу под него библиотеку, а там они активно юзаются

Если вам **_AX** нужен на **read-only**, то можно сделать прозрачно:

```
=== Cut ===
short reg_ax ( void ) ;
#define _AX        reg_ax()
#pragma aux        reg_ax = \
value [ax]
#if defined( __386__ ) || defined( __FLAT__ )
int        reg_eax ( void ) ;
#define _EAX reg_eax()
#pragma aux        reg_eax = \
```

```
value [eax]
#define   rAX reg_eax() // чтобы не задумываться о контексте
#else
#define   rAX reg_ax()
#endif
=== Cut ===
```

А если для модификации, то лучше не полениться и сделать как просит Ватком (то бишь оформите этот фрагмент как **inline-asm**).

Встречается проблема, когда надо собирать смешанный проект — часть модулей компилится Ваткомом, а часть Борландом (например ассемблером). Линкер падает по трапу, вываливается с бредовыми ошибками и вообще ведет себя плохо. Что делать?

На худой конец есть способ:

- борландовый

```
obj->wdisasm->.asm->wasm->
```

- ваткомовский

```
obj
```

А дальше это отдавать как обычно **wlink'y**.

Есть еще народное средство — нужно взять **tasm 3.2**, а еще лучше **tasm 4.0**, последний хорош тем, что имеет режимы совместимости по синтаксису со всеми своими предками...

Или **TASM32 5.0** с патчем (обязательно 32 bits)

При задании надписей заголовков окон, меню, кнопок и т.п. на русском языке все прекрасно видно пока я нахожусь в режиме дизайнера. Стоит только запустить созданную аппликуху — кириллица исчезает напрочь

Замените **WRC.DLL** из поставки **Optima 1.0** на **WRC.EXE** из поставки **Watcom 11.0** и все придет в норму.

Какой компилятор C (C++) лучше всех? Что лучше: Watcom C++ или Borland C++? Посоветуйте самый крутой компилятор?!

Смотря для чего. Если нужна многоплатформенность и хорошая кодогенерация, то лучше — Ватком. Но следует учитывать, что производство Ваткома прекращено и компилятор уже не соответствует в полной мере стандарту C++, и уже никогда не будет соответствовать. А вот для разработки приложений под Win32 лучше будет Borland C++ Builder, хотя качество кодогенерации у него ниже и ни о какой многоплатформенности говорить не приходится. Что же касается BC++ 3.1 — это не более, чем учебный компилятор, и он уже давно забыт.

Не следует забывать и о **gcc**, который есть подо все мыслимые платформы и почти полностью поддерживает стандарт, однако он (точнее, его Win32 версия — cygwin) не слишком удобна для создания оконных Win32 приложений, с консольными — все в порядке, причем консольные приложения можно создавать и досовой версией — **djgpp**, дополненной пакетом **rsxnt**.

А вообще — рассуждать, какой компилятор лучше, достаточно бесполезное занятие. Идеального компилятора не существует в природе. Совершенно негодные компиляторы не имеют широкого распространения, а тот десяток компиляторов (не считая специфических кросс-компиляторов под разные встроенные системы), которые имеют широкое распространение, имеют свои достоинства и недостатки, так что приходится подбирать компилятор, наиболее подходящий для решения конкретной задачи.

Есть ли в Watcom встроенный ассемблер?

Встроенного asm'a у него на самом деле нет. Есть правда возможность писать asm-функции через **#pragma aux ...**. Например:

```
#pragma aux DWordsMover =   \
  "mov esi, eax",   \
  "mov edi, ebx",   \
  "jcxz @@skipDwordsMover", \
  "rep movsd", \
  "@@skipDWordsMover:",     \
  parm [ebx] [eax] [ecx] modify [esi edi ecx]
void DWordsMover(void* dst, void* src, size_t sz);
```

В версии 11.0 точно имеется **asm{}**.

BC не хочет понимать метки в ассемблерной вставке — компилятор сказал, что не определена эта самая метка. Пришлось определить метку за пределами ASM-блока. Может быть есть более корректное решение?

Загляни в исходники RTL от BC++ 3.1 и увидишь там нечто красивое, например:

```
#define I asm
//........
I or si,si
I jz m1
I mov dx,1
m1:
I int 21h
```

и т.д.

Другой способ — компилировать с ключом **-B**. Правда, при этом могут возникнуть другие проблемы: если присутствуют имена **read** и **_read** (например), то компилятор в них запутается.

Было замечено, что борланд (3.1, например) иногда генерит разный код в зависимости от ключа **-B**. Как правило, при его наличии он становится «осторожнее» — начинает понимать, что не он один использует регистры.

Возврат адреса/ссылки локальных переменных — почему возвращает фигню: char* myview() { char s(SIZE); ... return s; }

Нужно поставить **static char s[SIZE];** чтобы возвращался адрес всегда существующей (статической) переменной, а автоматические переменные исчезают при выходе из функции — освобождается и может быть замусорено место из-под этих переменных на стеке.

Какие ограничения на имена с подчёркиванием?

При использовании зарезервированных имён (то есть с подчёркиваниями) возможен самый разный **undefined behavior**. Например, в компиляторе все слова с двойным подчёркиванием могут использоваться для управления файловой системой. Именно поэтому вполне допустимо (по стандарту), если Борланд в своём компиляторе, например, при встрече «нестандартной» лексемы **__asm** из сорца для VC++ просто потрёт какой-нибудь файл. На практике такого рода вариации **undefined behavior** встретить сложно, но вполне возможно.

Другие вариации **undefined behavior** — это всякие глюки при работе программы. То есть, если мы, например, в **printf** задействуем неизвестный библиотеке (нестандартный) флаг, то по стандарту вполне допустимо не проверять в библиотеке подобную фигню и передать управление куда-нибудь в область данных (ну нет в таблице переходов такого флага!).

Поведение переменных, описанных в заголовке цикла for

Переменная, объявленная в заголовке цикла (и прочих операторов!) действительна только внутри этого цикла.

Тем, кто хочет заставить вести себя также свои старые компилеры, это можно сделать следующим способом через **define**:

В новых редакциях C++ область видимости определённой в заголовке **for** переменной ограничивают телом цикла. Следующая подстановка ограничивает область видимости и для старых редакций, в которых она распространяется за пределы цикла:

```
#define for if(0);else for
```

а также для BC++ выключим вызываемые **if(0)** предупреждения:


```
"Condition is always false"
#pragma warn -ccc
```

Что есть const после имени метода?

Это означает, что этот метод не будет менять данные класса.

Методы с модификатором **const** могут изменять данные объекта, помеченные модификатором **mutable**.

```
class Bart
{
private:
 mutable int m_iSomething;
public:
 void addToSomething( int iValue ) const
 {
 m_iSomething += iValue;
 }
 Bart()
 {
 m_iSomething = 0;
 }
};
const Bart bartObject;
bartObject.addToSomething( 8 );
```

Будет скомпилировано и выполнено.

Как инициализировать статические члены класса?

```
struct a { static int i; };
int a::i; //зачем это нужно?
int main() { a::i = 11; return 0; }
```

А вот зачем:

```
struct b { int n; b(int i) :n(i) {} };
struct a { static b i; };
b a::i(0);
int main() { printf("%i\n",a::i.n); return 0; }
```

Описание некоего типа и переменная этого типа — не одно и то же. Где вы предлагаете размещать и конструировать статические поля? Может быть, в каждом файле, который включает заголовок с описанием класса? Или где?

Как не ошибиться в размере аллокируемого блока?

Для избежания подобного можно в C сымитировать Сиплюсный new:

```
#define tmalloc(type)       ((type*)malloc(sizeof(type)))
#define amalloc(type, size) ((type*)malloc(sizeof(type) * (size)))
```

Более того, в последнем **define** можно поставить **(size) + 1**, чтобы гарантированно избежать проблем с завершающим нулём в строках.

Можно сделать иначе. Поскольку присвоение от **malloc()** как правило делают на типизованную переменную, то можно прямо так и написать:

```
body = malloc(sizeof(*body));
```

теперь спокойно можно менять типы не заботясь о **malloc()**. Но это верно для Си, который не ругается на присвоение **void*** к **type*** (иначе пришлось бы кастить поинтер, и компилятор изменения типа просто не пережил бы).

Вообще в C нет смысла ставить преобразования от **void*** к указательному типу явно. Более того, этот код не переносим на C++ — в проекте стандарта C++ нет **malloc()** и **free()**, их нет даже в **hosted c++** заголовках. Проще будет:

```
#ifdef __cplusplus
# define tmalloc(type)      (new type)
# define amalloc(type, size) (new type[size])
#else
# define tmalloc(type)      malloc(sizeof(type))
# define amalloc(type, size) malloc(sizeof(type) * (size))
#endif
```

Что такое ссылка?

Ссылка — это псевдоним (другое имя) для объекта.

Ссылки часто используются для передачи параметра по ссылке:

```
void swap(int& i, int& j)
 {
  int tmp = i;
  i = j;
  j = tmp;
 }

 int main()
 {
  int x, y;
  // ...
  swap(x,y);
 }
```


В этом примере **i** и **j** — псевдонимы для переменных **x** и **y** функции **main**. Другими словами, **i** — это **x**. Не указатель на **x** и не копия **x**, а сам **x**. Все, что вы делаете с **i**, проделывается с **x**, и наоборот.

Вот таким образом вы как программист должны воспринимать ссылки. Теперь, рискуя дать вам неверное представление, несколько слов о том, каков механизм работы ссылок. В основе ссылки **i** на объект **x** — лежит, как правило, просто машинный адрес объекта **x**. Но когда вы пишете **i++**, компилятор генерирует код, который инкрементирует **x**. В частности, сам адрес, который компилятор использует, чтобы найти **x**, остается неизменным. Программист на C может думать об этом, как если бы использовалась передача параметра по указателю, в духе языка C, но, во-первых, **&** (взятие адреса) было бы перемещено из вызывающей функции в вызываемую, и, во-вторых, в вызываемой функции были бы убраны ***** (разыменование). Другими словами, программист на C может думать об **i** как о макроопределении для **(*p)**, где **p** — это указатель на **x** (т.е., компилятор автоматически разыменовывает подлежащий указатель: **i++** заменяется на **(*p)++**, а **i = 7** на ***p = 7**).

Важное замечание: несмотря на то что в качестве ссылки в окончательном машинном коде часто используется адрес, не думайте о ссылке просто как о забавно выглядящем указателе на объект. Ссылка — это объект. Это не указатель на объект и не копия объекта. Это сам объект.

Что происходит в результате присваивания ссылке?

Вы меняете состояние ссыльного объекта (того, на который ссылается ссылка).

Помните: ссылка — это сам объект, поэтому, изменяя ссылку, вы меняете состояние объекта, на который она ссылается. На языке производителей компиляторов ссылка — это **lvalue** (**left value** — значение, которое может появиться слева от оператора присваивания).

Что происходит, когда я возвращаю из функции ссылку?

В этом случае вызов функции может оказаться с левой стороны оператора (операции) присваивания.

На первый взгляд, такая запись может показаться странной. Например, запись **f() = 7** выглядит бессмысленной. Однако, если **a** — это объект класса **Array**, для большинства людей запись **a[i] = 7** является осмысленной, хотя **a[i]** — это всего лишь замаскированный вызов функции **Array::operator[](int)**, которая является оператором обращения по индексу для класса **Array**:

```
class Array {
 public:
  int size() const;
  float& operator[] (int index);
  // ...
 };

 int main()
 {
  Array a;
  for (int i = 0; i < a.size(); ++i)
a[i] = 7; // В этой строке вызывается Array::operator[](int)
 }
```

Как можно переустановить ссылку, чтобы она ссылалась на другой объект?

Невозможно в принципе.

Невозможно отделить ссылку от ее объекта.

В отличие от указателя, ссылка, как только она привязана к объекту, не может быть «перенаправлена» на другой объект. Ссылка сама по себе ничего не представляет, у нее нет имени, она сама — это другое имя для объекта. Взятие адреса ссылки дает адрес объекта, на который она ссылается. Помните: ссылка — это объект, на который она ссылается.

С этой точки зрения, ссылка похожа на **const** указатель, такой как **int* const p** (в отличие от указателя на **const**, такого как **const int* p**). Несмотря на большую схожесть, не путайте ссылки с указателями — это не одно и то же.

В каких случаях мне стоит использовать ссылки, и в каких — указатели?

Используйте ссылки, когда можете, а указатели — когда это необходимо.

Ссылки обычно предпочтительней указателей, когда вам не нужно их «перенаправлять». Это обычно означает, что ссылки особенно полезны в открытой (**public**) части класса. Ссылки обычно появляются на поверхности объекта, а указатели спрятаны внутри.

Исключением является тот случай, когда параметр или возвращаемый из функции объект требует выделения «охранного» значения для особых случаев. Это обычно реализуется путем взятия/возвращения указателя, и обозначением особого случая при помощи передачи нулевого указателя (**NULL**).

Ссылка же не может ссылаться на разыменованный нулевой указатель.

Примечание: программисты с опытом работы на C часто недолюбливают ссылки, из-за того что передача параметра по ссылке явно никак не обозначается в вызывающем коде. Однако с обретением некоторого опыта работы на C++, они осознают, что это одна из форм сокрытия информации, которая является скорее преимуществом, чем недостатком. Т.е., программисту следует писать код в терминах задачи, а не компьютера (programmers should write code in the language of the problem rather than the language of the machine).

Что такое встроенная функция?

Встроенная функция — это функция, код которой прямо вставляется в том месте, где она вызвана. Как и макросы, определенные через **#define**, встроенные функции улучшают производительность за счет стоимости вызова и (особенно!) за счет возможности дополнительной оптимизации («процедурная интеграция»).

Как встроенные функции могут влиять на соотношение безопасности и скорости?

В обычном C вы можете получить «инкапсулированные структуры», помещая в них указатель на **void**, и заставляя его указывать на настоящие данные, тип которых неизвестен пользователям структуры. Таким образом, пользователи не знают, как интерпретировать эти данные, а функции доступа преобразуют указатель на **void** к нужному скрытому типу. Так достигается некоторый уровень инкапсуляции.

К сожалению, этот метод идет вразрез с безопасностью типов, а также требует вызова функции для доступа к любым полям структуры (если вы позволили бы прямой доступ, то его мог бы получить кто угодно, поскольку будет известно, как интерпретировать данные, на которые указывает **void***. Такое поведение со стороны пользователя приведет к сложностям при последующем изменении структуры подлежащих данных).

Стоимость вызова функции невелика, но дает некоторую прибавку. Классы C++ позволяют встраивание функций, что дает вам безопасность инкапсуляции вместе со скоростью прямого доступа. Более того, типы параметры встраиваемых функций проверяются компилятором, что является преимуществом по сравнению с сишными **#define** макросами.

Зачем мне использовать встроенные функции? Почему не использовать просто #define макросы?

Поскольку **#define** макросы опасны.

В отличие от **#define** макросов, встроенные (**inline**) функции не подвержены известным ошибкам двойного вычисления, поскольку каждый аргумент встроенной функции вычисляется только один раз. Другими словами, вызов встроенной функции — это то же самое что и вызов обычной функции, только быстрее:

```
// Макрос, возвращающий модуль (абсолютное значение) i
  #define unsafe(i) \
( (i) >= 0 ? (i) : -(i) )
// Встроенная функция, возвращающая абсолютное значение i
  inline
  int safe(int i)
  {
  return i >= 0 ? i : -i;
  }
  int f();
  void userCode(int x)
  {
  int ans;
  ans = unsafe(x++); // Ошибка! x инкрементируется дважды
  ans = unsafe(f()); // Опасно! f() вызывается дважды
  ans = safe(x++); // Верно! x инкрементируется один раз
  ans = safe(f()); // Верно! f() вызывается один раз
  }
```

Также, в отличие от макросов, типы аргументов встроенных функций проверяются, и выполняются все необходимые преобразования.

Макросы вредны для здоровья; не используйте их, если это не необходимо.

Что такое ошибка в порядке статической инициализации («static initialization order fiasco»)?

Незаметный и коварный способ убить ваш проект.

Ошибка порядка статической инициализации — это очень тонкий и часто неверно воспринимаемый аспект C++. К сожалению, подобную ошибку очень сложно отловить, поскольку она происходит до вхождения в функцию **main()**.

Представьте себе, что у вас есть два статических объекта **x** и **y**, которые находятся в двух разных исходных файлах, скажем **x.cpp** и **y.cpp**. И путь конструктор объекта y вызывает какой-либо метод объекта **x**.

Вот и все. Так просто.

Проблема в том, что у вас ровно пятидесятипроцентная возможность катастрофы. Если случится, что единица трансляции с **x.cpp** будет проинициализирована первой, то все в порядке. Если же первой будет проинициализирована единица трансляции файла **y.cpp**, тогда конструктор объекта y будет запущен до конструктора **x**, и вам крышка. Т.е., конструктор **y** вызовет метод объекта **x**, когда сам **x** еще не создан.

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации. Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как предотвратить ошибку в порядке статической инициализации?

Используйте «создание при первом использовании», то есть, поместите ваш статический объект в функцию.

Представьте себе, что у нас есть два класса **Fred** и **Barney**. Есть глобальный объект типа **Fred**, с именем **x**, и глобальный объект типа **Barney**, с именем **y**. Конструктор **Barney** вызывает метод **goBowling()** объекта **x**. Файл **x.cpp** содержит определение объекта **x**:

```
// File x.cpp
 #include "Fred.hpp"
 Fred x;
```

Файл **y.cpp** содержит определение объекта **y**:

```
// File y.cpp
 #include "Barney.hpp"
 Barney y;
```

Для полноты представим, что конструктор **Barney::Barney()** выглядит следующим образом:

```
// File Barney.cpp
 #include "Barney.hpp"
 Barney::Barney()
 {
  // ...
  x.goBowling();
  // ...
 }
```

Проблема случается, если **y** создается раньше, чем **x**, что происходит в 50% случаев, поскольку **x** и **y** находятся в разных исходных файлах.

Есть много решений для этой проблемы, но одно очень простое и переносимое — заменить глобальный объект **Fred x**, глобальной функцией **x()**, которая возвращает объект типа **Fred** по ссылке.

```
// File x.cpp
#include "Fred.hpp"
 Fred& x()
 {
  static Fred* ans = new Fred();
  return *ans;
 }
```

Поскольку локальные статические объекты создаются в момент, когда программа в процессе работы в первый раз проходит через точку их объявления, инструкция **new Fred()** в примере выше будет выполнена только один раз: во время первого вызова функции **x()**. Каждый последующий вызов возвратит тот же самый объект **Fred** (тот, на который указывает **ans**). И далее все случаи использования объекта **x** замените на вызовы функции **x()**:

```
// File Barney.cpp
 #include "Barney.hpp"
 Barney::Barney()
 {
  // ...
  x().goBowling();
  // ...
 }
```

Это и называется «создание при первом использовании», глобальный объект **Fred** создается при первом обращении к нему.

Отрицательным моментом этой техники является тот факт, что объект **Fred** нигде не уничтожается.

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации. Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как бороться с ошибками порядка статической инициализации объектов — членов класса?

Предположим, у вас есть класс **X**, в котором есть статический объект **Fred**:


```
// File X.hpp
class X {
 public:
  // ...
 private:

  static Fred x_;
 };
```

Естественно, этот статический член инициализируется отдельно:

```
// File X.cpp
#include "X.hpp"
Fred X::x_;
```

Опять же естественно, объект **Fred** будет использован в одном или нескольких методах класса **X**:

```
void X::someMethod()
 {
  x_.goBowling();
 }
```

Проблема проявится, если кто-то где-то каким-либо образом вызовет этот метод, до того как объект **Fred** будет создан. Например, если кто-то создает статический объект **X** и вызывает его **someMethod()** во время статической инициализации, то ваша судьба всецело находится в руках компилятора, который либо создаст **X::x_**, до того как будет вызван **someMethod()**, либо же только после.

В любом случае, всегда можно сохранить переносимость (и это абсолютно безопасный метод), заменив статический член **X::x_** на статическую функцию-член:

```
// File X.hpp
class X {
 public:
  // ...
private:
  static Fred& x();
 };
```

Естественно, этот статический член инициализируется отдельно:

```
// File X.cpp
#include "X.hpp"
 Fred& X::x()
 {
  static Fred* ans = new Fred();
  return *ans;
```

```
}
```

После чего вы просто меняете все **x_** на **x()**:

```
void X::someMethod()
 {
  x().goBowling();
 }
```

Если для вас крайне важна скорость работы программы и вас беспокоит необходимость дополнительного вызова функции для каждого вызова **X::someMethod()**, то вы можете сделать **static Fred&**. Как вы помните, статические локальные переменные инициализируются только один раз (при первом прохождении программы через их объявление), так что **X::x()** теперь будет вызвана только один раз: во время первого вызова **X::someMethod()**:

```
void X::someMethod()
 {
  static Fred& x = X::x();
  x.goBowling();
 }
```

Примечание: ошибки статической инициализации не распространяются на базовые/встроенные типы, такие как **int** или **char***. Например, если вы создаете статическую переменную типа **float**, у вас не будет проблем с порядком инициализации. Проблема возникает только тогда, когда у вашего статического или глобального объекта есть конструктор.

Как мне обработать ошибку, которая произошла в конструкторе?

Сгенерируйте исключение.

Что такое деструктор?

Деструктор — это исполнение последней воли объекта.

Деструкторы используются для высвобождения занятых объектом ресурсов. Например, класс **Lock** может заблокировать ресурс для эксклюзивного использования, а его деструктор этот ресурс освободить. Но самый частый случай — это когда в конструкторе используется **new**, а в деструкторе — **delete**.

Деструктор это функция «готовься к смерти». Часто слово деструктор сокращается до **dtor**.

В каком порядке вызываются деструкторы для локальных объектов?

В порядке обратном тому, в каком эти объекты создавались: первым создан — последним будет уничтожен.

В следующем примере деструктор для объекта **b** будет вызван первым, а только затем деструктор для объекта **a**:

```
void userCode()
 {
  Fred a;
  Fred b;
  // ...
 }
```

В каком порядке вызываются деструкторы для массивов объектов?

В порядке обратном созданию: первым создан — последним будет уничтожен.

В следующем примере порядок вызова деструкторов будет таким: **a[9]**, **a[8]**, ..., **a[1]**, **a[0]**:

```
void userCode()
 {
  Fred a[10];
  // ...
 }
```

Могу ли я перегрузить деструктор для своего класса?

Нет.

У каждого класса может быть только один деструктор. Для класса **Fred** он всегда будет называться **Fred::~Fred()**. В деструктор никогда не передаётся никаких параметров, и сам деструктор никогда ничего не возвращает.

Всё равно вы не смогли бы указать параметры для деструктора, потому что вы никогда на вызываете деструктор напрямую (точнее, почти никогда).

Могу ли я явно вызвать деструктор для локальной переменной?

Нет!

Деструктор всё равно будет вызван еще раз при достижении закрывающей фигурной скобки **}** конца блока, в котором была создана локальная переменная. Этот вызов гарантируется языком, и он происходит автоматически; нет способа этот вызов предотвратить. Но последствия повторного вызова деструктора для одного и того же объекта могут быть плачевными. Бах! И вы покойник...

А что если я хочу, чтобы локальная переменная «умерла» раньше закрывающей фигурной скобки? Могу ли я при крайней необходимости вызвать деструктор для локальной переменной?

Нет!

Предположим, что (желаемый) побочный эффект от вызова деструктора для локального объекта **File** заключается в закрытии файла. И предположим, что у нас есть экземпляр **f** класса **File** и мы хотим, чтобы файл **f** был закрыт раньше конца своей области видимости (т.е., раньше **}**):

```
void someCode()
 {
  File f;
// ... [Этот код выполняется при открытом f] ...
// <-- Нам нужен эффект деструктора f здесь
// ... [Этот код выполняется после закрытия f] ...
 }
```

Для этой проблемы есть простое решение. Но пока запомните только следующее: нельзя явно вызывать деструктор.

Хорошо, я не буду явно вызывать деструктор. Но как мне справиться с этой проблемой?

Просто поместите вашу локальную переменную в отдельный блок **{...}**, соответствующий необходимому времени жизни этой переменной:

```
void someCode()
 {
  {
  File f;
  // ... [В этом месте f еще открыт] ...
  }
 // ^-- деструктор f будет автоматически вызван здесь!
 // ... [В этом месте f уже будет закрыт] ...
 }
```

А что делать, если я не могу поместить переменную в отдельный блок?

В большинстве случаев вы можете воспользоваться дополнительным блоком **{...}** для ограничения времени жизни вашей переменной. Но если по какой-то причине вы не можете добавить блок, добавьте функцию-член, которая будет выполнять те же действия, что и деструктор. Но помните: вы не можете сами вызывать деструктор!

Например, в случае с классом **File**, вы можете добавить метод **close()**. Обычный деструктор будет вызывать **close()**. Обратите внимание,

что метод **close()** должен будет как-то отмечать объект **File**, с тем чтобы последующие вызовы не пытались закрыть уже закрытый файл. Например, можно устанавливать переменную-член **fileHandle_** в какое-нибудь неиспользуемое значение, типа **-1**, и проверять вначале, не содержит ли **fileHandle_** значение **-1**.

```
class File {
 public:
  void close();
  ~File();
  // ...
 private:
 int fileHandle_;
 // fileHandle_ >= 0 если/только если файл открыт
 };
 File::~File()
 {
  close();
 }

 void File::close()
 {
  if (fileHandle_ >= 0) {
  // ... [Вызвать системную функцию для закрытия файла] ...
  fileHandle_ = -1;
  }
 }
```

Обратите внимание, что другим методам класса **File** тоже может понадобиться проверять, не установлен ли **fileHandle_** в **-1** (т.е., не закрыт ли файл).

Также обратите внимание, что все конструкторы, которые не открывают файл, должны устанавливать **fileHandle_** в **-1**.

А могу ли я явно вызывать деструктор для объекта, созданного при помощи new?

Скорее всего, нет.

За исключением того случая, когда вы использовали синтаксис размещения для оператора **new**, вам следует просто удалять объекты при помощи **delete**, а не вызывать явно деструктор. Предположим, что вы создали объект при помощи обычного **new**:

```
Fred* p = new Fred();
```

В таком случае деструктор **Fred::~Fred()** будет автоматически вызван, когда вы удаляете объект:

```
delete p; // Вызывает p->~Fred()
```

Вам не следует явно вызывать деструктор, поскольку этим вы не освобождаете память, выделенную для объекта **Fred**. Помните: **delete p** делает сразу две вещи: вызывает деструктор и освобождает память.

Что такое «синтаксис размещения» new («placement new») и зачем он нужен?

Есть много случаев для использования синтаксиса размещения для **new**. Самое простое — вы можете использовать синтаксис размещения для помещения объекта в определенное место в памяти. Для этого вы указываете место, передавая указатель на него в оператор **new**:

```
#include <new>  // Необходимо для использования синтаксиса размещения
#include "Fred.h"  // Определение класса Fred

 void someCode()
 {
  char memory[sizeof(Fred)];  // #1
  void* place = memory;    // #2
  Fred* f = new(place) Fred(); // #3
  // Указатели f и place будут равны
  // ...
 }
```

В строчке #1 создаётся массив из **sizeof(Fred)** байт, размер которого достаточен для хранения объекта **Fred**. В строчке #2 создаётся указатель **place**, который указывает на первый байт массива (опытные программисты на C наверняка заметят, что можно было и не создавать этот указатель; мы это сделали лишь чтобы код был более понятным). В строчке #3 фактически происходит только вызов конструктора **Fred::Fred()**. Указатель **this** в конструкторе **Fred** будет равен указателю **place**. Таким образом, возвращаемый указатель тоже будет равен **place**.

Совет: Не используйте синтаксис размещения **new**, за исключением тех случаев, когда вам действительно нужно, чтобы объект был размещён в определённом месте в памяти. Например, если у вас есть аппаратный таймер, отображённый на определённый участок памяти, то вам может понадобиться поместить объект **Clock** по этому адресу.

Опасно: Используя синтаксис размещения **new** вы берёте на себя всю ответственность за то, что передаваемый вами указатель указывает на достаточный для хранения объекта участок памяти с тем выравниванием (alignment), которое необходимо для вашего объекта. Ни компилятор, ни библиотека не будут проверять корректность ваших действий в этом случае. Если ваш класс **Fred** должен быть выровнен по четырёхбайтовой границе, но вы передали в **new** указатель на не выровненный участок памяти, у вас могут быть большие неприятности (если вы не знаете, что такое «выравнивание» (alignment), пожалуйста, не используйте синтаксис размещения **new**). Мы вас предупредили.

Также на вас ложится вся ответственность по уничтожению размещённого объекта. Для этого вам необходимо явно вызвать деструктор:

```
void someCode()
{
 char memory[sizeof(Fred)];
 void* p = memory;
 Fred* f = new(p) Fred();
 f->~Fred();
 // Явный вызов деструктора для размещённого объекта
}
```

Это практически единственный случай, когда вам нужно явно вызывать деструктор.

Когда я пишу деструктор, должен ли я явно вызывать деструкторы для объектов-членов моего класса?

Нет. Никогда не надо явно вызывать деструктор (за исключением случая с синтаксисом размещения **new**.

Деструктор класса (неявный, созданный компилятором, или явно описанный вами) автоматически вызывает деструкторы объектов-членов класса. Эти объекты уничтожаются в порядке обратном порядку их объявления в теле класса:

```
class Member {
 public:
  ~Member();
  // ...
 };

 class Fred {
 public:
  ~Fred();
  // ...
 private:
```

```
 Member x_;
 Member y_;
 Member z_;
};

Fred::~Fred()
{
 // Компилятор автоматически вызывает z_.~Member()
 // Компилятор автоматически вызывает y_.~Member()
 // Компилятор автоматически вызывает x_.~Member()
}
```

Когда я пишу деструктор производного класса, нужно ли мне явно вызывать деструктор предка?

Нет. Никогда не надо явно вызывать деструктор (за исключением случая с синтаксисом размещения **new**).

Деструктор производного класса (неявный, созданный компилятором, или явно описанный вами) автоматически вызывает деструкторы предков. Предки уничтожаются после уничтожения объектов-членов производного класса. В случае множественного наследования непосредственные предки класса уничтожаются в порядке обратном порядку их появления в списке наследования.

```
class Member {
 public:
  ~Member();
  // ...
 };
 class Base {
 public:
  virtual ~Base();  // Виртуальный деструктор[20.4]
  // ...
 };
 class Derived : public Base {
 public:
  ~Derived();
  // ...
 private:
  Member x_;
 };

 Derived::~Derived()
 {
  // Компилятор автоматически вызывает x_.~Member()
```



```
  // Компилятор автоматически вызывает Base::~Base()
 }
```

Примечание: в случае виртуального наследования порядок уничтожения классов сложнее. Если вы полагаетесь на порядок уничтожения классов в случае виртуального наследования, вам понадобится больше информации, чем изложено здесь.

Расскажите все-таки о пресловутых нулевых указателях

Для каждого типа указателей существует (согласно определению языка) особое значение — «нулевой указатель», которое отлично от всех других значений и не указывает на какой-либо объект или функцию. Таким образом, ни оператор **&**, ни успешный вызов **malloc()** никогда не приведут к появлению нулевого указателя. (**malloc** возвращает нулевой указатель, когда память выделить не удается, и это типичный пример использования нулевых указателей как особых величин, имеющих несколько иной смысл «память не выделена» или «теперь ни на что не указываю».)

Нулевой указатель принципиально отличается от неинициализированного указателя. Известно, что нулевой указатель не ссылается ни на какой объект; неинициализированный указатель может ссылаться на что угодно.

В приведенном выше определении уже упоминалось, что существует нулевой указатель для каждого типа указателя, и внутренние значения нулевых указателей разных типов могут отличаться. Хотя программистам не обязательно знать внутренние значения, компилятору всегда необходима информация о типе указателя, чтобы различить нулевые указатели, когда это нужно.

Как «получить» нулевой указатель в программе?

В языке C константа **0**, когда она распознается как указатель, преобразуется компилятором в нулевой указатель. То есть, если во время инициализации, присваивания или сравнения с одной стороны стоит переменная или выражение, имеющее тип указателя, компилятор решает, что константа **0** с другой стороны должна превратиться в нулевой указатель и генерирует нулевой указатель нужного типа.

Следовательно, следующий фрагмент абсолютно корректен:

```
char *p = 0;
if(p != 0)
```

Однако, аргумент, передаваемый функции, не обязательно будет распознан как значение указателя, и компилятор может оказаться не способным распознать голый 0 как нулевой указатель. Например, сис-

темный вызов UNIX «execl» использует в качестве параметров переменное количество указателей на аргументы, завершаемое нулевым указателем. Чтобы получить нулевой указатель при вызове функции, обычно необходимо явное приведение типов, чтобы 0 воспринимался как нулевой указатель.

```
execl("/bin/sh", "sh", "-c", "ls", (char *)0);
```

Если не делать преобразования (char *), компилятор не поймет, что необходимо передать нулевой указатель и вместо этого передаст число 0. (Заметьте, что многие руководства по UNIX неправильно объясняют этот пример.)

Когда прототипы функций находятся в области видимости, передача аргументов идет в соответствии с прототипом и большинство приведений типов может быть опущено, так как прототип указывает компилятору, что необходим указатель определенного типа, давая возможность правильно преобразовать нули в указатели. Прототипы функций не могут, однако, обеспечить правильное преобразование типов в случае, когда функция имеет список аргументов переменной длины, так что для таких аргументов необходимы явные преобразования типов. Всегда безопаснее явные преобразования в нулевой указатель, чтобы не наткнуться на функцию с переменным числом аргументов или на функцию без прототипа, чтобы временно использовать не-ANSI компиляторы, чтобы продемонстрировать, что вы знаете, что делаете. (Кстати, самое простое правило для запоминания.)

Что такое NULL и как он определен с помощью #define?

Многим программистам не нравятся нули, беспорядочно разбросанные по программам. По этой причине макрос препроцессора **NULL** определен в **<stdio.h>** или **<stddef.h>** как значение **0**. Программист, который хочет явно различать **0** как целое и **0** как нулевой указатель может использовать **NULL** в тех местах, где необходим нулевой указатель. Это только стилистическое соглашение; препроцессор преобразует **NULL** опять в 0, который затем распознается компилятором в соответствующем контексте как нулевой указатель. В отдельных случаях при передаче параметров функции может все же потребоваться явное указание типа перед **NULL** (как и перед 0).

Как #define должен определять NULL на машинах, использующих ненулевой двоичный код для внутреннего представления нулевого указателя?

Программистам нет необходимости знать внутреннее представление(я) нулевых указателей, ведь об этом обычно заботится компилятор.

Если машина использует ненулевой код для представления нулевых указателей, на совести компилятора генерировать этот код, когда программист обозначает нулевой указатель как **"0"** или **NULL**.

Следовательно, определение **NULL** как **0** на машине, для которой нулевые указатели представляются ненулевыми значениями так же правомерно как и на любой другой, так как компилятор должен (и может) генерировать корректные значения нулевых указателей в ответ на **0**, встретившийся в соответствующем контексте.

Пусть NULL был определен следующим образом: #define NULL ((char *)0). Означает ли это, что функциям можно передавать NULL без преобразования типа?

В общем, нет. Проблема в том, что существуют компьютеры, которые используют различные внутренние представления для указателей на различные типы данных. Предложенное определение через **#define** годится, когда функция ожидает в качестве передаваемого параметра указатель на **char**, но могут возникнуть проблемы при передаче указателей на переменные других типов, а верная конструкция:

```
FILE *fp = NULL;
```

может не сработать.

Тем не менее, ANSI C допускает другое определение для **NULL**:

```
#define NULL ((void *)0)
```

Кроме помощи в работе некорректным программам (но только в случае машин, где указатели на разные типы имеют одинаковые размеры, так что помощь здесь сомнительна) это определение может выявить программы, которые неверно используют **NULL** (например, когда был необходим символ ASCII NUL).

Я использую макрос #define Nullptr(type) (type *)0, который помогает задавать тип нулевого указателя

Хотя этот трюк и популярен в определенных кругах, он стоит немного. Он не нужен при сравнении и присваивании. Он даже не экономит буквы.

Его использование показывает тому, кто читает программу, что автор здорово «сечет» в нулевых указателях, и требует гораздо более аккуратной проверки определения макроса, его использования и всех остальных случаев применения указателей.

Корректно ли использовать сокращенный условный оператор if(p) для проверки того, что указатель ненулевой? А что если внутреннее представление для нулевых указателей отлично от нуля?

Когда C требует логическое значение выражения (в инструкциях **if**, **while**, **for** и **do** и для операторов **&&**, **||**, **!** и **?**) значение **false** получается, когда выражение равно нулю, а значение **true** получается в противоположном случае. Таким образом, если написано:

```
if(expr)
```

где «**expr**» — произвольное выражение, компилятор на самом деле поступает так, как будто было написано:

```
if(expr != 0)
```

Подставляя тривиальное выражение, содержащее указатель «p» вместо «**expr**», получим:

```
if(p)
```

эквивалентно

```
if(p != 0)
```

и это случай, когда происходит сравнение, так что компилятор поймет, что неявный ноль — это нулевой указатель и будет использовать правильное значение. Здесь нет никакого подвоха, компиляторы работают именно так и генерируют в обоих случаях идентичный код. Внутреннее представление указателя не имеет значения.

Оператор логического отрицания **!** может быть описан так:

```
!expr
```

на самом деле эквивалентно

```
expr?0:1
```

Читателю предлагается в качестве упражнения показать, что

```
if(!p)
```

эквивалентно

```
if(p == 0)
```

Хотя «сокращения» типа **if(p)** совершенно корректны, кое-кто считает их использование дурным стилем.

Если «NULL» и «0» эквивалентны, то какую форму из двух использовать?

Многие программисты верят, что «NULL» должен использоваться во всех выражениях, содержащих указатели как напоминание о том, что значение должно рассматриваться как указатель. Другие же чувствуют, что путаница, окружающая «**NULL**» и «**0**», только усугубляется, если «**0**» спрятать в операторе **#define** и предпочитают использовать «**0**» вместо «**NULL**». Единственного ответа не существует. Программисты на C должны понимать, что «**NULL**» и «**0**» взаимозаменяемы и что «**0**» без преобразования типа можно без сомнения использовать при инициализации, присваивании и сравнении. Любое использование «**NULL**» (в противоположность «**0**») должно рассматриваться как ненавязчивое напоминание, что используется указатель; программистам не нужно ничего делать (как для своего собственного понимания, так и для компилятора) для того, чтобы отличать нулевые указатели от целого числа **0**. **NULL** нельзя использовать, когда необходим другой тип нуля.

Даже если это и будет работать, с точки зрения стиля программирования это плохо.(ANSI позволяет определить **NULL** с помощью **#define как (void *)0**. Такое определение не позволит использовать **NULL** там, где не подразумеваются указатели). Особенно не рекомендуется использовать **NULL** там, где требуется нулевой код ASCII (NUL). Если необходимо, напишите собственное определение:

```
#define NUL '\0'
```

Но не лучше ли будет использовать NULL (вместо 0) в случае, когда значение NULL изменяется, быть может, на компьютере с ненулевым внутренним представлением нулевых указателей?

Нет. Хотя символические константы часто используются вместо чисел из-за того, что числа могут измениться, в данном случае причина, по которой используется **NULL**, иная. Еще раз повторим: язык гарантирует, что **0**, встреченный там, где по контексту подразумевается указатель, будет заменен компилятором на нулевой указатель. **NULL** используется только с точки зрения лучшего стиля программирования.

Я в растерянности. Гарантируется, что NULL равен 0, а нулевой указатель нет?

Термин «**null**» или «**NULL**» может не совсем обдуманно использоваться в нескольких смыслах:

1. Нулевой указатель как абстрактное понятие языка.

2. Внутреннее (на стадии выполнения) представление нулевого указателя, которое может быть отлично от нуля и различаться для различных типов указателей. О внутреннем представлении нулевого указателя должны заботиться только создатели компилятора. Программистам на C это представление не известно.

3. Синтаксическое соглашение для нулевых указателей, символ «0».

4. Макрос **NULL** который с помощью **#define** определен как «**0**» или «**(void *)0**».

5. Нулевой код ASCII (NUL), в котором все биты равны нулю,но который имеет мало общего с нулевым указателем, разве что названия похожи.

6. «Нулевой стринг», или, что то же самое, пустой стринг ("").

Почему так много путаницы связано с нулевыми указателями? Почему так часто возникают вопросы?

Программисты на С традиционно хотят знать больше, чем это необходимо для программирования, о внутреннем представлении кода. Тот факт, что внутреннее представление нулевых указателей для большинства машин совпадает с их представлением в исходном тексте, т.е. нулем, способствует появлению неверных обобщений. Использование макроса (**NULL**) предполагает, что значение может впоследствии измениться, или иметь другое значение для какого-нибудь компьютера. Конструкция **if(p == 0)** может быть истолкована неверно, как преобразование перед сравнением **p** к целому типу, а не **0** к типу указателя. Наконец, часто не замечают, что термин «**null**» употребляется в разных смыслах (перечисленных выше).

Хороший способ устранить путаницу — вообразить, что язык С имеет ключевое слово (возможно, **nil**, как в Паскале), которое обозначает нулевой указатель. Компилятор либо преобразует «**nil**» в нулевой указатель нужного типа, либо сообщает об ошибке, когда этого сделать нельзя. На самом деле, ключевое слово для нулевого указателя в С — это не «**nil**» а «**0**». Это ключевое слово работает всегда, за исключением случая, когда компилятор воспринимает в неподходящем контексте «0» без указания типа как целое число, равное нулю, вместо того, чтобы сообщить об ошибке. Программа может не работать, если предполагалось, что «**0**» без явного указания типа — это нулевой указатель.

Я все еще в замешательстве. Мне так и не понятна возня с нулевыми указателями

Следуйте двум простым правилам:

1. Для обозначения в исходном тексте нулевого указателя, используйте «**0**» или «**NULL**».

2. Если «**0**» или «**NULL**» используются как фактические аргументы при вызове функции, приведите их к типу указателя, который ожидает вызываемая функция.

Учитывая всю эту путаницу, связанную с нулевыми указателями, не лучше ли просто потребовать, чтобы их внутреннее представление было нулевым?

Если причина только в этом, то поступать так было бы неразумно, так как это неоправданно ограничит конкретную реализацию, которая (без таких ограничений) будет естественным образом представлять нулевые указатели специальными, отличными от нуля значениями, особенно когда эти значения автоматически будут вызывать специальные аппаратные прерывания, связанные с неверным доступом.

Кроме того, что это требование даст на практике? Понимание нулевых указателей не требует знаний о том, нулевое или ненулевое их внутреннее представление. Предположение о том, что внутреннее представление нулевое, не приводит к упрощению кода (за исключением некоторых случаем сомнительного использования **calloc**. Знание того, что внутреннее представление равно нулю, не упростит вызовы функций, так как размер указателя может быть отличным от размера указателя на **int**. (Если вместо «**0**» для обозначения нулевого указателя использовать «**nil**», необходимость в нулевом внутреннем представлении нулевых указателей даже бы не возникла).

Ну а если честно, на какой-нибудь реальной машине используются ненулевые внутренние представления нулевых указателей или разные представления для указателей разных типов?

Серия Prime 50 использует сегмент 07777, смещение 0 для нулевого указателя, по крайней мере, для PL/I. Более поздние модели используют сегмент **0**, смещение **0** для нулевых указателей Си, что делает необходимыми новые инструкции, такие как TCNP (проверить нулевой указатель Си), которые вводятся для совместимости с уцелевшими скверно написанными С программами, основанными на неверных предположениях. Старые машины Prime с адресацией слов были печально знамениты тем, что указатели на байты (**char ***) у них были большего размера, чем указатели на слова (**int ***).

Серия Eclipse MV корпорации Data General имеет три аппаратно поддерживаемых типа указателей (указатели на слово, байт и бит), два из которых — **char *** и **void *** используются компиляторами Си. Указатель **word *** используется во всех других случаях.

Некоторые центральные процессоры Honeywell-Bull используют код 06000 для внутреннего представления нулевых указателей.

Серия CDC Cyber 180 использует 48-битные указатели, состоящие из кольца (ring), сегмента и смещения. Большинство пользователей имеют в качестве нулевых указателей код 0xB00000000000.

Символическая Лисп-машина с теговой архитектурой даже не имеет общеупотребительных указателей; она использует пару <**NIL,0**> (вообще говоря, несуществующий <объект, смещение> хендл) как нулевой указатель Си.

В зависимости от модели памяти, процессоры 80*86 (PC) могут использовать либо 16-битные указатели на данные и 32-битные указатели на функции, либо, наоборот, 32-битные указатели на данные и 16-битные — на функции.

Старые модели HP 3000 используют различные схемы адресации для байтов и для слов. Указатели на **char** и на **void**, имеют, следовательно, другое представление, чем указатели на **int** (на структуры и т.п.), даже если адрес одинаков.

Что означает ошибка во время исполнения «null pointer assignment» (запись по нулевому адресу). Как мне ее отследить?

Это сообщение появляется только в системе MS-DOS и означает, что произошла запись либо с помощью неинициализированного, либо нулевого указателя в нулевую область.

Отладчик обычно позволяет установить точку останова при доступе к нулевой области. Если это сделать нельзя, вы можете скопировать около 20 байт из области 0 в другую и периодически проверять, не изменились ли эти данные.

Я слышал, что char a() эквивалентно char *a

Ничего подобного. (То, что вы слышали, касается формальных параметров функций.) Массивы — не указатели. Объявление массива «**char a[6];**» требует определенного места для шести символов, которое будет известно под именем «**a**». То есть, существует место под именем «**a**», в которое могут быть помещены **6** символов. С другой стороны, объявление указателя «**char *p;**» требует места только для самого указателя. Указатель будет известен под именем «**p**» и может указывать на любой символ (или непрерывный массив символов).

Важно понимать, что ссылка типа **x[3]** порождает разный код в зависимости от того, массив **x** или указатель.

В случае выражения **p[3]** компилятор генерирует код, чтобы начать с позиции «p», считывает значение указателя, прибавляет к указателю 3 и, наконец, читает символ, на который указывает указатель.

Что понимается под «эквивалентностью указателей и массивов» в Си?

Большая часть путаницы вокруг указателей в С происходит от непонимания этого утверждения. «Эквивалентность» указателей и массивов не позволяет говорить не только об идентичности, но и о взаимозаменяемости.

«Эквивалентность» относится к следующему ключевому определению: значение типа массив **T**, которое появляется в выражении, превращается (за исключением трех случаев) в указатель на первый элемент массива; тип результирующего указателя — указатель на **T**. (Исключение составляют случаи, когда массив оказывается операндом **sizeof**, оператора **&** или инициализатором символьной строки для массива литер.)

Вследствие этого определения нет заметной разницы в поведении оператора индексирования **[]**, если его применять к массивам и указателям. Согласно правилу, приведенному выше, в выражении типа **a[i]** ссылка на массив «**a**» превращается в указатель и дальнейшая индексация происходит так, как будто существует выражение с указателем **p[i]** (хотя доступ к памяти будет различным). В любом случае выражение **x[i]**, где **x** — массив или указатель) равно по определению ***((x)+(i))**.

Почему объявления указателей и массивов взаимозаменяемы в в качестве формальных параметров?

Так как массивы немедленно превращаются в указатели, массив на самом деле не передается в функцию. По общему правилу, любое похожее на массив объявление параметра:

```
f(a)
char a[];
```

рассматривается компилятором как указатель, так что если был передан массив, функция получит:

```
f(a)
char *a;
```

Это превращение происходит только для формальных параметров функций, больше нигде. Если это превращение раздражает вас, избегайте его; многие пришли к выводу, что порождаемая этим путаница перевешивает небольшое преимущество от того, что объявления смотрятся как вызов функции и/или напоминают о том, как параметр будет использоваться внутри функции.

Как массив может быть значением типа lvalue, если нельзя присвоить ему значение?

Стандарт ANSI C определяет «модифицируемое lvalue», но массив к этому не относится.

Почему sizeof неправильно определяет размер массива, который передан функции в качестве параметра?

Оператор **sizeof** сообщает размер указателя, который на самом деле получает функция.

Кто-то объяснил мне, что массивы это на самом деле только постоянные указатели

Это слишком большое упрощение. Имя массива — это константа, следовательно, ему нельзя присвоить значение, но массив — это не указатель.

С практической точки зрения в чем разница между массивами и указателями?

Массивы автоматически резервируют память, но не могут изменить расположение в памяти и размер. Указатель должен быть задан так, чтобы явно указывать на выбранный участок памяти (возможно с помощью **malloc**), но он может быть по нашему желанию переопределен (т.е. будет указывать на другие объекты) и, кроме того, указатель имеет много других применений, кроме службы в качестве базового адреса блоков памяти.

В рамках так называемой эквивалентности массивов и указателей, массивы и указатели часто оказываются взаимозаменяемыми.

Особенно это касается блока памяти, выделенного функцией **malloc**, указатель на который часто используется как настоящий массив.

Я наткнулся на шуточный код, содержащий «выражение» 5["abcdef"]. Почему такие выражения возможны в Си?

Да, индекс и имя массива можно переставлять в Си. Этот забавный факт следует из определения индексации через указатель, а именно, **a[e]** идентично ***((a)+(e))**, для любого выражения **e** и основного выражения **a**, до тех пор пока одно из них будет указателем, а другое целочисленным выражением. Это неожиданная коммутативность часто со странной гордостью упоминается в С-текстах, но за пределами Соревнований по Непонятному Программированию (Obfuscated C Contest)

Мой компилятор ругается, когда я передаю двумерный массив функции, ожидающей указатель на указатель

Правило, по которому массивы превращаются в указатели не может применяться рекурсивно. Массив массивов (т.е. двумерный массив в Си) превращается в указатель на массив, а не в указатель на указатель.

Указатели на массивы могут вводить в заблуждение и применять их нужно с осторожностью. (Путаница еще более усугубляется тем, что существуют некорректные компиляторы, включая некоторые версии pcc и полученные на основе **pcc** программы **lint**, которые неверно воспринимают присваивание многоуровневым указателям многомерных массивов.) Если вы передаете двумерный массив функции:

```
int array[NROWS][NCOLUMNS];
f(array);
```

описание функции должно соответствовать

```
f(int a[][NCOLUMNS]) {...}
```

или

```
f(int (*ap)[NCOLUMNS]) {...} /* ap - указатель на массив */
```

В случае, когда используется первое описание, компилятор неявно осуществляет обычное преобразование «массива массивов» в «указатель на массив»; во втором случае указатель на массив задается явно.

Так как вызываемая функция не выделяет место для массива, нет необходимости знать его размер, так что количество «строк» **NROWS** может быть опущено. «Форма» массива по-прежнему важна, так что размер «столбца» **NCOLUMNS** должен быть включен (а для массивов размерности 3 и больше, все промежуточные размеры).

Если формальный параметр функции описан как указатель на указатель, то передача функции в качестве параметра двумерного массива будет, видимо, некорректной.

Как писать функции, принимающие в качестве параметра двумерные массивы, «ширина» которых во время компиляции неизвестна?

Это непросто. Один из путей — передать указатель на элемент **[0][0]** вместе с размерами и затем симулировать индексацию «вручную»:

```
f2(aryp, nrows, ncolumns)
int *aryp;
int nrows, ncolumns;

{ ... array[i][j] это aryp[i * ncolumns + j] ... }
```

Этой функции массив может быть передан так:

```
f2(&array[0][0], NROWS, NCOLUMNS);
```

Нужно, однако, заметить, что программа,выполняющая индексирование многомерного массива «вручную» не полностью соответствует стандарту ANSI C; поведение (**&array[0][0])[x]** не определено при **x > NCOLUMNS**.

gcc разрешает объявлять локальные массивы, которые имеют размеры, задаваемые аргументами функции, но это — нестандартное расширение.

Как объявить указатель на массив?

Обычно этого делать не нужно. Когда случайно говорят об указателе на массив, обычно имеют в виду указатель на первый элемент массива.

Вместо указателя на массив рассмотрим использование указателя на один из элементов массива. Массивы типа **T** превращаются в указатели типа **T**, что удобно; индексация или увеличение указателя позволяет иметь доступ к отдельным элементам массива. Истинные указатели на массивы при увеличении или индексации указывают на следующий массив и в общем случае если и полезны, то лишь при операциях с массивами массивов.

Если действительно нужно объявить указатель на целый массив, используйте что-то вроде **int (*ap)[N];** где **N** — размер массива. Если размер массива неизвестен, параметр **N** может быть опущен, но получившийся в результате тип «указатель на массив неизвестного размера» — бесполезен.

Исходя из того, что ссылки на массив превращаются в указатели, скажите в чем разница для массива int array(NROWS)(NCOLUMNS); между array и &array?

Согласно ANSI/ISO стандарту Си, **&array** дает указатель типа «**указатель-на-массив-T**», на весь массив.

В языке C до выхода стандарта ANSI оператор **&** в **&array** игнорировался, порождая предупреждение компилятора. Все компиляторы Си, встречая просто имя массива, порождают указатель типа **указатель-на-T**, т.е. на первый элемент массива.

Как динамически выделить память для многомерного массива?

Лучше всего выделить память для массива указателей, а затем инициализировать каждый указатель так, чтобы он указывал на динамически создаваемую строку. Вот пример для двумерного массива:


```
int **array1 = (int **)malloc(nrows * sizeof(int *));
for(i = 0; i < nrows; i++)
array1[i] = (int *)malloc(ncolumns * sizeof(int));
```

(В «реальной» программе, **malloc** должна быть правильно объявлена, а каждое возвращаемое **malloc** значение — проверено.)

Можно поддерживать монолитность массива, (одновременно затрудняя последующий перенос в другое место памяти отдельных строк), с помощью явно заданных арифметических действий с указателями:

```
int **array2 = (int **)malloc(nrows * sizeof(int *));
 array2[0] = (int *)malloc(nrows * ncolumns * sizeof(int));
for(i = 1; i < nrows; i++)
array2[i] = array2[0] + i * ncolumns;
```

В любом случае доступ к элементам динамически задаваемого массива может быть произведен с помощью обычной индексации: **array[i][j]**.

Если двойная косвенная адресация, присутствующая в приведенных выше примерах, вас по каким-то причинам не устраивает, можно имитировать двумерный массив с помощью динамически задаваемого одномерного массива:

```
int *array3 = (int *)malloc(nrows * ncolumns * sizeof(int));
```

Теперь, однако, операции индексирования нужно выполнять вручную, осуществляя доступ к элементу **i,j** с помощью **array3[i*ncolumns+j]**. (Реальные вычисления можно скрыть в макросе, однако вызов макроса требует круглых скобок и запятых, которые не выглядят в точности так, как индексы многомерного массива.)

Наконец, можно использовать указатели на массивы:

```
int (*array4)[NCOLUMNS] =
 (int(*)[NCOLUMNS])malloc(nrows * sizeof(*array4));,
```

но синтаксис становится устрашающим, и «всего лишь» одно измерение должно быть известно во время компиляции.

Пользуясь описанными приемами, необходимо освобождать память, занимаемую массивами (это может проходить в несколько шагов), когда они больше не нужны, и не следует смешивать динамически создаваемые массивы с обычными, статическими.

Как мне равноправно использовать статически и динамически задаваемые многомерные массивы при передаче их в качестве параметров функциям?

Идеального решения не существует. Возьмем объявления

```
int array[NROWS][NCOLUMNS];
int **array1;
int **array2;
int *array3;
int (*array4)[NCOLUMNS];
```

соответствующие способам выделения памяти и функции, объявленные как:

```
f1(int a[][NCOLUMNS], int m, int n);
f2(int *aryp, int nrows, int ncolumns);
f3(int **pp, int m, int n);
```

Тогда следующие вызовы должны работать так, как ожидается:

```
f1(array, NROWS, NCOLUMNS);
f1(array4, nrows, NCOLUMNS);
f2(&array[0][0], NROWS, NCOLUMNS);
f2(*array2, nrows, ncolumns);
f2(array3, nrows, ncolumns);
f2(*array4, nrows, NCOLUMNS);
f3(array1, nrows, ncolumns);
f3(array2, nrows, ncolumns);
```

Следующие два вызова, возможно, будут работать, но они включают сомнительные приведения типов, и работают лишь в том случае, когда динамически задаваемое число столбцов **ncolumns** совпадает с **NCOLUMS**:

```
f1((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1((int (*)[NCOLUMNS])array3, nrows, ncolumns);
```

Необходимо еще раз отметить, что передача **&array[0][0]** функции **f2** не совсем соответствует стандарту.

Если вы способны понять, почему все вышеперечисленные вызовы работают и написаны именно так, а не иначе, и если вы понимаете, почему сочетания, не попавшие в список, работать не будут, то у вас очень хорошее понимание массивов и указателей (и нескольких других областей) Си.

Вот изящный трюк: если я пишу int realarray(10); int *array = &realarray(-1);, то теперь можно рассматривать «array» как массив, у которого индекс первого элемента равен единице

Хотя этот прием внешне привлекателен, он не удовлетворяет стандартам Си. Арифметические действия над указателями определены лишь тогда, когда указатель ссылается на выделенный блок памяти или на воображаемый завершающий элемент, следующий сразу за блоком. В противном случае поведение программы не определено, даже если указатель не переназначается. Код, приведенный выше, плох тем, что при уменьшении смещения может быть получен неверный адрес (возможно, из-за циклического перехода адреса при пересечении границы сегмента).

У меня определен указатель на char, который указывает еще и на int, причем мне необходимо переходить к следующему элементу типа int. Почему ((int *)p)++; не работает?

В языке С оператор преобразования типа не означает «будем действовать так, как будто эти биты имеют другой тип»; это оператор, который действительно выполняет преобразования, причем по определению получается значение типа **rvalue**, которому нельзя присвоить новое значение и к которому не применим оператор ++. (Следует считать аномалией то, что компиляторы **pcc** и расширения **gcc** вообще воспринимают выражения приведенного выше типа.)

Скажите то, что думаете:

```
p = (char *)((int *)p + 1);
```

или просто

```
p += sizeof(int);
```

Могу я использовать void **, чтобы передать функции по ссылке обобщенный указатель?

Стандартного решения не существует, поскольку в С нет общего типа **указатель-на-указатель**. **void *** выступает в роли обобщенного указателя только потому, что автоматически осуществляются преобразования в ту и другую сторону, когда встречаются разные типы указателей. Эти преобразования не могут быть выполнены (истинный тип указателя неизвестен), если осуществляется попытка косвенной адресации, когда **void **** указывает на что-то отличное от **void ***.

Почему не работает фрагмент кода: char *answer; printf("Type something:\n"); gets(answer); printf("You typed \"%s\"\n", answer);

Указатель «**answer**», который передается функции **gets** как место, в котором должны храниться вводимые символы, не инициализирован,

т.е. не указывает на какое-то выделенное место. Иными словами, нельзя сказать, на что указывает «**answer**». (Так как локальные переменные не инициализируются, они вначале обычно содержат «мусор», то есть даже не гарантируется, что в начале «**answer**» — это нулевой указатель.

Простейший способ исправить программу — использовать локальный массив вместо указателя, предоставив компилятору заботу о выделении памяти:

```
#include <string.h>
char answer[100], *p;
printf("Type something:\n");
 fgets(answer, sizeof(answer), stdin);
 if((p = strchr(answer, '\n')) != NULL)
 *p = '\0';
 printf("You typed \"%s\"\n", answer);
```

Заметьте, что в этом примере используется **fgets()** вместо **gets()**, что позволяет указать размер массива, так что выход за пределы массива, когда пользователь введет слишком длинную строку, становится невозможным. (К сожалению, **fgets()** не удаляет автоматически завершающий символ конца строки **\n**, как это делает **gets()**). Для выделения памяти можно также использовать **malloc()**.

Не могу заставить работать strcat. В моей программе char *s1 = "Hello, "; char *s2 = "world!"; char *s3 = strcat(s1, s2); но результаты весьма странные

Проблема снова состоит в том, что не выделено место для результата объединения. С не поддерживает автоматически переменные типа **string**.

Компиляторы С выделяют память только под объекты, явно указанные в исходном тексте (в случае стрингов это может быть массив литер или символы, заключенные в двойные кавычки). Программист должен сам позаботиться о том, чтобы была выделена память для результата, который получается в процессе выполнения программы, например результата объединения строк. Обычно это достигается объявлением массива или вызовом **malloc**.

Функция **strcat** не выделяет память; вторая строка присоединяется к первой. Следовательно, одно из исправлений — в задании первой строки в виде массива достаточной длины:

```
char s1[20] = "Hello, ";
```

Так как **strcat** возвращает указатель на первую строку (в нашем случае **s1**), переменная **s3** — лишняя.

В справочнике о функции strcat сказано, что она использует в качестве аргументов два указателя на char. Откуда мне знать о выделении памяти?

Как правило, при использовании указателей всегда необходимо иметь в виду выделение памяти, по крайней мере, быть уверенным, что компилятор делает это для вас. Если в документации на библиотечную функцию явно ничего не сказано о выделении памяти, то обычно это проблема вызывающей функции.

Краткое описание функции в верхней части страницы справочника в стиле UNIX может ввести в заблуждение. Приведенные там фрагменты кода ближе к определению, необходимому для разработчика функции, чем для того, кто будет эту функцию вызывать. В частности, многие функции, имеющие в качестве параметров указатели (на структуры или стринги, например), обычно вызываются с параметрами, равными адресам каких-то уже существующих объектов (структур или массивов). Другой распространенный пример — функция **stat()**.

Предполагается, что функция, которую я использую, возвращает строку, но после возврата в вызывающую функцию эта строка содержит «мусор»

Убедитесь, что правильно выделена область памяти, указатель на которую возвращает ваша функция. Функция должна возвращать указатель на статически выделенную область памяти или на буфер, передаваемый функции в качестве параметра, или на память, выделенную с помощью **malloc()**, но не на локальный (auto) массив. Другими словами, никогда не делайте ничего похожего на:

```
char *f()
{
  char buf[10];
  /* ... */
  return buf;
}
```

Приведем одну поправку (непригодную в случае, когда **f()** вызывается рекурсивно, или когда одновременно нужны несколько возвращаемых значений):

```
static char buf[10];
```

Почему в некоторых исходных текстах значения, возвращаемые malloc(), аккуратно преобразуются в указатели на выделяемый тип памяти?

До того, как стандарт ANSI/ISO ввел обобщенный тип указателя **void ***, эти преобразования были обычно необходимы для подавления

предупреждений компилятора о приравнивании указателей разных типов. (Согласно стандарту C ANSI/ISO, такие преобразования типов указателей не требуются).

Можно использовать содержимое динамически выделяемой памяти после того как она освобождена?

Нет. Иногда в старых описаниях **malloc()** говорилось, что содержимое освобожденной памяти «остается неизменным»; такого рода поспешная гарантия никогда не была универсальной и не требуется стандартом ANSI.

Немногие программисты стали бы нарочно использовать содержимое освобожденной памяти, но это легко сделать нечаянно. Рассмотрите следующий (корректный) фрагмент программы, в котором освобождается память, занятая односвязным списком:

```
struct list *listp, *nextp;
for(listp = base; listp != NULL; listp = nextp) {
nextp = listp->next;
free((char *)listp);
}
```

и подумайте, что получится, если будет использовано на первый взгляд более очевидное выражение для тела цикла:

```
listp = listp->next
```

без временного указателя **nextp**.

Откуда free() знает, сколько байт освобождать?

Функции **malloc/free** запоминают размер каждого выделяемого и возвращаемого блока, так что не нужно напоминать размер освобождаемого блока.

А могу я узнать действительный размер выделяемого блока?

Нет универсального ответа.

Я выделяю память для структур, которые содержат указатели на другие динамически создаваемые объекты. Когда я освобождаю память, занятую структурой, должен ли я сначала освободить память, занятую подчиненным объектом?

Да. В общем, необходимо сделать так, чтобы каждый указатель, возвращаемый **malloc()** был передан **free()** точно один раз (если память освобождается).

В моей программе сначала с помощью malloc() выделяется память, а затем большое количество памяти освобождается с помощью free(), но количество занятой памяти (так сообщает команда операционной системы) не уменьшается

Большинство реализаций **malloc/free** не возвращают освобожденную память операционной системе (если таковая имеется), а просто делают освобожденную память доступной для будущих вызовов **malloc()** в рамках того же процесса.

Должен ли я освобождать выделенную память перед возвратом в операционную систему?

Делать это не обязательно. Настоящая операционная система восстанавливает состояние памяти по окончании работы программы.

Тем не менее, о некоторых персональных компьютерах известно, что они ненадежны при восстановлении памяти, а из стандарта ANSI/ISO можно лишь получить указание, что эти вопросы относятся к «качеству реализации».

Правильно ли использовать нулевой указатель в качестве первого аргумента функции realloc()? Зачем это нужно?

Это разрешено стандартом ANSI C (можно также использовать **realloc(...,0)** для освобождения памяти), но некоторые ранние реализации C это не поддерживают, и мобильность в этом случае не гарантируется. Передача нулевого указателя **realloc()** может упростить написание самостартующего алгоритма пошагового выделения памяти.

В чем разница между calloc и malloc? Получатся ли в результате применения calloc корректные значения нулевых указателей и чисел с плавающей точкой? Освобождает ли free память, выделенную calloc, или нужно использовать cfree?

По существу **calloc(m,n)** эквивалентна:

```
p = malloc(m * n);
memset(p, 0, m * n);
```

Заполнение нулями означает зануление всех битов, и, следовательно, не гарантирует нулевых значений для указателей и для чисел с плавающей точкой. Функция **free** может (и должна) использоваться для освобождения памяти, выделенной **calloc**.

Что такое alloca и почему использование этой функции обескураживает?

alloca выделяет память, которая автоматически освобождается, когда происходит возврат из функции, в которой вызывалась **alloca**. То

есть, память, выделенная **alloca**, локальна по отношению к «стековому кадру» или контексту данной функции.

Использование **alloca** не может быть мобильным, реализации этой функции трудны на машинах без стека. Использование этой функции проблематично (и очевидная реализация на машинах со стеком не удается), когда возвращаемое ей значение непосредственно передается другой функции, как, например, в **fgets(alloca(100), 100, stdin)**.

По изложенным выше причинам **alloca** (вне зависимости от того, насколько это может быть полезно) нельзя использовать в программах, которые должны быть в высокой степени мобильны.

Почему вот такой код: a(i) = i++; не работает?

Подвыражение **i++** приводит к побочному эффекту — значение **i** изменяется, что приводит к неопределенности, если **i** уже встречается в том же выражении.

Пропустив код int i = 7; printf("%d\n", i++ * i++); через свой компилятор, я получил на выходе 49. А разве, независимо от порядка вычислений, результат не должен быть равен 56?

Хотя при использовании постфиксной формы операторов ++ и -- увеличение и уменьшение выполняется после того как первоначальное значение использовано, тайный смысл слова «после» часто понимается неверно. Не гарантируется, что увеличение или уменьшение будет выполнено немедленно после использования первоначального значения перед тем как будет вычислена любая другая часть выражения. Просто гарантируется, что изменение будет произведено в какой-то момент до окончания вычисления (перед следующей «точкой последовательности» в терминах ANSI C). В приведенном примере компилятор умножил предыдущее значение само на себя и затем дважды увеличил **i** на **1**.

Поведение кода, содержащего многочисленные двусмысленные побочные эффекты не определено. Даже не пытайтесь выяснить, как ваш компилятор все это делает (в противоположность неумным упражнениям во многих книгах по C).

Я экспериментировал с кодом: int i = 2; i = i++; Некоторые компиляторы выдавали i=2, некоторые 3, но один выдал 4. Я знаю, что поведение не определено, но как можно получить 4?

Неопределенное (undefined) поведение означает, что может случиться все что угодно.

Люди твердят, что поведение не определено, а я попробовал ANSI-компилятор и получил то, что ожидал

Компилятор делает все, что ему заблагорассудится, когда встречается с неопределенным поведением (до некоторой степени это относится и к случаю зависимого от реализации и неописанного поведения). В частности, он может делать то, что вы ожидаете. Неблагоразумно, однако, полагаться на это.

Могу я использовать круглые скобки, чтобы обеспечить нужный мне порядок вычислений? Если нет, то разве приоритет операторов не обеспечивает этого?

Круглые скобки, как и приоритет операторов обеспечивают лишь частичный порядок при вычислении выражений. Рассмотрим выражение:

```
f() + g() * h() .
```

Хотя известно, что умножение будет выполнено раньше сложения, нельзя ничего сказать о том, какая из трех функций будет вызвана первой.

Тогда как насчет операторов &&, || и запятой? Я имею в виду код типа if((c = getchar()) == EOF || c == '\n')"

Для этих операторов, как и для оператора **?:** существует специальное исключение; каждый из них подразумевает определенный порядок вычислений, т.е. гарантируется вычисление слева-направо.

Если я не использую значение выражения, то как я должен увеличивать переменную i: так: ++i или так: i++?

Применение той или иной формы сказывается только на значении выражения, обе формы полностью эквивалентны, когда требуются только их побочные эффекты.

Почему неправильно работает код: int a = 1000, b = 1000; long int c = a * b;

Согласно общим правилам преобразования типов языка Си, умножение выполняется с использованием целочисленной арифметики, и результат может привести к переполнению и/или усечен до того как будет присвоен стоящей слева переменной типа **long int**. Используйте явное приведение типов, чтобы включить арифметику длинных целых:

```
long int c = (long int)a * b;
```

Заметьте, что код **(long int)(a * b)** не приведет к желаемому результату.

Что такое стандарт ANSI C?

В 1983 году Американский институт национальных стандартов (ANSI) учредил комитет X3J11, чтобы разработать стандарт языка Си. После длительной и трудной работы, включающей выпуск нескольких публичных отчетов, работа комитета завершилась 14 декабря 1989 г.созданием стандарта ANS X3.159-1989. Стандарт был опубликован весной 1990 г.

В большинстве случаев ANSI C узаконил уже существующую практику и сделал несколько заимствований из C++ (наиболее важное — введение прототипов функций). Была также добавлена поддержка национальных наборов символов (включая подвергшиеся наибольшим нападкам трехзнаковые последовательности). Стандарт ANSI C формализовал также стандартную библиотеку.

Опубликованный стандарт включает «Комментарии» («Rationale»), в которых объясняются многие решения и обсуждаются многие тонкие вопросы, включая несколько затронутых здесь. («Комментарии» не входят в стандарт ANSI X3.159-1989, они приводятся в качестве дополнительной информации.)

Стандарт ANSI был принят в качестве международного стандарта ISO/IEC 9899:1990, хотя нумерация разделов иная (разделы 2–4 стандарта ANSI соответствуют разделам 5–7 стандарта ISO), раздел «Комментарии» не был включен.

Как получить копию Стандарта?

ANSI X3.159 был официально заменен стандартом ISO 9899. Копию стандарта можно получить по адресу:

```
American National Standards Institute
11 W. 42nd St., 13th floor
New York, NY 10036 USA
(+1) 212 642 4900
```

или

```
Global Engineering Documents
2805 McGaw Avenue
Irvine, CA 92714 USA
(+1) 714 261 1455
(800) 854 7179 (U.S. & Canada)
```

В других странах свяжитесь с местным комитетом по стандартам или обратитесь в Национальный Комитет по Стандартам в Женеве:

```
ISO Sales
```



```
Case Postale 56
CH-1211 Geneve 20
 Switzerland
```

Есть ли у кого-нибудь утилиты для перевода C-программ, написанных в старом стиле, в ANSI C и наоборот? Существуют ли программы для автоматического создания прототипов?

Две программы, **protoize** и **unprotoize** осуществляют преобразование в обе стороны между функциями, записанными в новом стиле с прототипами, и функциями, записанными в старом стиле. (Эти программы не поддерживают полный перевод между «классическим» и ANSI C).

Упомянутые программы были сначала вставками в FSF GNU компилятор C, **gcc**, но теперь они — часть дистрибутива **gcc**.

Программа **unproto** — это фильтр, располагающийся между препроцессором и следующим проходом компилятора — на лету переводит большинство особенностей ANSI C в традиционный Си.

GNU пакет GhostScript содержит маленькую программу **ansi2knr**.

Есть несколько генераторов прототипов, многие из них — модификации программы **lint**. Версия 3 программы CPROTO была помещена в конференцию comp.sources.misc в марте 1992 г. Есть другая программа, которая называется «ctxtract».

В заключение хочется спросить: так ли уж нужно преобразовывать огромное количество старых программ в ANSI C? Старый стиль написания функций все еще допустим.

Я пытаюсь использовать ANSI-строкообразующий оператор #, чтобы вставить в сообщение значение символической константы, но вставляется формальный параметр макроса, а не его значение

Необходимо использовать двухшаговую процедуру для того чтобы макрос раскрывался как при строкообразовании:

```
#define str(x) #x
#define xstr(x) str(x)
#define OP plus
char *opname = xstr(OP);
```

Такая процедура устанавливает **opname** равным «**plus**», а не «**OP**».

Такие же обходные маневры необходимы при использовании оператора склеивания лексем **##**, когда нужно соединить значения (а не имена формальных параметров) двух макросов.

Не понимаю, почему нельзя использовать неизменяемые значения при инициализации переменных и задании размеров массивов, как в следующем примере: const int n = 5; int a[n];

Квалификатор **const** означает «только для чтения». Любой объект, квалифицированный как **const**, представляет собой нормальный объект, существующий во время исполнения программы, которому нельзя присвоить другое значение. Следовательно, значение такого объекта — это не константное выражение в полном смысле этого слова. (В этом смысле C не похож на C++). Если есть необходимость в истинных константах, работающих во время компиляции, используйте препроцессорную директиву **#define**.

Какая разница между «char const *p» и «char * const p»?

«**char const *p**» — это указатель на постоянную литеру (ее нельзя изменить); «**char * const p**» — это неизменяемый указатель на переменную (ее можно менять) типа **char**. Зарубите это себе на носу.

Почему нельзя передать char ** функции, ожидающей const char **?

Можно использовать **указатель-на-T** любых типов **T**, когда ожидается **указатель-на-const-T**, но правило (точно определенное исключение из него), разрешающее незначительные отличия в указателях, не может применяться рекурсивно, а только на самом верхнем уровне.

Необходимо использовать точное приведение типов (т.е. в данном случае (**const char ****)) при присвоении или передаче указателей, которые имеют различия на уровне косвенной адресации, отличном от первого.

Мой ANSI компилятор отмечает несовпадение, когда встречается с декларациями: extern int func(float); int func(x) float x; {...

Вы смешали декларацию в новом стиле «**extern int func(float);**» с определением функции в старом стиле «**int func(x) float x;**».

Смешение стилей, как правило, безопасно, но только не в этом случае. Старый C (и ANSI C при отсутствии прототипов и в списках аргументов переменной длины) «расширяет» аргументы определенных типов при передаче их функциям.

Аргументы типа **float** преобразуются в тип **double**, литеры и короткие целые преобразуются в тип **int**. (Если функция определена в старом стиле, параметры автоматически преобразуются в теле функции к менее емким, если таково их описание там.)

Это затруднение может быть преодолено либо с помощью определений в новом стиле:


```
int func(float x) { ... }
```

либо с помощью изменения прототипа в новом стиле таким образом, чтобы он соответствовал определению в старом стиле:

```
extern int func(double);
```

(В этом случае для большей ясности было бы желательно изменить и определение в старом стиле так, чтобы параметр, если только не используется его адрес, был типа **double**.)

Возможно, будет безопасней избегать типов **char**, **short int**, **float** для возвращаемых значений и аргументов функций.

Можно ли смешивать определения функций в старом и новом стиле?

Смешение стилей абсолютно законно, если соблюдается осторожность. Заметьте, однако, что определение функций в старом стиле считается выходящим из употребления, и в один прекрасный момент поддержка старого стиля может быть прекращена.

Почему объявление extern f(struct x {int s;} *p); порождает невнятное предупреждение «struct x introduced in prototype scope» (структура объявлена в зоне видимости прототипа)?

В странном противоречии с обычными правилами для областей видимости структура, объявленная только в прототипе, не может быть совместима с другими структурами, объявленными в этом же файле. Более того, вопреки ожиданиям тег структуры не может быть использован после такого объявления (зона видимости объявления простирается до конца прототипа). Для решения проблемы необходимо, чтобы прототипу предшествовало «пустое» объявление:

```
struct x;
```

которое зарезервирует место в области видимости файла для определения структуры **x**. Определение будет завершено объявлением структуры внутри прототипа.

У меня возникают странные сообщения об ошибках внутри кода, «выключенного» с помощью #ifdef

Согласно ANSI C, текст, «выключенный» с помощью **#if**, **#ifdef** или **#ifndef** должен состоять из «корректных единиц препроцессирования».

Это значит, что не должно быть незакрытых комментариев или кавычек (обратите особое внимание, что апостроф внутри сокращенно записанного слова смотрится как начало литерной константы).

Внутри кавычек не должно быть символов новой строки. Следовательно, комментарии и псевдокод всегда должны находиться между не-

посредственно предназначенными для этого символами начала и конца комментария /* и */.

Могу я объявить main как void, чтобы прекратились раздражающие сообщения «main return no value»? (Я вызываю exit(), так что main ничего не возвращает)

Нет. **main** должна быть объявлена как возвращающая **int** и использующая либо два, либо ни одного аргумента (подходящего типа). Если используется **exit()**, но предупреждающие сообщения не исчезают, вам нужно будет вставить лишний **return**, или использовать, если это возможно, директивы вроде «**notreached**».

Объявление функции как **void** просто не влияет на предупреждения компилятора; кроме того, это может породить другую последовательность вызова/возврата, несовместимую с тем, что ожидает вызывающая функция (в случае **main** это исполняющая система языка Си).

В точности ли эквивалентен возврат статуса с помощью exit(status) возврату с помощью return?

Формально, да, хотя несоответствия возникают в некоторых старых нестандартных системах, в тех случаях, когда данные, локальные для **main()**, могут потребоваться в процессе завершения выполнения (может быть при вызовах **setbuf()** или **atexit()**), или при рекурсивном вызове **main()**.

Почему стандарт ANSI гарантирует только шесть значимых символов (при отсутствии различия между прописными и строчными символами) для внешних идентификаторов?

Проблема в старых компоновщиках, которые не зависят ни от стандарта ANSI, ни от разработчиков компиляторов. Ограничение состоит в том, что только первые шесть символов значимы, а не в том, что длина идентификатора ограничена шестью символами. Это ограничение раздражает, но его нельзя считать невыносимым. В Стандарте оно помечено как «выходящее из употребления», так что в следующих редакциях оно, вероятно, будет ослаблено.

Эту уступку современным компоновщикам, ограничивающим количество значимых символов, обязательно нужно делать, не обращая внимания на бурные протесты некоторых программистов. (В «Комментариях» сказано, что сохранение этого ограничения было «наиболее болезненным».

Если вы не согласны или надеетесь с помощью какого-то трюка заставить компилятор, обремененный ограничивающим количество значимых символов компоновщиком, понимать большее количество этих

символов, читайте превосходно написанный раздел 3.1.2 X3.159 «Комментариев».

Какая разница между memcpy и memmove?

memmove гарантирует правильность операции копирования, если две области памяти перекрываются. memcpy не дает такой гарантии и, следовательно, может быть более эффективно реализована. В случае сомнений лучше применять **memmove**.

Мой компилятор не транслирует простейшие тестовые программы, выдавая всевозможные сообщения об ошибках

Видимо, ваш компилятор разработан до приема стандарта ANSI и поэтому не способен обрабатывать прототипы функций и тому подобное.

Почему не определены некоторые подпрограммы из стандартной ANSI-библиотеки, хотя у меня ANSI совместимый компилятор?

Нет ничего необычного в том, что компилятор, воспринимающий ANSI синтаксис, не имеет ANSI-совместимых головных файлов или стандартных библиотек.

Почему компилятор «Frobozz Magic C», о котором говорится, что он ANSI-совместимый, не транслирует мою программу? Я знаю, что текст подчиняется стандарту ANSI, потому что он транслируется компилятором gcc

Практически все компиляторы (а **gcc** — более других) поддерживают некоторые нестандартные расширения. Уверены ли вы, что отвергнутый текст не применяет одно из таких расширений? Опасно экспериментировать с компилятором для исследования языка. Стандарт может допускать отклонения, а компилятор — работать неверно.

Почему мне не удаются арифметические операции с указателем типа void *?

Потому что компилятору не известен размер объекта, на который указывает **void ***. Перед арифметическими операциями используйте оператор приведения к типу (**char ***) или к тому типу, с которым собираетесь работать.

Правильна ли запись a[3]="abc"? Что это значит?

Эта запись верна в ANSI C (и, возможно, в некоторых более ранних компиляторах), хотя полезность такой записи сомнительна. Объявляется массив размера три, инициализируемый тремя буквами **'a'**, **'b'** и **'c'** без завершающего стринг символа **'\0'**. Массив, следовательно, не может использоваться как стринг функциями **strcpy**, **printf %s** и т.п.

Что такое #pragma и где это может пригодиться?

Директива **#pragma** обеспечивает особую, точно определенную «лазейку» для выполнения зависящих от реализации действий: контроль за листингом, упаковку структур, подавление предупреждающих сообщений (вроде комментариев **/* NOTREACHED */** старой программы **lint**) и т.п.

Что означает «#pragma once»? Я нашел эту директиву в одном из головных файлов

Это расширение, реализованное в некоторых препроцессорах, делает головной файл идемпотентным, т.е. эффект от однократного включения файла равен эффекту от многократного включения. Эта директива приводит к тому же результату, что и прием с использованием **#ifndef**.

Вроде бы существует различие между зависимым от реализации, неописанным(unspecified) и неопределенным (undefined) поведением. В чем эта разница?

Если говорить кратко, то при зависимом от реализации поведении необходимо выбрать один вариант и документировать его. При неописанном поведении также выбирается один из вариантов, но в этом случае нет необходимости это документировать. Неопределенное поведение означает, что может произойти все что угодно. Ни в одном из этих случаев Стандарт не выдвигает требований; в первых двух случаях Стандарт иногда предлагает (а может и требовать) выбор из нескольких близких вариантов поведения.

Если вы заинтересованы в написании мобильных программ, можете игнорировать различия между этими тремя случаями, поскольку всех их необходимо будет избегать.

Как написать макрос для обмена любых двух значений?

На этот вопрос нет хорошего ответа. При обмене целых значений может быть использован хорошо известный трюк с использованием исключающего ИЛИ, но это не сработает для чисел с плавающей точкой или указателей.

Не годится этот прием и в случае, когда оба числа — на самом деле одно и то же число. Из-за многих побочных эффектов не годится и «очевидное» суперкомпактное решение для целых чисел **a^=b^=a^=b**. Когда макрос предназначен для переменных произвольного типа (обычно так и бывает), нельзя использовать временную переменную, поскольку не известен ее тип, а стандартный C не имеет оператора **typeof**.

Если вы не хотите передавать тип переменной третьим параметров, то, возможно, наиболее гибким, универсальным решением будет отказ от использования макроса.

У меня есть старая программа, которая пытается конструировать идентификаторы с помощью макроса #define Paste(a, b) a/**/b, но у меня это не работает

То, что комментарий полностью исчезает, и, следовательно, может быть использован для склеивания соседних лексем (в частности, для создания новых идентификаторов), было недокументированной особенностью некоторых ранних реализаций препроцессора, среди которых заметна была реализация Рейзера (Reiser). Стандарт ANSI, как и K&R, утверждает, что комментарии заменяются единичными пробелами. Но поскольку необходимость склеивания лексем стала очевидной, стандарт ANSI ввел для этого специальный оператор ##, который может быть использован так:

```
#define Paste(a, b) a##b
```

Как наилучшим образом написать cpp макрос, в котором есть несколько инструкций?

Обычно цель состоит в том, чтобы написать макрос, который не отличался бы по виду от функции. Это значит, что завершающая точка с запятой ставится тем, кто вызывает макрос, а в самом теле макроса ее нет.

Тело макроса не может быть просто составной инструкцией, заключенной в фигурные скобки, поскольку возникнут сообщения об ошибке (очевидно, из-за лишней точки с запятой, стоящей после инструкции) в том случае, когда макрос вызывается после **if**, а в инструкции **if/else** имеется **else**-часть.

Обычно эта проблема решается с помощью такого определения:

```
#define Func() do { \
/* объявления */ \
что-то1; \
что-то2; \
/* ... */ \
} while(0) /* (нет завершающей ; ) */
```

Когда при вызове макроса добавляется точка с запятой, это расширение становится простой инструкцией вне зависимости от контекста. (Оптимизирующий компилятор удалит излишние проверки или переходы по условию **0**, хотя **lint** это может и не принять.)

Если требуется макрос, в котором нет деклараций или ветвлений, а все инструкции — простые выражения, то возможен другой подход, когда пишется одно, заключенное в круглые скобки выражение, использующее одну или несколько запятых. Такой подход позволяет также реализовать «возврат» значения).

Можно ли в головной файл с помощью #include включить другой головной файл?

Это вопрос стиля, и здесь возникают большие споры. Многие полагают, что «вложенных с помощью **#include** файлов» следует избегать: авторитетный Indian Hill Style Guide неодобрительно отзывается о таком стиле; становится труднее найти соответствующее определение; вложенные **#include** могут привести к сообщениям о многократном объявлении, если головной файл включен дважды; также затрудняется корректировка управляющего файла для утилиты **Make**. С другой стороны, становится возможным использовать модульный принцип при создании головных файлов (головной файл включает с помощью **#include** то, что необходимо только ему; в противном случае придется каждый раз использовать дополнительный **#include**, что способно вызвать постоянную головную боль); с помощью утилит, подобных **grep** (или файла **tags**) можно легко найти нужные определения вне зависимости от того, где они находятся, наконец, популярный прием:

```
#ifndef HEADERUSED
#define HEADERUSED
...содержимое головного файла...
#endif
```

делает головной файл «идемпотентным», то есть такой файл можно безболезненно включать несколько раз; средства автоматической поддержки файлов для утилиты **Make** (без которых все равно не обойтись в случае больших проектов) легко обнаруживают зависимости при наличии вложенных **#include**.

Работает ли оператор sizeof при использовании средства препроцессора #if?

Нет. Препроцессор работает на ранней стадии компиляции, до того как становятся известны типы переменных. Попробуйте использовать константы, определенные в файле <**limits.h**>, предусмотренном ANSI, или «сконфигурировать» вместо этого командный файл. (А еще лучше написать программу, которая по самой своей природе нечувствительна к размерам переменных).

Можно ли с помощью #if узнать, как организована память машины — по принципу: младший байт — меньший адрес или наоборот?

Видимо, этого сделать нельзя. (Препроцессор использует для внутренних нужд только длинные целые и не имеет понятия об адресации).

А уверены ли вы, что нужно точно знать тип организации памяти? Уж лучше написать программу, которая от этого не зависит.

Во время компиляции мне необходимо сложное препроцесссирование, и я никак не могу придумать, как это сделать с помощью cpp

cpp не задуман как универсальный препроцессор. Чем заставлять **cpp** делать что-то ему не свойственное, подумайте о написании небольшого специализированного препроцессора. Легко раздобыть утилиту типа **make(1)**, которая автоматизирует этот процесс.

Если вы пытаетесь препроцессировать что-то отличное от Си, воспользуйтесь универсальным препроцессором, (таким как **m4**).

Мне попалась программа, в которой, на мой взгляд, слишком много директив препроцессора #ifdef. Как обработать текст, чтобы оставить только один вариант условной компиляции, без использования cpp, а также без раскрытия всех директив #include и #define?

Свободно распространяются программы **unifdef**, **rmifdef** и **scpp**, которые делают в точности то, что вам нужно.

Как получить список предопределенных идентификаторов?

Стандартного способа не существует, хотя необходимость возникает часто. Если руководство по компилятору не содержит этих сведений, то, возможно, самый разумный путь — выделить текстовые строки из исполнимых файлов компилятора или препроцессора с помощью утилиты типа **strings(1)** системы Unix. Имейте в виду, что многие зависящие от системы предопределенные идентификаторы (например, «**unix**») нестандартны (поскольку конфликтуют с именами пользователя) и поэтому такие идентификаторы удаляются или меняются.

Как написать cpp макрос с переменным количеством аргументов?

Популярна такая уловка: определить макрос с одним аргументом, и вызывать его с двумя открывающими и двумя закрывающими круглыми скобками:

```
#define DEBUG(args) (printf("DEBUG: "), printf args)
 if(n != 0) DEBUG(("n is %d\n", n));
```

Очевидный недостаток такого подхода в том, что нужно помнить о дополнительных круглых скобках. Другие решения — использовать различные макросы (DEBUG1, DEBUG2, и т.п.) в зависимости от количества аргументов, или манипулировать запятыми:

```
#define DEBUG(args) (printf("DEBUG: "), printf(args))
#define _ ,
DEBUG("i = %d" _ i)
```

Часто предпочтительнее использовать настоящую функцию, которая стандартным способом может использовать переменное число аргументов.

Как реализовать функцию с переменным числом аргументов?

Используйте головной файл <**stdarg.h**> (или, если необходимо, более старый <**varargs.h**>).

Вот пример функции, которая объединяет произвольное количество стрингов, помещая результат в выделенный с помощью **malloc** участок памяти.

```
#include <stdlib.h> /* для malloc, NULL, size_t */
#include <stdarg.h> /* для va_ макросов */
#include <string.h> /* для strcat и т.п. */

  char *vstrcat(char *first, ...)
  {
 size_t len = 0;
 char *retbuf;
 va_list argp;
 char *p;
 if(first == NULL)
  return NULL;
 len = strlen(first);
  va_start(argp, first);
  while((p = va_arg(argp, char *)) != NULL)
  len += strlen(p);
  va_end(argp);
  retbuf = malloc(len + 1);       /* +1 для \0 */
  if(retbuf == NULL)
  return NULL; /* ошибка */
  (void)strcpy(retbuf, first);
  va_start(argp, first);
  while((p = va_arg(argp, char *)) != NULL)
  (void)strcat(retbuf, p);
  va_end(argp);
```



```
  return retbuf;
 }
```

Вызывается функция примерно так:

```
char *str = vstrcat("Hello, ", "world!", (char *)NULL);
```

Обратите внимание на явное приведение типа в последнем аргументе. (Помните также, что после вызова такой функции нужно освободить память).

Если компилятор разрабатывался до приема стандарта ANSI, перепишите определение функции без прототипа (**"char *vstrcat(first) char *first; {"**) включите <**stdio.h**> вместо <**stdlib.h**>, добавьте **"extern char *malloc();"**, и используйте **int** вместо **size_t**. Возможно, придется удалить приведение (**void**) и использовать **varargs.h** вместо **stdarg**.

Помните, что в прототипах функций со списком аргументов переменной длины не указывается тип аргументов. Это значит, что по умолчанию будет происходить «расширение» типов аргументов.

Это также значит, что тип нулевого указателя должен быть явно указан.

Как написать функцию, которая бы, подобно printf, получала строку формата и переменное число аргументов, а затем для выполнения большей части работы передавала бы все это printf?

Используйте **vprintf**, **vfprintf** или **vsprintf**.

Перед вами подпрограмма «**error**», которая после строки «**error**:» печатает сообщение об ошибке и символ новой строки.

```
#include <stdio.h>
#include <stdarg.h>

  void
  error(char *fmt, ...)
  {
 va_list argp;
 fprintf(stderr, "error: ");
 va_start(argp, fmt);
 vfprintf(stderr, fmt,     argp);
 va_end(argp);
 fprintf(stderr, "\n");
  }
```

Чтобы использовать старый головной файл <**varargs.h**> вместо <**stdarg.h**>, измените заголовок функции:

```
void error(va_alist)
```

```
va_dcl
   {
 char *fmt;
```

измените строку с **va_start**:

```
va_start(argp);
```

и добавьте строку

```
fmt = va_arg(argp, char *);
```

между вызовами **va_start** и **vfprintf**. Заметьте, что после **va_dcl** нет точки с запятой.

Как определить, сколько аргументов передано функции?

Для переносимых программ такая информация недоступна. Некоторые старые системы имели нестандартную функцию **nargs()**, но ее полезность всегда была сомнительна, поскольку обычно эта функция возвращает количество передаваемых машинных слов, а не число аргументов. (Структуры и числа с плавающей точкой обычно передаются в нескольких словах).

Любая функция с переменным числом аргументов должна быть способна по самим аргументам определить их число. Функции типа **printf** определяют число аргументов по спецификаторам формата (%d и т.п.) в строке формата (вот почему эти функции так скверно ведут себя при несовпадении списка аргументов и строки формата). Другой общепринятый прием — использовать признак конца списка (часто это числа **0**, **-1**, или нулевой указатель, приведенный к нужному типу).

Мне не удается добиться того, чтобы макрос va_arg возвращал аргумент типа указатель-на-функцию

Манипуляции с переписыванием типов, которыми обычно занимается **va_arg**, кончаются неудачей, если тип аргумента слишком сложен — вроде указателя на функцию. Если, однако, использовать **typedef** для определения указателя на функцию, то все будет нормально.

Как написать функцию с переменным числом аргументов, которая передает их другой функции с переменным числом аргументов?

В общем случае задача неразрешима. В качестве второй функции нужно использовать такую, которая принимает указатель типа **va_list**, как это делает **vfprintf** в приведенном выше примере.

Если аргументы должны передаваться непосредственно (а не через указатель типа **va_list**), и вторая функция принимает переменное число аргументов (и нет возможности создать альтернативную функцию, принимающую указатель **va_list**), то создание переносимой программы

невозможно. Проблема может быть решена, если обратиться к языку ассемблера соответствующей машины.

Как вызвать функцию со списком аргументов, создаваемым в процессе выполнения?

Не существует способа, который бы гарантировал переносимость. Если у вас пытливый ум, раздобудьте редактор таких списков, в нем есть несколько безрассудных идей, которые можно попробовать...

Переменные какого типа правильнее использовать как булевы? Почему в языке C нет стандартного типа логических переменных? Что использовать для значений true и false — #define или enum?

В языке C нет стандартного типа логических переменных, потому что выбор конкретного типа основывается либо на экономии памяти, либо на выигрыше времени. Такие вопросы лучше решать программисту (использование типа **int** для булевой переменной может быть быстрее, тогда как использование типа **char** экономит память).

Выбор между **#define** и **enum** — личное дело каждого, и споры о том, что лучше, не особенно интересны.

Используйте любой из вариантов:

```
#define TRUE 1 #define YES 1
#define FALSE 0 #define NO 0
enum bool {false, true}; enum bool {no, yes};
```

или последовательно в пределах программы или проекта используйте числа 1 и 0. (Возможно, задание булевых переменных через **enum** предпочтительнее, если используемый вами отладчик раскрывает содержимое enum-переменных).

Некоторые предпочитают такие способы задания:

```
#define TRUE (1==1)
#define FALSE     (!TRUE)
```

или задают «вспомогательный» макрос:

```
#define Istrue(e) ((e) != 0)
```

Не видно, что они этим выигрывают.

Разве не опасно задавать значение TRUE как 1, ведь в C любое не равное нулю значение рассматривается как истинное? А если оператор сравнения или встроенный булев оператор возвратит нечто, отличное от 1?

Истинно (да-да!), что любое ненулевое значение рассматривается в C как значение «ИСТИНА», но это применимо только «на входе», где

ожидается булева переменная. Когда булева переменная генерируется встроенным оператором, гарантируется, что она равна 0 или 1.

Следовательно, сравнение

```
if((a == b) == TRUE)
```

как ни смешно оно выглядит, будет вести себя, как ожидается, если значению **TRUE** соответствует 1. Как правило, явные проверки на **TRUE** и **FALSE** нежелательны, поскольку некоторые библиотечные функции (стоит упомянуть **isupper**, **isalpha** и т.п.), возвращают в случае успеха ненулевое значение, которое не обязательно равно 1. (Кроме того, если вы верите, что «**if((a == b) == TRUE)**» лучше чем «**if(a == b)**», то почему не пойти дальше и не написать:

```
if(((a == b) == TRUE) == TRUE)
```

Хорошее «пальцевое» правило состоит в том, чтобы использовать **TRUE** и **FALSE** (или нечто подобное) только когда булевым переменным или аргументам функции присваиваются значения или когда значение возвращается булевой функцией, но никогда при сравнении.

Макроопределения препроцессора **TRUE** и **FALSE** используются для большей наглядности, а не потому, что конкретные значения могут измениться.

Какова разница между enum и рядом директив препроцессора #define?

В настоящее время разница невелика. Хотя многие, возможно, предпочли бы иное решение, стандарт ANSI утверждает, что произвольному числу элементов перечисления могут быть явно присвоены целочисленные значения. (Запрет на присвоение значений без явного приведения типов, позволил бы при разумном использовании перечислений избежать некоторых ошибок.)

Некоторые преимущества перечислений в том, что конкретные значения задаются автоматически, что отладчик может представлять значения перечислимых переменных в символьном виде, а также в том, что перечислимые переменные подчиняются обычным правилам областей действия. (Компилятор может также выдавать предупреждения, когда перечисления необдуманно смешиваются с целочисленными переменными.

Такое смешение может рассматриваться как проявление плохого стиля, хотя формально это не запрещено). Недостаток перечислений в том, что у программиста мало возможностей управлять размером переменных (и предупреждениями компилятора тоже).

Я слышал, что структуры можно копировать как целое, что они могут быть переданы функциям и возвращены ими, но в K&R I сказано, что этого делать нельзя

В K&R I сказано лишь, что ограничения на операции со структурами будут сняты в следующих версиях компилятора; эти операции уже были возможны в компиляторе Денниса Ритчи, когда издавалась книга K&R I.

Хотя некоторые старые компиляторы не поддерживают копирование структур, все современные компиляторы имеют такую возможность, предусмотренную стандартом ANSI C, так что не должно быть колебаний при копировании и передаче структур функциям.

Каков механизм передачи и возврата структур?

Структура, передаваемая функции как параметр, обычно целиком размещается на стеке, используя необходимое количество машинных слов. (Часто для снижения ненужных затрат программисты предпочитают передавать функции указатель на структуру вместо самой структуры).

Структуры часто возвращаются функциями в ту область памяти, на которую указывает дополнительный поддерживаемый компилятором «скрытый» аргумент. Некоторые старые компиляторы используют для возврата структур фиксированную область памяти, хотя это делает невозможным рекурсивный вызов такой функции, что противоречит стандарту ANSI.

Эта программа работает правильно, но после завершения выдает дамп оперативной памяти. Почему? struct list { char *item; struct list *next; } /* Здесь функция main */ main(argc, argv) ...

Из-за пропущенной точки с запятой компилятор считает, что **main** возвращает структуру. (Связь структуры с функцией **main** трудно определить, мешает комментарий). Так как для возврата структур компилятор обычно использует в качестве скрытого параметра указатель, код, сгенерированный для **main()** пытается принять три аргумента, хотя передаются (в данном случае стартовым кодом Си) только два.

Почему нельзя сравнивать структуры?

Не существует разумного способа сделать сравнение структур совместимым с низкоуровневой природой языка Си. Побайтовое сравнение может быть неверным из-за случайных бит в неиспользуемых «дырках» (такое заполнение необходимо, чтобы сохранить выравнивание для последующих полей). Почленное сравнение потребовало бы неприемлемого количества повторяющихся машинных инструкций в случае больших структур.

Если необходимо сравнить две структуры, напишите для этого свою собственную функцию. C++ позволит создать оператор ==, чтобы связать его с вашей функцией.

Как читать/писать структуры из файла/в файл?

Писать структуры в файл можно непосредственно с помощью **fwrite**:

```
fwrite((char *)&somestruct, sizeof(somestruct), 1, fp);
```

а соответствующий вызов **fread** прочитает структуру из файла.

Однако файлы, записанные таким образом будут не особенно переносимы. Заметьте также, что на многих системах нужно использовать в функции **fopen** флаг «**b**».

Мне попалась программа, в которой структура определяется так: struct name { int namelen; char name(1); }; затем идут хитрые манипуляции с памятью, чтобы массив name вел себя будто в нем несколько элементов. Такие манипуляции законны/мобильны?

Такой прием популярен, хотя Деннис Ритчи назвал это «слишком фамильярным обращением с реализацией Си». ANSI полагает, что выход за пределы объявленного размера члена структуры не полностью соответствует стандарту, хотя детальное обсуждение всех связанных с этим проблем не входит в задачу данных вопросов и ответов. Похоже, однако, что описанный прием будет одинаково хорошо принят всеми известными реализациями Си. (Компиляторы, тщательно проверяющие границы массивов, могут выдать предупреждения). Для страховки будет лучше объявить переменную очень большого размера чем очень малого. В нашем случае

```
...
 char name[MAXSIZE];
...
```

где **MAXSIZE** больше, чем длина любого имени, которое будет сохранено в массиве **name[]**. (Есть мнение, что такая модификация будет соответствовать Стандарту).

Как определить смещение члена структуры в байтах?

Если это возможно, необходимо использовать макрос **offsetof**, который определен стандартом ANSI. Если макрос отсутствует, предлагается такая (не на все 100% мобильная) его реализация:

```
#define offsetof(type, mem) ((size_t) \
((char *)&((type *) 0)->mem - (char *)((type *) 0)))
```


Для некоторых компиляторов использование этого макроса может оказаться незаконным.

Как осуществить доступ к членам структур по их именам во время выполнения программы?

Создайте таблицу имен и смещений, используя макрос **offsetof()**.

Смещение члена структуры **b** в структуре типа **a** равно:

```
offsetb = offsetof(struct a, b)
```

Если **structp** указывает на начало структуры, а **b** — член структуры типа **int**, смещение которого получено выше, **b** может быть установлен косвенно с помощью:

```
*(int *)((char *)structp + offsetb) = value;
```

Почему sizeof выдает больший размер структурного типа, чем я ожидал, как будто в конце структуры лишние символы?

Это происходит (возможны также внутренние «дыры», когда необходимо выравнивание при задании массива непрерывных структур.

Мой компилятор оставляет дыры в структурах, что приводит к потере памяти и препятствует «двоичному» вводу/выводу при работе с внешними файлами. Могу я отключить «дырообразование» или как-то контролировать выравнивание?

В вашем компиляторе, возможно, есть расширение, (например, **#pragma**), которое позволит это сделать, но стандартного способа не существует.

Можно ли задавать начальные значения объединений?

Стандарт ANSI допускает инициализацию первого члена объединения.

Не существует стандартного способа инициализации других членов (и тем более нет такого способа для старых компиляторов, которые вообще не поддерживают какой-либо инициализации).

Как передать функции структуру, у которой все члены — константы?

Поскольку в языке C нет возможности создавать безымянные значения структурного типа, необходимо создать временную структуру.

Какой тип целочисленной переменной использовать?

Если могут потребоваться большие числа, (больше 32767 или меньше -32767), используйте тип **long**. Если нет, и важна экономия памяти (большие массивы или много структур), используйте **short**. Во всех ос-

тальных случаях используйте **int**. Если важно точно определить момент переполнения и/или знак числа не имеет значения, используйте соответствующий тип **unsigned**. (Но будьте внимательны при совместном использовании типов **signed** и **unsigned** в выражениях). Похожие соображения применимы при выборе между **float** и **double**.

Хотя тип **char** или **unsigned char** может использоваться как целочисленный тип наименьшего размера, от этого больше вреда, чем пользы из-за непредсказуемых перемен знака и возрастающего размера программы.

Эти правила, очевидно, не применимы к адресам переменных, поскольку адрес должен иметь совершенно определенный тип.

Если необходимо объявить переменную определенного размера, (единственной причиной тут может быть попытка удовлетворить внешним требованиям к организации памяти), непременно изолируйте объявление соответствующим **typedef**.

Каким должен быть новый 64-битный тип на новых 64-битных машинах?

Некоторые поставщики С компиляторов для 64-битных машин поддерживают тип **long int** длиной 64 бита. Другие же, опасаясь, что слишком многие уже написанные программы зависят от **sizeof(int) == sizeof(long) == 32 бита**, вводят новый 64-битный тип **long long** (или **__longlong**).

Программисты, желающие писать мобильные программы, должны, следовательно, изолировать 64-битные типы с помощью средства **typedef**.

Разработчики компиляторов, чувствующие необходимость ввести новый целочисленный тип большего размера, должны объявить его как «имеющий по крайней мере 64 бит» (это действительно новый тип, которого нет в традиционном Си), а не как «имеющий точно 64 бит».

У меня совсем не получается определение связанного списка. Я пишу: typedef struct { char *item; NODEPTR next; } *NODEPTR; но компилятор выдает сообщение об ошибке. Может структура в С содержать ссылку на себя?

Структуры в Си, конечно же, могут содержать указатели на себя. В приведенном тексте проблема состоит в том, что определение **NODEPTR** не закончено в том месте, где объявляется член структуры «**next**». Для исправления, снабдите сначала структуру тегом («**struct node**»). Далее объявите «**next**» как «**struct node *next;**», и/или поместите декларацию **typedef** целиком до или целиком после объявления структуры.

Одно из возможных решений будет таким:

```
struct node
 {
 char *item;
 struct node *next;
 };
 typedef struct node *NODEPTR;
```

Есть по крайней мере три других одинаково правильных способа сделать то же самое.

Сходная проблема, которая решается примерно так же, может возникнуть при попытке определить с помощью средства **typedef** пару ссылающихся друг на друга структур.

Как объявить массив из N указателей на функции, возвращающие указатели на функции возвращающие указатели на char?

Есть по крайней мере три варианта ответа:

1.

```
char *(*(*a[N])())();
```

2. Писать декларации по шагам, используя **typedef**:

```
typedef char *pc; /* указатель на char */
typedef pc fpc(); /* функция,возвращающая указатель на char */
typedef fpc *pfpc; /* указатель на.. см. выше */
typedef pfpc fpfpc(); /* функция, возвращающая... */
typedef fpfpc *pfpfpc; /* указатель на... */
pfpfpc a[N];  /* массив... */
```

3. Использовать программу **cdecl,** которая переводит с английского на С и наоборот.

```
cdecl> declare a as array of pointer to function returning
pointer to function returning pointer to char
char *(*(*a[])())()
```

cdecl может также объяснить сложные декларации, помочь при явном приведении типов, и, для случая сложных деклараций, вроде только что разобранного, показать набор круглых скобок, в которые заключены аргументы. Версии **cdecl** можно найти в comp.sources.unix и в K&R II.

Я моделирую Марковский процесс с конечным числом состояний, и у меня есть набор функций для каждого состояния. Я хочу, чтобы смена состояний происходила путем возврата функцией указателя на функцию, соответствующую следующему состоянию. Однако, я обнаружил ограничение в механизме деклараций языка Си: нет возможности объявить функцию, возвращающую указатель на функцию, возвращающую указатель на функцию, возвращающую указатель на функцию...

Да, непосредственно это сделать нельзя. Пусть функция возвращает обобщенный указатель на функцию, к которому перед вызовом функции будет применен оператор приведения типа, или пусть она возвращает структуру, содержащую только указатель на функцию, возвращающую эту структуру.

Мой компилятор выдает сообщение о неверной повторной декларации, хотя я только раз определил функцию и только раз вызвал

Подразумевается, что функции, вызываемые без декларации в области видимости (или до такой декларации), возвращают значение типа **int**.

Это приведет к противоречию, если впоследствии функция декларирована иначе. Если функция возвращает нецелое значение, она должна быть объявлена до того как будет вызвана.

Как наилучшим образом декларировать и определить глобальные переменные?

Прежде всего заметим, что хотя может быть много деклараций (и во многих файлах) одной «глобальной» (строго говоря «внешней») переменной, (или функции), должно быть всего одно определение. (Определение — это такая декларация, при которой действительно выделяется память для переменной, и присваивается, если нужно, начальное значение). Лучше всего поместить определение в какой-то главный (для программы или ее части). с файл, с внешней декларацией в головном файле **.h**, который при необходимости подключается с помощью **#include**. Файл, в котором находится определение переменной, также должен включать головной файл с внешней декларацией, чтобы компилятор мог проверить соответствие декларации и определения.

Это правило обеспечивает высокую мобильность программ и находится в согласии с требованиями стандарта ANSI C. Заметьте, что многие компиляторы и компоновщики в системе UNIX используют «общую модель», которая разрешает многократные определения без инициализации.

Некоторые весьма странные компиляторы могут требовать явной инициализации, чтобы отличить определение от внешней декларации.

С помощью препроцессорного трюка можно устроить так, что декларация будет сделана лишь однажды, в головном файле, и она с помощью **#define** «превратится» в определение точно при одном включении головного файла.

Что означает ключевое слово extern при декларации функции?

Слово **extern** при декларации функции может быть использовано из соображений хорошего стиля для указания на то, что определение функции, возможно, находится в другом файле. Формально между:

```
extern int f();
```

и

```
int f();
```

нет никакой разницы.

Я, наконец, понял, как объявлять указатели на функции, но как их инициализировать?

Используйте нечто такое:

```
extern int func();
int (*fp)() = func;
```

Когда имя функции появляется в выражении, но функция не вызывается (то есть, за именем функции не следует "("), оно «сворачивается», как и в случае массивов, в указатель (т.е. неявным образом записанный адрес).

Явное объявление функции обычно необходимо, так как неявного объявления внешней функции в данном случае не происходит (опять-таки из-за того, что за именем функции не следует "(").

Я видел, что функции вызываются с помощью указателей и просто как функции. В чем дело?

По первоначальному замыслу создателя С указатель на функцию должен был «превратиться» в настоящую функцию с помощью оператора * и дополнительной пары круглых скобок для правильной интерпретации.

```
int r, func(), (*fp)() = func;
r = (*fp)();
```

На это можно возразить, что функции всегда вызываются с помощью указателей, но что «настоящие» функции неявно превращаются в указатели (в выражениях, как это происходит при инициализациях) и это

не приводит к каким-то проблемам. Этот довод, широко распространенный компилятором pcc и принятый стандартом ANSI, означает, что выражение:

```
r = fp();
```

работает одинаково правильно, независимо от того, что такое **fp** — функция или указатель на нее. (Имя всегда используется однозначно; просто невозможно сделать что-то другое с указателем на функцию, за которым следует список аргументов, кроме как вызвать функцию.)

Явное задание * безопасно и все еще разрешено (и рекомендуется, если важна совместимость со старыми компиляторами).

Где может пригодиться ключевое слово auto?

Нигде, оно вышло из употребления.

Что плохого в таких строках: char c; while((c = getchar())!= EOF)...

Во-первых, переменная, которой присваивается возвращенное **getchar** значение, должна иметь тип **int. getchar** и может вернуть все возможные значения для символов, в том числе EOF. Если значение, возвращенное **getchar** присваивается переменной типа **char**, возможно либо обычную литеру принять за EOF, либо EOF исказится (особенно если использовать тип **unsigned char**) так, что распознать его будет невозможно.

Как напечатать символ '%' в строке формата printf? Я попробовал \%, но из этого ничего не вышло

Просто удвойте знак процента %%.

Почему не работает scanf("%d",i)?

Для функции **scanf** необходимы адреса переменных, по которым будут записаны данные, нужно написать **scanf("%d", &i);**

Почему не работает double d; scanf("%f", &d);

scanf использует спецификацию формата **%lf** для значений типа **double** и **%f** для значений типа **float**. (Обратите внимание на несходство с **printf**, где в соответствии с правилом расширения типов аргументов спецификация **%f** используется как для **float**, так и для **double**).

Почему фрагмент программы while(!feof(infp)) { fgets(buf, MAXLINE, infp); fputs(buf, outfp); } дважды копирует последнюю строку?

Это вам не Паскаль. Символ EOF появляется только после попытки прочесть, когда функция ввода натыкается на конец файла.

Чаще всего необходимо просто проверять значение, возвращаемое функцией ввода, (в нашем случае **fgets**); в использовании **feof()** обычно вообще нет необходимости.

Почему все против использования gets()?

Потому что нет возможности предотвратить переполнение буфера, куда читаются данные, ведь функции **gets()** нельзя сообщить его размер.

Почему переменной errno присваивается значение ENOTTY после вызова printf()?

Многие реализации стандартной библиотеки ввода/вывода несколько изменяют свое поведение, если стандартное устройство вывода — терминал. Чтобы определить тип устройства, выполняется операция, которая оканчивается неудачно (с сообщением **ENOTTY**), если устройство вывода — не терминал. Хотя вывод завершается успешно, **errno** все же содержит **ENOTTY**.

Запросы моей программы, а также промежуточные результаты не всегда отображаются на экране, особенно когда моя программа передает данные по каналу (pipe) другой программе

Лучше всего явно использовать **fflush(stdout)**, когда непременно нужно видеть то, что выдает программа. Несколько механизмов пытаются «в нужное время» осуществить **fflush**, но, похоже, все это правильно работает в том случае, когда **stdout** — это терминал.

При чтении с клавиатуры функцией scanf возникает чувство, что программа зависает, пока я перевожу строку

Функция **scanf** была задумана для ввода в свободном формате, необходимость в котором возникает редко при чтении с клавиатуры.

Что же касается ответа на вопрос, то символ «**\n**» в форматной строке вовсе не означает, что **scanf** будет ждать перевода строки. Это значит, что **scanf** будет читать и отбрасывать все встретившиеся подряд пробельные литеры (т.е. символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и новой страницы).

Похожее затруднение случается, когда **scanf** «застревает», получив неожиданно для себя нечисловые данные. Из-за подобных проблем часто лучше читать всю строку с помощью **fgets**, а затем использовать **sscanf** или другие функции, работающие со строками, чтобы интерпретировать введенную строку по частям. Если используется **sscanf**, не забудьте проверить возвращаемое значение для уверенности в том, что число прочитанных переменных равно ожидаемому.

Я пытаюсь обновить содержимое файла, для чего использую fopen в режиме «r+», далее читаю строку, затем пишу модифицированную строку в файл, но у меня ничего не получается

Непременно вызовите **fseek** перед записью в файл. Это делается для возврата к началу строки, которую вы хотите переписать; кроме того, всегда необходимо вызвать **fseek** или **fflush** между чтением и записью при чтении/записи в режимах «+». Помните также, что литеры можно заменить лишь точно таким же числом литер.

Как мне отменить ожидаемый ввод, так, чтобы данные, введенные пользователем, не читались при следующем запросе? Поможет ли здесь fflush(stdin)?

fflush определена только для вывода. Поскольку определение «**flush**» («смывать») означает завершение записи символов из буфера (а не отбрасывание их), непрочитанные при вводе символы не будут уничтожены с помощью **fflush**. Не существует стандартного способа игнорировать символы, еще не прочитанные из входного буфера **stdio**. Не видно также, как это вообще можно сделать, поскольку непрочитанные символы могут накапливаться в других, зависящих от операционной системы, буферах.

Как перенаправить stdin или stdout в файл?

Используйте **freopen**.

Если я использовал freopen, то как вернуться назад к stdout (stdin)?

Если необходимо переключаться между **stdin (stdout)** и файлом, наилучшее универсальное решение — не спешить использовать **freopen**.

Попробуйте использовать указатель на файл, которому можно по желанию присвоить то или иное значение, оставляя значение **stdout (stdin)** нетронутым.

Как восстановить имя файла по указателю на открытый файл?

Это проблема, вообще говоря, неразрешима. В случае операционной системы UNIX, например, потребуется поиск по всему диску (который, возможно, потребует специального разрешения), и этот поиск окончится неудачно, если указатель на файл был каналом (pipe) или был связан с удаленным файлом. Кроме того, обманчивый ответ будет получен для файла со множественными связями. Лучше всего самому запоминать имена при открытии файлов (возможно, используя специальные функции, вызываемые до и после **fopen**).

Почему strncpy не всегда завершает строку-результат символом '\0'?

strncpy была задумана для обработки теперь уже устаревших структур данных — «строк» фиксированной длины, не обязательно завершающихся символом '\0'. И, надо сказать, **strncpy** не совсем удобно использовать в других случаях, поскольку часто придется добавлять символ '\0' вручную.

Я пытаюсь сортировать массив строк с помощью qsort, используя для сравнения strcmp, но у меня ничего не получается

Когда вы говорите о «массиве строк», то, видимо, имеете в виду «массив указателей на **char**». Аргументы функции сравнения, работающей в паре с **qsort** — это указатели на сравниваемые объекты, в данном случае — указатели на указатели на **char**. (Конечно, **strcmp** работает просто с указателями на **char**).

Аргументы процедуры сравнения описаны как «обобщенные указатели» **const void *** или **char ***. Они должны быть превращены в то, что они представляют на самом деле, т.е. (**char ****) и дальше нужно раскрыть ссылку с помощью *; тогда **strcmp** получит именно то, что нужно для сравнения. Напишите функцию сравнения примерно так:

```
int pstrcmp(p1, p2) /* сравнить строки, используя указатели */
char *p1, *p2; /* const void * для ANSI C */
 {
  return strcmp(*(char **)p1, *(char **)p2);
 }
```

Сейчас я пытаюсь сортировать массив структур с помощью qsort. Процедура сравнения, которую я использую, принимает в качестве аргументов указатели на структуры, но компилятор выдает сообщение о неверном типе функции сравнения. Как мне преобразовать аргументы функции, чтобы подавить сообщения об ошибке?

Преобразования должны быть сделаны внутри функции сравнения, которая должна быть объявлена как принимающая аргументы типа «обобщенных указателей (**const void *** или **char ***).

Функция сравнения может выглядеть так:

```
int mystructcmp(p1, p2)
char *p1, *p2; /* const void * для ANSI C */
 {
 struct mystruct *sp1 = (struct mystruct *)p1;
 struct mystruct *sp2 = (struct mystruct *)p2;
 /* теперь сравнивайте sp1->что-угодно и sp2-> ... */
```

```
}
```

С другой стороны, если сортируются указатели на структуры, необходима косвенная адресация:

```
sp1 = *(struct mystruct **)p1
```

Как преобразовать числа в строки (операция, противоположная atoi)? Есть ли функция itoa?

Просто используйте **sprintf**. (Необходимо будет выделить память для результата. Беспокоиться, что **sprintf** — слишком сильное средство, которое может привести к перерасходу памяти и увеличению времени выполнения, нет оснований. На практике **sprintf** работает хорошо).

Как получить дату или время в С программе?

Просто используйте функции **time**, **ctime**, и/или **localtime**. (Эти функции существуют многие годы,они включены в стандарт ANSI).

Вот простой пример:

```
#include <stdio.h>
#include <time.h>
main()
{
 time_t now = time((time_t *)NULL);
 printf("It's %.24s.\n", ctime(&now));
 return 0;
 }
```

Я знаю, что библиотечная функция localtime разбивает значение time_t по отдельным членам структуры tm, а функция ctime превращает time_t в строку символов. А как проделать обратную операцию перевода структуры tm или строки символов в значение time_t?

Стандарт ANSI определяет библиотечную функцию **mktime**, которая преобразует структуру **tm** в **time_t**. Если ваш компилятор не поддерживает **mktime**, воспользуйтесь одной из общедоступных версий этой функции.

Перевод строки в значение **time_t** выполнить сложнее из-за большого количества форматов дат и времени, которые должны быть распознаны.

Некоторые компиляторы поддерживают функцию **strptime**; другая популярная функция — **partime** широко распространяется с пакетом RCS, но нет уверенности, что эти функции войдут в Стандарт.

Как прибавить n дней к дате? Как вычислить разность двух дат?

Вошедшие в стандарт ANSI/ISO функции **mktime** и **difftime** могут помочь при решении обеих проблем. **mktime()** поддерживает ненормализованные даты, т.е. можно прямо взять заполненную структуру **tm**, увеличить или уменьшить член **tm_mday**, затем вызвать **mktime()**, чтобы нормализовать члены **year**, **month** и **day** (и преобразовать в значение **time_t**).

difftime() вычисляет разность в секундах между двумя величинами типа **time_t**. **mktime()** можно использовать для вычисления значения **time_t** разности двух дат. (Заметьте, однако, что все эти приемы возможны лишь для дат, которые могут быть представлены значением типа **time_t**; кроме того, из-за переходов на летнее и зимнее время продолжительность дня не точно равна 86400 сек.).

Мне нужен генератор случайных чисел

В стандартной библиотеке С есть функция rand(). Реализация этой функции в вашем компиляторе может не быть идеальной, но и создание лучшей функции может оказаться очень непростым.

Как получить случайные целые числа в определенном диапазоне?

Очевидный способ:

```
rand() % N
```

где **N**, конечно, интервал, довольно плох, ведь поведение младших бит во многих генераторах случайных чисел огорчает своей неслучайностью. Лучше попробуйте нечто вроде:

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

Если вам не нравится употребление чисел с плавающей точкой, попробуйте:

```
rand() / (RAND_MAX / N + 1)
```

Оба метода требуют знания **RAND_MAX** (согласно ANSI, **RAND_MAX** определен в <**stdlib.h**>. Предполагается, что **N** много меньше **RAND_MAX**.

Каждый раз при запуске программы функция rand() выдает одну и ту же последовательность чисел

Можно вызвать **srand()** для случайной инициализации генератора случайных чисел. В качестве аргумента для **srand()** часто используется текущее время, или время, прошедшее до нажатия на клавишу (хотя едва ли существует мобильная процедура определения времен нажатия на клавиши).

Мне необходима случайная величина, имеющая два значения true/false. Я использую rand() % 2, но получается неслучайная последовательность: 0,1,0,1,0...

Некачественные генераторы случайных чисел (попавшие, к несчастью, в состав некоторых компиляторов) не очень то случайны, когда речь идет о младших битах. Попробуйте использовать старшие биты.

Я все время получаю сообщения об ошибках — не определены библиотечные функции, но я включаю все необходимые головные файлы

Иногда (особенно для нестандартных функций) следует явно указывать, какие библиотеки нужны при компоновке программы.

Я по-прежнему получаю сообщения, что библиотечные функции не определены, хотя и использую ключ -l, чтобы явно указать библиотеки во время компоновки

Многие компоновщики делают один проход по списку объектных файлов и библиотек, которые вы указали, извлекая из библиотек только те функции, удовлетворяющие ссылки, которые к этому моменту оказались неопределенными. Следовательно, порядок относительно объектных файлов, в котором перечислены библиотеки, важен; обычно просмотр библиотек нужно делать в самом конце. (Например, в операционной системе UNIX помещайте ключи -l в самом конце командной строки).

Мне необходим исходный текст программы, которая осуществляет поиск заданной строки

Ищите библиотеку **regesp** (поставляется со многими UNIX-системами) или достаньте пакет **regexp** Генри Спенсера (Henry Spencer).

Как разбить командную строку на разделенные пробельными литерами аргументы (что-то вроде argc и argv в main)?

В большинстве компиляторов имеется функция **strtok**, хотя она требует хитроумного обращения, а ее возможности могут вас не удовлетворить (например, работа в случае кавычек).

Вот я написал программу, а она ведет себя странно. Что в ней не так?

Попробуйте сначала запустить **lint** (возможно, с ключами **-a**, **-c**, **-h**, **-p**). Многие компиляторы С выполняют на самом деле только половину задачи, не сообщая о тех подозрительных местах в тексте программы, которые не препятствуют генерации кода.

Как мне подавить сообщение «warning: possible pointer alignment problem» («предупреждение: возможна проблема с выравниванием указателя»), которое выдает lint после каждого вызова malloc?

Проблема состоит в том, что **lint** обычно не знает, и нет возможности ему об этом сообщить, что **malloc** «возвращает указатель на область памяти, которая должным образом выровнена для хранения объекта любого типа». Возможна псевдореализация **malloc** с помощью **#define** внутри **#ifdef lint**, которая удалит это сообщение, но слишком прямолинейное применение **#define** может подавить и другие осмысленные сообщения о действительно некорректных вызовах. Возможно, будет проще игнорировать эти сообщения, может быть, делать это автоматически с помощью **grep -v**.

Где найти ANSI-совместимый lint?

Программа, которая называется **FlexeLint** (в виде исходного текста с удаленными комментариями и переименованными переменными, пригодная для компиляции на «почти любой» системе) может быть заказана по адресу:

```
Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426 USA
(+1) 610 584 4261
gimpel@netaxs.com
```

Lint для System V release 4 ANSI-совместим и может быть получен (вместе с другими С утилитами) от UNIX Support Labs или от дилеров System V.

Другой ANSI-совместимый **LINT** (способный также выполнять формальную верификацию высокого уровня) называется **LCLint** и доступен через: ftp: larch.lcs.mit.edu://pub/Larch/lclint/.

Ничего страшного, если программы **lint** нет. Многие современные компиляторы почти столь же эффективны в выявлении ошибок и подозрительных мест, как и **lint**.

Может ли простой и приятный трюк if(!strcmp(s1, s2)) служить образцом хорошего стиля?

Стиль не особенно хороший, хотя такая конструкция весьма популярна.

Тест удачен в случае равенства строк, хотя по виду условия можно подумать, что это тест на неравенство.

Есть альтернативный прием, связанный с использованием макроса:

```
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
```

Вопросы стиля программирования, как и проблемы веры, могут обсуждаться бесконечно. К хорошему стилю стоит стремиться, он легко узнаваем, но не определим.

Каков наилучший стиль внешнего оформления программы?

Не так важно, чтобы стиль был «идеален». Важнее, чтобы он применялся последовательно и был совместим (со стилем коллег или общедоступных программ).

Так трудно определимое понятие «хороший стиль» включает в себя гораздо больше, чем просто внешнее оформление программы; не тратьте слишком много времени на отступы и скобки в ущерб более существенным слагаемым качества.

У меня операции с плавающей точкой выполняются странно, и на разных машинах получаются различные результаты

Сначала убедитесь, что подключен головной файл <**math.h**> и правильно объявлены другие функции, возвращающие тип **double**.

Если дело не в этом, вспомните, что большинство компьютеров используют форматы с плавающей точкой, которые хотя и похоже, но вовсе не идеально имитируют операции с действительными числами. Потеря значимости, накопление ошибок и другие свойственные ЭВМ особенности вычислений могут быть весьма болезненными.

Не нужно предполагать, что результаты операций с плавающей точкой будут точными, в особенности не стоит проверять на равенство два числа с плавающей точкой. (Следует избегать любых ненужных случайных факторов.)

Все эти проблемы одинаково свойственны как Си, так и другим языкам программирования. Семантика операций с плавающей точкой определяется обычно так, «как это выполняет процессор»; иначе компилятор вынужден бы был заниматься непомерно дорогостоящей эмуляцией «правильной» модели вычислений.

Я пытаюсь проделать кое-какие вычисления, связанные с тригонометрией, включаю <math.h>, но все равно получаю сообщение: «undefined: _sin» во время компиляции

Убедитесь в том, что компоновщику известна библиотека, в которой собраны математические функции. Например, в операционной системе UNIX часто необходим ключ **-lm** в самом конце командной строки.

Почему в языке С нет оператора возведения в степень?

Потому что немногие процессоры имеют такую инструкцию. Вместо этого можно, включив головной файл <**math.h**>, использовать функцию **pow()**, хотя часто при небольших целых порядках явное умножение предпочтительней.

Как округлять числа?

Вот самый простой и честный способ:

```
(int)(x + 0.5)
```

Хотя для отрицательных чисел это не годится.

Как выявить специальное значение IEEE NaN и другие специальные значения?

Многие компиляторы с высококачественной реализацией стандарта **IEEE** операций с плавающей точкой обеспечивают возможность (например, макрос **isnan()**) явной работы с такими значениями, а Numerical C Extensions Group (NCEG) занимается стандартизацией таких средств. Примером грубого, но обычно эффективного способа проверки на **NaN** служит макрос:

```
#define isnan(x) ((x) != (x))
```

хотя не знающие об **IEEE** компиляторы могут выбросить проверку в процессе оптимизации.

У меня проблемы с компилятором Turbo C. Программа аварийно завершается, выдавая нечто вроде «floating point formats not linked»

Некоторые компиляторы для мини-эвм, включая Turbo C (а также компилятор Денниса Ритчи для PDP-11), не включают поддержку операций с плавающей точкой, когда им кажется, что это не понадобится.

В особенности это касается версий **printf** и **scanf**, когда для экономии места не включается поддержка **%e**, **%f** и **%g**. Бывает так, что эвристической процедуры Turbo C, которая определяет — использует программа операции с плавающей точкой или нет, оказывается недостаточно, и программист должен лишний раз вызвать функцию, использующую операции с плавающей точкой, чтобы заставить компилятор включить поддержку таких операций.

Как прочитать с клавиатуры один символ, не дожидаясь новой строки?

Вопреки популярному убеждению и желанию многих, этот вопрос (как и родственные вопросы, связанные с дублированием символов) не относится к языку Си. Передача символов с «клавиатуры» программе, написанной на Си, осуществляется операционной системой, эта опера-

ция не стандартизирована языком Си. Некоторые версии библиотеки **curses** содержат функцию **cbreak()**, которая делает как раз то, что нужно.

Если вы пытаетесь прочитать пароль с клавиатуры без вывода его на экран, попробуйте **getpass()**. В операционной системе UNIX используйте **ioctl** для смены режима работы драйвера терминала (**CBREAK** или **RAW** для «классических» версий; **ICANON**, **c_cc[VMIN]** и **c_cc[VTIME]** для System V или Posix). В системе MS-DOS используйте **getch()**. В системе VMS попробуйте функции управления экраном (**SMG$**) или **curses**, или используйте низкоуровневые команды **$QIO** с кодами **IO$_READVBLK** (и, может быть, **IO$M_NOECHO**) для приема одного символа за раз. В других операционных системах выкручивайтесь сами. Помните, что в некоторых операционных системах сделать нечто подобное невозможно, так как работа с символами осуществляется вспомогательными процессорами и не находится под контролем центрального процессора.

Вопросы, ответы на которые зависят от операционной системы, неуместны в **comp.lang.c**. Ответы на многие вопросы можно найти в FAQ таких групп как **comp.unix.questions** и **comp.os.msdos.programmer**.

Имейте в виду, что ответы могут отличаться даже в случае разных вариантов одной и той же операционной системы. Если вопрос касается специфики операционной системы, помните, что ответ, пригодный в вашей системе, может быть бесполезен всем остальным.

Как определить — есть ли символы для чтения (и если есть, то сколько?) И наоборот, как сделать, чтобы выполнение программы не блокировалось, когда нет символов для чтения?

Ответ на эти вопросы также целиком зависит от операционной системы.

В некоторых версиях **curses** есть функция **nodelay()**. В зависимости от операционной системы вы сможете использовать «неблокирующий ввод/вывод» или системный вызов «**select**» или **ioctl FIONREAD**, или **kbhit()**, или **rdchk()**, или опцию **O_NDELAY** функций **open()** или **fcntl()**.

Как очистить экран? Как выводить на экран негативное изображение?

Это зависит от типа терминала (или дисплея). Можете использовать такую библиотеку как **termcap** или **curses**, или какие-то другие функции, пригодные для данной операционной системы.

Как программа может определить полный путь к месту, из которого она была вызвана?

argv[0] может содержать весь путь, часть его или ничего не содержать.

Если имя файла в **argv[0]** имеется, но информация не полна, возможно повторение логики поиска исполнимого файла, используемой интерпретатором командного языка. Гарантированных или мобильных решений, однако, не существует.

Как процесс может изменить переменную окружения родительского процесса?

В общем, никак. Различные операционные системы обеспечивают сходную с UNIX возможность задания пары имя/значение. Может ли программа с пользой для себя поменять окружение, и если да, то как — все это зависит от операционной системы.

В системе UNIX процесс может модифицировать свое окружение (в некоторых системах есть для этого функции **setenv()** и/или **putenv()**) и модифицированное окружение обычно передается дочерним процессам но не распространяется на родительский процесс.

Как проверить, существует ли файл? Мне необходимо спрашивать пользователя перед тем как переписывать существующие файлы

В UNIX-подобных операционных системах можно попробовать функцию **access()**, хотя имеются кое-какие проблемы. (Применение **access()** может сказаться на последующих действиях, кроме того, возможны особенности исполнения в setuid-программах). Другое (возможно, лучшее) решение — вызвать **stat()**, указав имя файла. Единственный универсальный, гарантирующий мобильность способ состоит в попытке открыть файл.

Как определить размер файла до его чтения?

Если «размер файла» — это количество литер, которое можно прочитать, то, вообще говоря, это количество заранее неизвестно. В операционной системе Unix вызов функции **stat** дает точный ответ, и многие операционные системы поддерживают похожую функцию, которая дает приблизительный ответ. Можно с помощью **fseek** переместиться в конец файла, а затем вызвать **ftell**, но такой прием немобилен (дает точный ответ только в системе Unix, в других же случаях ответ почти точен лишь для определенных стандартом ANSI «двоичных» файлов).

В некоторых системах имеются подпрограммы **filesize** или **filelength**.

И вообще, так ли нужно заранее знать размер файла? Ведь самый точный способ определения его размера в С программе заключается в открытии и чтении. Может быть, можно изменить программу так, что размер файла будет получен в процессе чтения?

Как укоротить файл без уничтожения или переписывания?

В системах BSD есть функция **ftruncate()**, несколько других систем поддерживают **chsize()**, в некоторых имеется (возможно, недокументированный) параметр **fcntl F_FREESP**. В системе MS-DOS можно иногда использовать **write(fd, "", 0)**. Однако, полностью мобильного решения не существует.

Как реализовать задержку или определить время реакции пользователя, чтобы погрешность была меньше секунды?

У этой задачи нет, к несчастью, мобильных решений. Unix V7 и ее производные имели весьма полезную функцию **ftime()** с точностью до миллисекунды, но она исчезла в System V и Posix. Поищите такие функции: **nap()**, **setitimer()**, **msleep()**, **usleep()**, **clock()** и **gettimeofday()**. Вызовы **select()** и **poll()** (если эти функции доступны) могут быть добавлены к сервисным функциям для создания простых задержек. В системе MS-DOS возможно перепрограммирование системного таймера и прерываний таймера.

Как прочитать объектный файл и передать управление на одну из его функций?

Необходим динамический компоновщик и/или загрузчик. Возможно выделить память с помощью malloc и читать объектные файлы, но нужны обширные познания в форматах объектных файлов, модификации адресов и пр.

В системе BSD Unix можно использовать **system()** и **ld -A** для динамической компоновки. Многие (большинство) версии SunOS и System V имеют библиотеку **-ldl**, позволяющую динамически загружать объектные модули. Есть еще GNU пакет, который называется «**dld**».

Как выполнить из программы команду операционной системы?

Используйте **system()**.

Как перехватить то, что выдает команда операционной системы?

Unix и некоторые другие операционные системы имеют функцию **popen()**, которая переназначает поток **stdio** каналу, связанному с процессом, запустившим команду, что позволяет прочитать выходные данные (или передать входные). А можно просто перенаправить выход команды в файл, затем открыть его и прочесть.

Как работать с последовательными (COM) портами?

Это зависит от операционной системы. В системе Unix обычно осуществляются операции открытия, чтения и записи во внешнее устройство и используются возможности терминального драйвера для настройки характеристик. В системе MS-DOS можно либо использовать прерывания BIOSa, либо (если требуется приличная скорость) один из управляемых прерываниями пакетов для работы с последовательными портами.

Что можно с уверенностью сказать о начальных значениях переменных, которые явным образом не инициализированы? Если глобальные переменные имеют нулевое начальное значение, то правильно ли нулевое значение присваивается указателям и переменным с плавающей точкой?

«Статические» переменные (то есть объявленные вне функций и те, что объявлены как принадлежащие классу **static**) всегда инициализируются (прямо при старте программы) нулем, как будто программист написал «**=0**». Значит, переменные будут инициализированы как нулевые указатели (соответствующего типа), если они объявлены указателями, или значениями **0.0**, если были объявлены переменные с плавающей точкой.

Переменные автоматического класса (т.е. локальные переменные без спецификации **static**), если они явно не определены, первоначально содержат «мусор». Никаких полезных предсказаний относительно мусора сделать нельзя.

Память, динамически выделяемая с помощью **malloc** и **realloc** также будет содержать мусор и должна быть инициализирована, если это необходимо, вызывающей программой. Память, выделенная с помощью **calloc**, зануляет все биты, что не всегда годится для указателей или переменных с плавающей точкой.

Этот текст взят прямо из книги, но он не компилируется: f() { char a() = "Hello, world!"; }

Возможно, ваш компилятор создан до принятия стандарта ANSI и еще не поддерживает инициализацию «автоматических агрегатов» (то есть нестатических локальных массивов и структур).

Чтобы выкрутиться из этой ситуации, сделайте массив статическим или глобальным, или инициализируйте его с помощью **strcpy**, когда вызывается **f()**. (Всегда можно инициализировать автоматическую переменную **char *** стрингом литер.)