Мобильное программирование в среде ОС UNIX

Содержание

Мобильное программирование в среде ОС UNIX

Стандартные библиотеки

Библиотека системных вызовов

Библиотека ввода/вывода

Дополнительные библиотеки

Файлы заголовков

Мобильность на уровне исходных текстов

Особенности мобильного программирования на языке Си

Обеспечение независимости от особенностей версии ОС UNIX

Бинарная совместимость

Возможности достижения бинарной совместимости

Преимущества и ограничения

### Стандартные библиотеки

Очевидным требованием к операционной среде, поддерживающей мобильное прикладное программирование, является то, что все функции, предоставляемые ею прикладной программе, должны быть четко специфицированы и должны точно соответствовать этим спецификациям в любой реализации операционной среды. В UNIX-ориентированных средах это требование удовлетворяется за счет наличия нескольких стандартизованных библиотек функций и соответствующих наборов файлов заголовков (header-файлов).

#### Библиотека системных вызовов

Базовой библиотекой любого варианта системы UNIX является библиотека системных вызовов. Сейчас невозможно найти два варианта ОС UNIX с разными названиями, наборы системных вызовов которых полностью бы совпадали. Однако, любой такой вариант поддерживает системные вызовы, которые специфицированы в стандартах, упоминаемых в разделе 7.5. К полностью стандартным системным вызовам относятся системные вызовы для работы с файлами (включая специальные файлы), системные вызовы для управления процессами (fork и семейство exec), системные вызовы класса IPC (хотя, как мы упоминали в п. 3.5.4, в UNIX System V механизм программных каналов реализован не в виде набора системных вызовов ядра ОС, а как набор библиотечных функций над пакетом TLI). Приведенное в скобках замечание на самом деле является очень важным. Пользователя, стремящегося создать мобильное приложение с использованием системных вызовов, не должны волновать детали реализации. Важно, чтобы состав системных вызовов, их интерфейсы и семантика соответствовали стандартам.

Теперь мы можем сформулировать правило прикладного мобильного программирования с использованием системных вызовов:

Проектируя и разрабатывая прикладную систему, убедитесь, что вы не используете системные вызовы, не входящие в стандарт.

Придерживаясь этого правила, с большой вероятностью вы не будете иметь проблем с переносом программы в среду другого варианта ОС UNIX по причине несовместимости наборов системных вызовов.

#### Библиотека ввода/вывода

Традиционной для ОС UNIX библиотекой функций более высокого уровня, чем библиотека системных вызовов, является, так называемая, стандартная библиотека ввода/вывода (stdio). Основной набор функций этой библиотеки служит для выполнения файловых операций с буферизацией данных в памяти пользовательского процесса. Библиотека ввода/вывода фактически стандартизована очень давно, и ей можно безопасно пользоваться в любой операционной среде. В частности, единообразные библиотеки ввода/вывода поддерживаются во всех современных реализациях системы программирования языка Си, выполненных не в среде ОС UNIX (включая реализации в среде MS-DOS).

Поэтому можно сформулировать правило мобильного программирования с использованием библиотеки ввода/вывода:

Если для разрабатываемой вами прикладной программы достаточно возможностей библиотеки ввода/вывода, ограничьтесь использованием этой библиотеки.

Придерживаясь этого правила, с большой вероятностью вы не будете иметь проблем, связанных с вводом/выводом, при переносе вашей программы в любую операционную среду (не обязательно UNIX-ориентированную), в которой поддерживается стандартная библиотека ввода/вывода.

#### Дополнительные библиотеки

Понятно, что при прикладном программировании используются не только библиотеки системных вызовов и ввода/вывода. Существует масса других библиотечных функций, предназначенных, например, для разнообразных преобразований форматов данных, математических вычислений и т.д. К таким библиотекам нужно относиться очень осторожно, поскольку в целях повышения эффективности соответствующие функции могут быть машинно-зависимыми и по этой причине обладать специфическими интерфейсами (хотя, скорее всего, не зависят от особенностей операционной системы). Сама по себе машинная зависимость библиотечной функции не представляет опасности, поскольку при переносе программы на компьютер с другой архитектурой все равно потребуется перекомпиляция и перекомпоновка прикладной программы, но специфичность интерфейсов может причинить большие неприятности.

Наиболее безопасным решением на сегодняшний день (при программировании на языке Си) является использование библиотек, специфицированных в стандарте языка Си. Наверное, стандартных библиотек Си окажется недостаточно в случае сложных приложений, но если при указании опции "ANSI" ваша система программирования успешно производит сборку выполняемой программы, можно быть почти уверенным, что вы не будете иметь проблем при переносе программы на компьютер, на котором установлен компилятор стандартного языка Си.

Поэтому можно сформулировать правило мобильного программирования с использованием дополнительных библиотек:

Если для разрабатываемой вами прикладной системы оказывается достаточным использование библиотек, специфицированных в стандарте языка Си, ограничьтесь использованием этих библиотек.

Если стандартных библиотек оказывается недостаточно и приходится использовать функцию из некоторой дополнительной библиотеки, поддерживаемой в вашей системе, постарайтесь проверить, насколько она стандартна. Если вы не уверены в стандартности используемой функции, то лучше напишите собственную интерфейсную функцию с известным вам интерфейсом, а при переносе прикладной программы состыкуйте эту функцию (может быть, придется ее переписать) с подходящей библиотечной функцией целевой системы (однако нет гарантии, что вам удастся ее найти).

#### Файлы заголовков

Использование текстовых файлов заголовков (header-файлов), которые вставляются в текст программы на языке Си с помощью директивы include препроцессора Си, является традиционной техникой программирования на языке Си в среде ОС UNIX, обеспечивающей синтаксическую правильность использования библиотечных функций (в том числе и системных вызовов) в прикладной программе. Ранее файлы заголовков, главным образом, содержали определения типов и символических констант (символические константы - это константы, которым сопоставлены имена посредством директивы define препроцессора Си), используемых в интерфейсах соответствующих библиотечных функций. Корректное применение файлов заголовков позволяло программистам не заботиться о правильности типов данных, используемых при обращении к библиотечным функциям и обработке их результатов.

Однако, традиционные файлы заголовков не гарантировали того, что набор параметров вызываемой библиотечной функции соответствовал ее интерфейсу, поскольку объявление функции, содержащее ее интерфейс, в файле компиляции отсутствовало. В лучшем случае ошибки такого рода устойчиво проявлялись во время выполнения программы, хотя далеко не всегда было просто понять их природу. В худшем случае ошибка возникала при переносе программы, поскольку одноименные библиотечные функции действительно обладали разными интерфейсами в разных средах, и в исходной операционной среде ошибки в параметрах не было.

Эту проблему удалось решить (хотя и не абсолютно) за счет введения в язык Си понятия прототипа функции. Грубо говоря, прототип функции - это часть ее объявления, содержащая только интерфейс (без тела функции). Наличие прототипа любой функции допускается в любом файле компиляции, даже не обязательно содержащем вызов этой функции. Однако, если вызов функции содержится в файле компиляции, то набор параметров вызова должен точно соответствовать интерфейсу вызываемой функции, определенному в ее прототипе.

Дальнейший ход рассуждений очевиден. Для группы родственных библиотечных функций делается общий файл заголовков, содержащий необходимые определения типов данных и символических констант, а также набор прототипов этих библиотечных функций. После включения в файл компиляции такого файла заголовков на стадии компиляции будут обнаружены все синтаксические ошибки обращения к библиотечным функциям. В предыдущем параграфе мы отметили, что это решение не абсолютно. Это действительно так, поскольку в принципе никто не может заставить программиста на языке Си включать в текст программы все требуемые файлы заголовков. Однако, такова специфика мира программирования: каждый волен усложнять свою жизнь в такой степени, в которой ему или ей это нравится.

Последнее замечание относительно файлов заголовков. В последнее время они содержат большое количество операторов условной компиляции, относящихся большей частью к определению символических констант. Дело в том, что в зависимости от версии операционной системы (мы имеем в виду версии одной линии ОС UNIX, например, UNIX System V) значения констант, используемых с одним и тем же смыслом, часто меняются. Конечно, прикладная программа не должна зависеть от таких изменений. Наличие операторов условной компиляции внутри файла заголовков разрешает эту проблему.

Поэтому последнее правило этого раздела можно сформулировать следующим образом:

При программировании на языке Си с использованием библиотечных функций используйте все требуемые файлы заголовков. Это поможет быстрее найти ошибки и повысит мобильность прикладной программы.

### Мобильность на уровне исходных текстов

Материал, рассмотренный нами в предыдущем разделе, относится к вопросам мобильного программирования в связи с использованием функций операционной среды. Однако, если говорить о переносимости программ между компьютерами с разной архитектурой, имея в виду использование языка Си (не слишком высокого уровня), то нужно учитывать ряд требований, которым должна удовлетворять программа.

#### Особенности мобильного программирования на языке Си

Особая роль языка программирования Си состоит в том, что он, с одной стороны, позволяет писать для UNIX-систем практически столь же эффективный код, что и языки ассемблера, а с другой, является основным средством переноса программ между UNIX-системами. Можно сказать, что Си является машинно-независимым языком ассемблера для UNIX-систем. Это делает его основным средством написания эффективных и переносимых программ для этого класса вычислительных систем. Стандартизация языка сначала Американским национальным институтом стандартов (ANSI), а затем и Международной организацией по стандартам (ISO) закрепила эту роль, распространив ее и на персональные компьютеры. Будем ссылаться на версию языка Си, определенную стандартом, как на язык ANSI C.

Сказанное не означает, что любая программа, написанная на ANSI C и отлаженная в одной вычислительной системе (ВС), безусловно переносима на любую другую вычислительную систему, также имеющую компилятор языка Си, отвечающий требованиям ANSI. Однако, язык ANSI C определен таким образом, чтобы можно было писать программы, подвергающиеся минимальным изменениям при их переносе на другие вычислительные системы.

Программа на ANSI C переносима из исходной ВС в целевую, если она успешно компилируется в целевой ВС и ее работа функционально эквивалентна работе в исходной ВС.

На переносимость программы влияют особенности как аппаратного, так и программного окружения языка в исходной и в целевой ВС. Можно выделить четыре фактора, влияющих на переносимость программы:

* архитектура вычислительных систем;
* метрические ограничения компиляторов;
* алгоритмы работы компиляторов;
* особенности операционных систем.

Архитектура существенно влияет на семантику языка, а, следовательно, и на переносимость программных файлов. Во-первых, архитектура определяет множества значений арифметических типов, фиксируя тем самым семантику большинства операций языка. Во-вторых, от архитектуры, а именно, от системы команд, зависит интерпретация операций языка, остающихся недоопределенными даже после фиксирования множеств значений соответствующих типов. В-третьих, от архитектуры зависит схема размещения данных тех или иных типов в соответствующих элементах памяти.

Даже если программа удовлетворяет всем ограничениям ANSI C и прошла стадию компиляции в исходной ВС, может случиться, что в целевой ВС она эту стадию не пройдет из-за того, что некоторые метрические характеристики программы не удовлетворяют ограничениям, принятым в целевой ВС. Примерами таких характеристик являются: число уровней вложенностей составных операторов, операторов цикла и операторов выбора варианта; число описателей указателя, массива и функции, модифицирующих базовый тип в описании объекта; число выражений, вложенных друг в друга по круглым скобкам и т.п.

От алгоритмов работы компилятора зависит, например, порядок вычисления выражений, что влияет как на значения выражений, так и на вырабатываемый ими побочный эффект.

Наконец, семантика многих стандартных библиотечных функций (например, функций ввода/вывода) зависит от особенностей операционной системы.

Все перечисленные факторы учтены в определении ANSI C путем фиксирования неуточняемого (стандартом) поведения программ, неопределенного поведения программ и поведения программ, определяемого реализацией.

Неуточняемое поведение (unspecified behavior) - это поведение правильных программ с корректными данными в ситуациях, для которых стандарт не выдвигает никаких требований.

Неопределенное поведение (undefined behavior) - поведение (динамически) ошибочных программ с возможно некорректными данными или объектами с неопределенными значениями, для которых стандарт не выдвигает никаких требований. Диапазон неопределенного поведения может быть очень разнообразен: от полного игнорирования ситуации с непредсказуемыми результатами до поведения (во время трансляции или выполнения) в соответствии с документацией, описывающей характеристики среды (с выдачей диагностических сообщений или без таковой); возможны случаи преждевременного завершения трансляции или вычислений (с обязательной выдачей диагностического сообщения).

Поведение, определяемое реализацией (implementation-defined behavior) - поведение правильно написанной программы с правильными данными, которое зависит от характеристик реализации и которое должно быть документировано каждой реализацией.

В качестве общей рекомендации по написанию переносимых программ можно посоветовать, во-первых, безусловно избегать использования в программах языковых конструкций с неопределенным поведением, во-вторых, избегать конструкций с неуточняемым поведением в случаях, когда результат ее работы не является однозначным, и, наконец, минимизировать число конструкций, чье поведение определяется реализацией и существенно влияет на результат работы программы.

Другая общая рекомендация заключается в использовании возможностей препроцессора Си для локализации непереносимых фрагментов программы. Это касается использования макроимен вместо явных констант, зависящих от реализации; использования условной трансляции для включения в окончательный текст программы того или иного фрагмента в зависимости от вычислительной системы (особенно это касается конструкций, чье поведение определяется реализацией и существенно влияет на результат работы программы) и т.д.

Далее мы перечисляем все случаи неуточняемого, неопределенного и зависящего от реализации поведения программ, а, кроме того, в наименее очевидных случаях объясняем их влияние на переносимость. После этого приводятся требования стандарта к метрическим ограничениям компиляторов.

**Неуточняемое поведение**

Не уточняются следующие вопросы:

* Метод и время инициации статических данных.

В зависимости от того, вычисляются ли инициирующие константные выражения в окружении трансляции или в окружении выполнения программы, статические данные могут получать различные начальные значения.

* Ситуация, когда выдается печатный символ, а активная позиция находится в конце строки.
* Ситуация, когда выдается символ "шаг назад", а активная позиция находится в начале строки.
* Ситуация, когда выдается символ "горизонтальная табуляция", а активная позиция находится "на" или "за" последней определенной позицией горизонтальной табуляции.
* Ситуация, когда выдается символ "вертикальная табуляция", а активная позиция находится "на" или "за" последней определенной позицией вертикальной табуляции.

Предыдущие четыре ситуации влияют на вывод текста на дисплей.

* Представление плавающих типов.

Переносимая программа не должна использовать информацию о представлении (т.е. о битовой структуре) плавающих типов, поскольку именно в реализации плавающей арифметики существенно различаются разные вычислительные системы.

* Порядок вычисления выражений - в любом порядке, учитывающем только правила предшествования операций и расстановку скобок.
* Порядок, в котором возникают побочные эффекты.
* Порядок, в котором вычисляются параметры вызова функции и само значение этой функции.

За исключением тех случаев, когда порядок вычисления выражения зафиксирован синтаксическими правилами или указан в стандарте каким-либо другим образом (для операции вызова функции (), операций логического умножения, логического сложения, условной операции и операции перечисления выражений), порядок вычисления подвыражений и порядок возникновения побочных эффектов не уточняется. Выражение, содержащее более, чем одно вхождение одной и той же коммутативной и ассоциативной бинарной операции (\*, +, &, ^, |), может свободно перегруппировываться, независимо от наличия скобок, при условии, что типы операндов или результаты от такой перегруппировки не изменятся. В переносимой программе следует избегать выражений, порядок вычисления которых существенно влияет на их значения или вырабатываемые побочные эффекты. Если же такое выражение возникает, то содержащий его оператор всегда можно разбить на эквивалентную последовательность из нескольких операторов, не содержащих подобных выражений. Например, оператор

x=f()+g();

можно заменить на последовательность операторов

y=f();

x=y+g();

или

y=g();

x=f()+y;

в зависимости от нужного порядка вызова функций f() и g().

Чтобы зафиксировать некоторое конкретное группирование операций, нужно присвоить значение выражения, которое требуется явно выделить, некоторому объекту данных, либо поставить перед группирующими скобками унарный оператор плюс.

* Отведение памяти под формальные параметры.

Переносимая программа не должна использовать информацию о распределении памяти под формальные параметры, поскольку не только разные компиляторы по-разному решают эту задачу, но даже один компилятор может различным образом отводить память под формальные параметры при различных режимах своей работы.

* Значение индикатора позиции файла после успешного выполнения функции ungetc для текстового потока до тех пор, пока не будут введены или уничтожены все запомненные символы.
* Подробности о значении, запоминаемом в случае успешной работы функции fgetpos.
* Подробности о значении, вырабатываемом для текстового потока в случае успешной работы функции ftell.
* Порядок и взаимное расположение областей памяти, захватываемых функциями calloc, malloc и realloc.
* Какой из двух элементов, оказавшихся равными при сравнении, возвращается функцией bsearch.
* Порядок расстановки в отсортированном функцией qsort массиве двух элементов, оказавшихся равными при сравнении.
* Структура календарного времени, возвращаемого функцией time.

Переносимая программа не использует перечисленную информацию, поскольку она либо различается для разных реализаций языка, либо даже является случайной в рамках одной реализации.

**Неопределенное поведение**

Поведение не определяется для следующих ситуаций:

* В исходной программе обнаружен символ, не входящий в требуемый набор. Исключение делается для препроцессорных лексем, символьных и строковых констант, а также примечаний.
* Делается попытка модифицировать строковую константу.
* Идентификаторы, которые должны обозначать одну и ту же сущность, различаются хотя бы одним символом.
* В символьной или строковой константе обнаружена неизвестная управляющая последовательность.
* Лексически первое описание функции или объекта данных с внешней связью не имеет файловой области видимости, а последующее описание лексически идентичного идентификатора имеет либо внутреннюю, либо внешнюю связь, что противоречит первому описанию.
* Арифметическое преобразование дает результат, который не может быть представлен в отведенном пространстве.
* Арифметическая операция неверна (например, деление на 0) или выдает результат, который нельзя представить в отведенном пространстве (например, переполнение или потеря значимости).
* Число фактических параметров вызова не согласуется с числом формальных параметров функции, которая не имеет действующего в данной области видимости прототипа.
* Типы фактических параметров вызова после расширения не согласуются с расширенными типами формальных параметров функции, которая не имеет действующего в данной области видимости прототипа и не имеет прототипа, действующего в области видимости, соответствующей области определения функции.
* Прототип функции имеется в области видимости, соответствующей области определения функции, формальный параметр описан с типом, который изменяется в результате действия расширений типа, проводимых по умолчанию, а функция вызывается, когда в области видимости нет семантически эквивалентного прототипа.
* Вызывается функция, обрабатывающая переменное число параметров, но прототип с эллиптической нотацией отсутствует в данной области видимости.
* Вызывается функция с прототипом, видимым в данной области, ее формальный параметр описан с типом, который изменяется в результате действия расширений типа, проводимых по умолчанию, но в области определения функции не видно семантически эквивалентного прототипа функции.
* Встретилась неверная ссылка на массив, ссылка на пустой указатель или ссылка на объект, размещенный в области автоматически распределяемой памяти завершившегося блока.
* Указатель на функцию преобразуется в указатель на функцию другого типа и используется для вызова функции, тип которой отличается от первоначального.
* Указатель на объект, не являющийся элементом массива, используется в операции прибавления или вычитания константы.
* Вычисляется разность указателей, относящихся к разным массивам.
* Результат выражения сдвигается на отрицательную величину или на величину, большую или равную (в битах) размеру сдвигаемого результата.
* Сравниваются указатели, относящиеся к разным составным объектам.
* Значение объекта присваивается перекрывающемуся по памяти объекту.
* Делается попытка изменить объект, описанный как константа, с помощью указателя на тип, в котором нет атрибута const.
* Объект, описанный с атрибутом volatile, указывается с помощью указателя на тип, не имеющего такого же атрибута.
* Описания объекта, имеющего внешнюю связь, в двух разных файлах или в разных областях видимости одного файла, дают этому объекту разные типы.
* Значение автоматического неинициированного объекта используется до первого присваивания.
* Используется результат работы функции, которая, однако, не возвращает никакого значения.
* Функция, обрабатывающая переменное число параметров, определяется без списка типов параметров в эллиптической нотации.
* Фактический параметр макровызова не имеет ни одной препроцессорной лексемы.
* Внутри списка параметров макровызова имеются препроцессорные лексемы, которые могут быть проинтерпретированы как директивы препроцессора.
* В результате выполнения препроцессорной операции слияния лексем (##) получается неверная препроцессорная лексема.
* Эффект, возникающий в программе при переопределении зарезервированного внешнего идентификатора.
* Параметр identifier в макровызове offset соответствует битовому полю записи.
* Фактический параметр библиотечной функции имеет неверное значение, если только поведение этой функции в подобном случае не описано явно.
* Библиотечная функция, обрабатывающая переменное число параметров, не описана.
* Для доступа к настоящей функции assert использована макродиректива #undef.
* Фактический параметр функции, обрабатывающей символы, выходит за область определения.
* Вызов функции setjmp производится в ином контексте, нежели при сравнении с целочисленным выражением из констант в переключателе или в условном операторе.
* Значение автоматического объекта, не имеющего атрибута volatile, изменилось между вызовами setjmp и longjmp.
* Функция longjmp вызывается из динамически вложенной программы обработки сигнала.
* Сигнал возникает не в результате работы функций abort или raise, а при обработке сигнала вызывается библиотечная функция, не являющаяся самой функцией signal, или со статическим объектом проделывается не присваивание ему значения статической переменной с атрибутом volatile типа sig\_atomic\_t.
* Параметр parmN макроопределения va\_start описывается в классе регистровой памяти.
* При вызове макроимени va\_arg очередного фактического параметра не оказалось.
* Тип фактического параметра из списка параметров не согласуется с типом, указанным в макровызове va\_arg.
* Функция va\_end вызывается без предварительного обращения к макровызову va\_start.
* Из функции с переменным числом параметров, список которых был проинициирован с помощью макровызова va\_start, возврат производится до вызова va\_end.
* Формат в функциях fprintf и fscanf не соответствует списку фактических параметров.
* В формате функций fprintf или fscanf обнаружена неверная спецификация преобразования.
* Среди спецификаторов преобразования для спецификации, не входящей в список o, x, X, e, E, f, g и G встретился признак #.
* Фактическим параметром функции fprintf, не соответствующим преобразованиям %s и %p, является составной объект или указатель на составной объект.
* Отдельное преобразование в функции fprintf породило более 509 выходных символов.
* Фактическим параметром преобразования %p функции fscanf является значение указателя, выданное при преобразовании %p функцией fprintf во время предыдущих запусков программы.
* Результат преобразования, выполняемого функцией fscanf, не может быть представлен в объеме памяти, отведенной для него, или полученный объект имеет неподходящий тип.
* Результат преобразования строки в число с помощью функций atof, atoi или atol не может быть представлен.
* Фактический параметр функций free или realloc не совпадает с ранее полученными указателями, выработанными функциями calloc, malloc или realloс, или указывается объект, ранее уничтоженный вызовом функций free или realloc.
* Ссылка на память, освобожденную функциями free или realloc.
* При вызове из функции exit функция, зарегистрированная обращением к atexit, производит доступ к автоматическому объекту программы.
* Результат целочисленных арифметических функций (abs, div, labs или ldiv) не может быть представлен.
* Массив, в который идет запись копированием или конкатенацией, слишком мал.
* Функции memcpy, strcpy или strncpy копируют объект в перекрывающийся с ним по памяти другой объект.
* В формате функции strftime обнаружена неверная спецификация преобразования.

Все перечисленные ситуации являются ошибочными, однако разные реализации могут по-разному реагировать на них. Может даже случиться, что в некоторых реализациях программы с неопределенным поведением работают и выдают нужные результаты. Однако такие программы, как правило, невозможно перенести на другую вычислительную систему.

Например, используя в расчетной программе неверные арифметические операции (деление на ноль или операции, приводящие к переполнению или потере значимости), можно добиться удовлетворительной, с точки зрения конечного результата, работы этой программы за счет использования нюансов обработки таких исключительных ситуаций в рамках конкретной вычислительной системы. На других же вычислительных системах эта программа либо вообще не будет работать, либо будет выдавать неудовлетворительные результаты. Больше того, может потребоваться даже изменение алгоритма, реализуемого программой, из-за невозможности воспроизвести использованные нюансы исходной вычислительной системы хотя бы потому, что программист мог и не знать обо всех использованных тонкостях аппаратуры по принципу "есть результат и ладно" (кстати, техническая документация может и не содержать описания всех тонкостей).

Возникновения ситуаций с неопределенным поведением можно, а при разработке переносимых программ, безусловно, нужно избегать.

**Поведение, зависящее от реализации**

Каждая реализация должна описать поведение во всех ситуациях, перечисленных в этом разделе.

Семантика фактических параметров функции main.

Для облегчения переноса программы полезно локализовать обработку внешних аргументов.

Число значащих начальных символов (сверх 31) в идентификаторе без внешней связи.

В переносимой программе не используется свыше 31 значащего символа в идентификаторах без внешней связи.

Число значащих начальных символов (сверх 6) в идентификаторе с внешней связью.

В переносимой программе не используется свыше 6 значащих символов в идентификаторах с внешней связью.

Имеет ли значение регистр символов, входящих в идентификаторы с внешней связью.

При разработке переносимых программ лучше исходить из того, что регистр символов, входящих в идентификатор с внешней связью, не имеет значения (т.е. не различаются заглавные и прописные буквы).

Символы входного алфавита, кроме явно определенных в стандарте.

Это касается, в основном, символов, используемых в символьных и строковых константах (например, русские буквы).

Символы из набора времени выполнения (за исключением пустого символа и (в окружении выполнения) явно определенных символов входного символьного набора) и их коды.

В переносимых программах нежелательно использование информации о кодах символов, поскольку они могут различаться в разных реализациях.

Соответствие символов входного алфавита (в символьных и строковых константах) символам алфавита времени выполнения.

В основном это касается управляющих символов. Например, символ "конец строки" (\n) в разных реализациях может быть представлен в потоках ввода-вывода различными последовательностями кодов. Надо стараться писать программу так, чтобы ее поведение не зависело от конкретного представления управляющих символов в окружении выполнения.

Число и порядок символов в целом.

Эти различия несущественны в самостоятельных программах, которые не позволяют себе играть типами (например, преобразуя указатель на целое в указатель на символы и проверяя содержимое памяти по указателю), но могут проявиться при обработке данных, поступающих извне.

Число и порядок следования разрядов в символах из набора символов времени выполнения.

Значение символьной константы, состоящей из символа или управляющей последовательности, не представимой в алфавите времени выполнения.

Переносимой программе не следует использовать информацию этих двух пунктов.

Значение символьной константы, состоящей более, чем из одного символа.

В переносимой программе не следует использовать символьные константы более, чем из одного символа.

Следует ли трактовать "простые" символьные объекты как знаковые или беззнаковые.

Переносимая программа не должна зависеть от того, является ли тип char знаковым или беззнаковым.

Представление и наборы значений различных целочисленных типов.

В переносимой программе лучше всего исходить из минимальных наборов значений, зафиксированных стандартом, а также из той минимальной информации о представлении, которая в приводится в стандарте.

Результат преобразования целого к более короткому знаковому целому или результат преобразования беззнакового целого к знаковому целому той же длины, если значение не может быть представлено.

Переносимая программа не использует эту информацию.

Результаты поразрядных операций над знаковыми целыми.

В переносимой программе следует использовать только такие поразрядные операции, результат которых не зависит от реализации.

Знак остатка целочисленного деления.

Переносимая программа не использует эту информацию.

Является ли сдвиг вправо значения знакового целочисленного типа логическим или арифметическим.

Переносимая программа не должна зависеть от вида сдвига вправо знаковых целых.

Представление и наборы значений различных типов вещественных чисел.

Переносимая программа не зависит от представления вещественных чисел. Наборы значений вещественных типов влияют на точность вычислений.

Способ округления, когда вещественное число преобразуется к более узкому вещественному числу.

В переносимой программе лучше всего исходить из того, что способ округления неизвестен.

Тип целого, которое может вместить максимальный размер массива, то есть тип size\_t - тип результата операции sizeof.

Результат преобразования указателя в целое и наоборот.

Тип целого, которое может вместить разность между двумя указателями на один и тот же массив - ptrdiff\_t.

Переносимая программа не должна использовать информацию предыдущих трех пунктов.

Элемент смеси union используется как элемент другого типа.

Переносимая программа не должна осуществлять доступ к элементу смеси после того, как был изменен элемент смеси другого типа, поскольку в этом случае используется информация о битовой структуре представления значения соответствующего типа.

Дополнение пустот и выравнивание элементов записей.

Это обычно не доставляет проблем, если только двоичные данные, записанные одной реализацией, не читаются другой. Конечно же, не следует использовать эту информацию в переносимой программе.

Считается ли "простое" целое битовое поле знаковым или беззнаковым.

Переходит ли битовое поле, не умещающееся в одном целом, в следующее.

Порядок расположения битовых полей в целом.

Может ли битовое поле пересекать физические границы ячеек памяти.

Переносимая программа не должна использовать всю эту информацию.

Максимальное число описателей, которые могут модифицировать базовый тип.

Переносимой программе нужно исходить из того, что любая реализация должна допускать использование в модификации базового типа, либо непосредственно, либо через эквивалентность типов, по крайней мере 12 описателей указателей, массивов и функций (в любых комбинациях).

Максимальное число вариантов в переключателе.

Переносимая программа должна исходить из того, что число вариантов в переключателе не должно превышать 255.

Будет ли значение односимвольной символьной константы в выражении, управляющем условным включением фрагментов программ, совпадать со значением такой же константы в наборе символов окружения выполнения. Может ли такая константа иметь отрицательное значение.

Метод связи с входными файлами, подлежащими включению в программу.

Обработка имен в кавычках, относящихся к включаемым файлам.

Поведение каждой директивы #pragma.

Определение имен \_\_DATE\_\_ и \_\_TIME\_\_, когда, соответственно дата и время трансляции не может быть доступно.

Константа, получающаяся при подстановке макроопределения NULL, обозначающая пустой указатель.

Предыдущие 6 пунктов описывают зависящее от реализации поведение препроцессора. Остальные пункты описывают определяемое реализацией поведение библиотечных программ.

Диагностическое сообщение и способ завершения программы, применяемый в функции assert.

Наборы символов, проверяемые в функциях isalnum, isalpha, iscntrl, islower, isprint и isupper.

Значения, выдаваемые математическими функциями при возникновении ошибок области определения.

Устанавливают ли математические функции целое выражение errno в положение ERANGE при возникновении потери значимости.

Набор сигналов для функции signal.

Семантика каждого сигнала, распознаваемого библиотечной функцией signal.

Обработка умолчаний и входов в программу для каждого вида сигналов, распознаваемых функцией signal.

Восстанавливается ли стандартная обработка, если при обработке сигнала функцией, указанной при вызове функции signal, возникает сигнал SIGILL.

Нужно ли заканчивать последнюю строку текстового потока символом "конец строки".

Появятся ли при вводе обычные пробелы, записанные в текстовый поток непосредственно перед символом конца строки текста.

Количество символов NULL, которые дописываются к двоичному потоку.

Характеристики буферизации файлов.

Существует ли файл нулевой длины.

Правила образования правильных имен файлов.

Может ли один файл открываться много раз.

Результат выполнения функции remove над открытым файлом.

Эффект работы функции rename, если файл с новым именем существовал ранее.

Выходная строка, получающаяся при работе преобразования %p в функции fprintf.

Входная строка, поступающая для преобразования %p в функции fscanf.

Интерпретация символа ^, который есть ни первый, ни последний символ в списке сканирования в преобразовании %[ в функции fscanf.

Значение, которое получает errno от функций fgetpos и ftell в случае неудачи.

Сообщения, выдаваемые функцией perror.

Поведение функций calloc, malloc и realloc в случае, если размер запрошенной памяти равен нулю.

Поведение функции abort по отношению к открытым и временным файлам.

Статус, возвращаемый функцией exit, если значение фактического параметра не равно нулю, или значениям макроимен EXIT\_SUCCESS и EXIT\_FAILURE.

Набор имен окружения и метод изменения списка окружения, используемый функцией getenv.

Содержание и режим выполнения командной строки функцией system.

Знак значения, возвращаемого функцией сравнения (memcmp, strcmp или strncmp), если первая пара различающихся символов разнится в старшем разряде.

Содержание строк сообщений об ошибках, возвращаемых функцией strerror.

Местный временной пояс и летнее время.

Точка отсчета для функции clock.

**Метрические ограничения переносимой программы**

Переносимая программа должна удовлетворять следующим метрическим ограничениям:

* 15 уровней вложенности составных операторов, операторов цикла и операторов выбора варианта.
* 6 уровней вложенности условной трансляции.
* 12 описателей указателя, массива и функции, модифицирующих базовый тип в описании объекта.
* 127 выражений, вложенных друг в друга по круглым скобкам.
* 31 значащий символ в начале идентификатора с внутренней связью или имени макроопределения.
* 6 значащих символов в начале имен, имеющих внешнюю связь.
* 511 внешних имен в одном исходном файле.
* 127 имен в одном блоке.
* 1024 имени макроопределений, одновременно действующих в одном исходном файле.
* 31 параметр в вызове или определении функции.
* 31 параметр в макровызове или макроопределении с параметрами.
* 509 символов в одной логической исходной строке.
* 509 символов в строковой константе (после конкатенации).
* 32767 байтов для размещения объекта.
* 8 уровней вложенности по включаемым файлам.
* 255 меток выбора варианта в переключателях.

#### Обеспечение независимости от особенностей версии ОС UNIX

Специфичным видом мобильности приложений на уровне исходных текстов является возможность их выполнения с несколькими версиями одного и того же варианта ОС UNIX, включая ранние версии, далекие от современных стандартов. Достаточно часто поздние версии не обеспечивают полной совместимости с более ранними версиями, поскольку такая совместимость не дала бы возможности добиться в поздних версиях соответствия стандартам.

По-видимому, единственным на текущий момент практическим приемом для достижения такого рода мобильности приложений на уровне исходных текстов является развитое применение операторов условной компиляции в текстах программ (условной компиляции на уровне файлов включения часто оказывается недостаточно, поскольку в зависимости от версии системы в прикладной системе приходится использовать разные комбинации системных вызовов и других библиотечных функций). Обычно в основных файлах включения поддерживаются символические константы, значения которых позволяют судить об особенностях используемой версии ОС. Опираясь на значения этих констант, можно добиться того, что правильно написанная прикладная программа будет правильно компилироваться (и собираться) в среде конкретной версии операционной системы. Наличие единого текста облегчает сопровождение прикладной программы и облегчает достижение его одинаковой функциональности при работе с разными версиями системы (хотя такие тексты, переполненные операторами условной компиляции, обычно очень трудно читаются; можно только рекомендовать по мере возможности локализовать куски программы, зависящие от версии ОС).

### Бинарная совместимость

Если обычно достижение мобильности прикладных программ является целью прикладных программистов, то иногда достижение бинарной совместимости при выполнении прикладных программ является задачей разработчиков операционных систем. Под бинарной совместимостью операционной системы О2 с операционной системой О1 понимается возможность выполнения в среде О2 без перекомпиляции (а, возможно, и без перекомпоновки) приложений, написанных для выполнения в среде О1. Естественно, что бинарная совместимость двух операционных сред теоретически достижима только в том случае, когда обе операционные системы О1 и О2 базируются на некоторой общей аппаратной платформе (реально, чаще всего приходится слышать о бинарной совместимости разных вариантов ОС UNIX, работающих на платформах Intel).

Двоичная совместимость новой операционной системы с некоторой существующей ОС требуется в том случае, когда, во-первых, необходимо доказать пользователям, что новая система не только обладает новыми качествами, но и настолько технологична, что может выполнять существующие приложения даже без потребности их перекомпиляции. Во-вторых, двоичная совместимость позволяет немедленно сделать доступным в новой операционной среде весь накопленный в старой среде багаж приложений (исходные тексты которых будут, скорее всего, недоступны), что может оказаться очень существенным для потенциальных конечных пользователей (потребителей приложений) новой системы.

#### Возможности достижения бинарной совместимости

Конечно, прежде всего требуется полная аппаратная совместимость используемых платформ (по крайней мере, на уровне пользователя). Далее возможны два варианта. В первом, более простом варианте обеспечивается двоичная совместимость в смысле использования в новой операционной среде объектных файлов, откомпилированных в расчете на прежнюю операционную среду. Для получения выполняемой программы в новой среде требуется перекомпоновка программы (конечно, для этого компоновщик выполняемых программ новой ОС должен понимать структуру объектных модулей старой системы). Этот вариант близок к подходу переносимости программ на уровне исходных текстов, поскольку старые объектные модули содержат только пользовательский код и вызовы библиотечных функций и, очевидно, будут выполняться в новой среде без проблем. Все, что остается сделать (но это очень непростая задача) - это добиться полной совместимости со старой средой на уровне системных библиотек всех уровней. Нужно заметить, что этот вид бинарной совместимости не очень эффектен и не очень практичен, поскольку наборы объектных файлов приложений получить не намного проще, чем их исходные тексты. Обычно доступны выполняемые программы.

Во втором варианте в новой операционной системе возможно выполнение построенных в старой операционной среде выполняемых файлов. Это уже полностью скомпонованные программы, содержащие, кроме обычных пользовательских команд, только специальные команды вызова функций ядра операционной системы (обычно, разновидности команды trap). С одной стороны, для обеспечения этого вида бинарной совместимости не требуется воспроизводить весь набор библиотек старой операционной среды, но, с другой стороны, требуется полностью воспроизвести интерфейс с ядром старой операционной системы на самом низком уровне. Понятно, что это выполнимая, но трудная техническая задача (поскольку детали этого интерфейса обычно публично недоступны).

#### Преимущества и ограничения

О преимуществах подхода бинарной совместимости мы уже сказали в начале этого раздела: привлечение прикладных программистов возможностью использовать свои старые программы без каких-либо переделок и привлечение конечных пользователей возможностью использовать все накопленные приложения.

Ограничением систем, обеспечивающих бинарную совместимость является то, что позволяя использовать старые приложения, они тормозят использование новых качеств системы (а новые системы всегда обладают новыми качествами, в противном случае их было бы бессмысленно делать). Кроме того, в наше время аппаратные архитектуры развиваются так быстро, что их развитие тормозится требованиями аппаратной совместимости платформ одной линии даже на пользовательском уровне. Как кажется (это субъективное мнение автора), в этих условиях достижение бинарной совместимости операционных систем на одной аппаратной платформе становится слишком дорогостоящей и неблагодарной задачей.