Fighting the Lemmings

Martin Husemann <martin@NetBSD.org>
Перевод: Serj Kalichev <serj.kalichev@gmail.com>

26 марта 2008 г.
Version 1.01

Аннотация

Цитата из cvs лога справочной странички byteorder.3 системы NetBSD:

```
revision 1.8
date: 2001/11/29 22:55:57; author: ross; state: Exp; lines: +1 -6
Delete the old BUGS section entry:
> On the VAX bytes are handled backwards from most everyone else in
> the world. This is not expected to be fixed in the near future.
```

Сколько иронии в этой записи...

Содержание

1 Введение

В UNIX сообществе ходила крылатая фраза “Весь мир - VAX”. Её вспоминали каждый раз, когда сталкивались с кодом или техническими решениями, не имеющими под собой сколько-нибудь простого и рационального объяснения.

В наши дни все пишут под Linux и всё работает на машинах с архитектурой i386. При переносе программного обеспечения на другие (зачастую сильно отличающиеся) архитектуры, люди сталкиваются с, мягко говоря, “не совсем оптимальными” решениями и тогда они строят разного рода подпорки, совершают другие шаманские действия с программным кодом. Впоследствии это вызовет проблемы. К сожалению, или к счастью, усовершенствования в gcc приведут к тому, что почти все эти заплатки превратятся в ошибки.

На первый взгляд не-переносимый код может показаться проще. То же относится и к жёстко специализированным архитектурным решениям. Скорее всего, в более далёкой перспективе, все они станут неработоспособными.

Эта статья попытается указать на некоторые типичные проблемы, возникающие при написании переносимого кода. Не смотря на то, что многие проблемы очевидны, ошибки, с ними связанные, продолжают регулярно появляться в реальном коде.

1.1 Зачем заботиться о переносимости?

Не существует способа объективно измерить такое свойство исходного кода, как переносимость. Это вообще многогранное понятие. Можно по факту измерить объём усилий, затраченных на портирование конкретного кода, можно оценить насколько более “туманным” станет после этого код. Однако это не даст никакого представления о том, насколько трудоёмким будет следующее портирование.

Сегодня многие проекты ориентированны на строго определённую архитектуру. Авторы таких проектов поддаются соблазну не учитывать требования переносимости при создании программного обеспечения, называя это “оптимизацией” для данной целевой системы. Но целевые системы изменяются — меняется версия операционной системы, меняется сопутствующее программное обеспечение, меняется графическая подсистема. Изменения могут стать ещё более радикальными, если сменить операционную систему или аппаратную часть.

Если при создании кода переносимость будет одной из наиболее приоритетных задач, то количество проблем, возникающих во время неизбежных внешних изменений, резко снизится.

1.2 Переносимость - понятие относительное

Ниже приведён знаменитый пример из IOCCC (International Obfuscated C Code Contest - Международный Конкурс Невразумительного C Кода). Этот код переносим между системами VAX и PDP-11:

```
short main[] = {
      277, 04735, -4129, 25, 0, 477, 1019, 0xbef, 0, 12800,
      -113, 21119, 0x52d7, -1006, -7151, 0, 0x4bc, 020004,
      14880, 10541, 2056, 04010, 4548, 3044, -6716, 0x9,
      4407, 6, 5568, 1, -30460, 0, 0x9, 5570, 512, -30419,
      0x7e82, 0760, 6, 0, 4, 02400, 15, 0, 4, 1280, 4, 0,
      4, 0, 0, 0, 0x8, 0, 4, 0, ',', 0, 12, 0, 4, 0, '#',
      0, 020, 0, 4, 0, 30, 0, 026, 0, 0x6176, 120, 25712,
      'p', 072163, 'r', 29303, 29801, 'e'
};
```

(Победитель 1984 года. Авторы - Sjoerd Mullender и Robbert van Renesse. http://www.de.ioccc.org/1984/mullender.c)

Приведённый массив данных представляет из себя работоспособный код для VAX и PDP-11, который выводит на экран сообщение. Первое слово в массиве — инструкция перехода для PDP-11 по которой управление передаётся коду, специфичному для этой архитектуры. В то же время на VAX стартовый C-код использует инструкцию “calls”, которая предполагает, что по начальному адресу подпрограммы находится битовая маска регистров, которые надо сохранить. Таким образом на VAX исполнение кода начинается со второго слова.

1.3 Переносимость - дорогое удовольствие

Переносимость редко является основной целью проекта и, поэтому может восприниматься как помеха для быстрого достижения результатов. Естественно, это заблуждение. Портирование — дорогое удовольствие, если требования переносимости не учитывались в процессе разработки, но замена ПО — ещё дороже.

1.4 Переносимость - результат реального портирования

Можно заранее ориентироваться на переносимость, однако без реального портирования, в коде всегда останутся ошибки.

1.5 Существует ли единый тест на переносимость?

К сожалению — нет. Хотя портирование на NetBSD/sparc64 практически является таким тестом ;-)

2 Аспекты переносимости

2.1 Различия в GUI

Об этом можно написать целые книги. Различия в устройстве GUI, реализации и API — очевидны. Иногда даже в пределах одной архитектуры и операционной системы эти отличия хорошо заметны. Взять хотя бы GNOME и KDE. Существует ряд библиотек, например wxGTK, которые пытаются обеспечить единый интерфейс для различных GUI.

Так как в области GUI нет скрытых ловушек, статья больше не будет возвращаться к этой теме.

2.2 Различия операционных систем и API

Подходы к портированию будут различаться, в зависимости от того, на какой диапазон операционных систем предполагается портировать код. Существуют стандарты POSIX, придерживаясь которых, можно получить переносимый код для UNIX-подобных систем. Но это не избавит вас от проблемы с ASCII файлами, если Windows входит в число целевых систем. Функция CreateProcess() (создать новый процесс в win32 API) отличается от конструкции fork()/exec() и pthread_create(). Так что с этим ничего нельзя поделать. Поможет разве что #ifdef. А платой за переносимость станет запутанный, непонятный код. Иногда более разумным вариантом будет создание (или использование уже существующих) библиотек - обёрток.

2.3 Различия компиляторов

Использование расширений языка, выходящих за пределы стандарта C/C++, разрушает переносимость. Однако часто эти отличия компиляторов удаётся спрятать за макросами. К счастью, большая часть проблем такого рода проявляется на этапе компиляции. Соответственно их сравнительно легко выявить и устранить.

2.4 Код на ассемблере

Код на ассемблере ограничивает переносимость той машиной, для которой код был написан. А часто ещё и используемым компилятором. Простое решение проблемы - использовать условную компиляцию и Makefile для выбора между различными реализациями алгоритма на ассемблере и переносимой реализацией на C. Это приводит к разному поведению программы во время исполнения. Сравните (оптимизированные с помощью ассемблера) версии zlib и OpenSSL для машин i386 и версию на C для любого другого процессора со сравнимой скоростью. Это примеры для машин i386 демонстрируют ситуацию, когда переносимость не противоречит производительности.

2.5 Порядок следования байт

В зависимости от типа процессора, многобайтовые значения хранятся в памяти с разным порядком следования байт. В некоторых процессорах порядок следования байт можно выбирать (с помощью заводской прошивки,

матплаты или операционной системы). Для многих программ порядок следования байт неважен, так как они не используют внешнее двоичное представление данных (файл или сетевое соединение).

Двоичное представление может быть основано на простой последовательности байт — "байт-за-байтом" или использовать уже существующие макросы для изменения порядка следования байт, с последующим сохранением результата.

Пример на основе подхода "байт-за-байтом":

```
#define PUT_32BIT(f, val) \
      fputc((val >> 24) & 0xff , f); \
      fputc((val >> 16) & 0xff , f); \
      fputc((val >> 8) & 0xff , f);  \
      fputc(val & 0xff , f)
PUT_32BIT(f, origVal);
```

Пример изменения порядка следования байт с последующем сохранением результата:

```
int32_t val;
val = htole32(origVal) ;
fwrite(&val, sizeof val, 1, f) ;
```

Второй подход широко используется в коде сетевой подсистемы BSD. Нельзя сказать, что этот подход однозначно лучше и понятнее, но он позволяет преобразовывать структуры целиком и определять структуры, сходные с форматом хранения/передачи данных.

Надо заметить, что даже если порядок следования байт в процессоре и в двоичном представлении "случайно" оказался одинаковым, то данные всё равно пройдут процедуру преобразования.

2.6 Размер типа integer на различных машинах

Разные процессоры предъявляют различные требования к "правильному" представлению типов данных. Стандарт языка C намеренно не фиксирует представление большинства предопределённых типов данных и предоставляет компилятору право самому выбрать оптимальное представление. Из-за этого сильно усложняется разработка кода, определенного типа. Более поздние версии стандарта C учли это обстоятельство и ввели целочисленные типы фиксированного размера и "быстрые" целочисленные типы.

Существует множество синонимов к типам, определяемым в стандарте C (или POSIX). К примеру, time_t, size_t, off_t. Некоторые машины используют одинаковый размер для двух типов данных (типы int и long на i386) - стандарт это позволяет. На машине i386 компилятор не выдаст предупреждения, если передать указатель на int в функцию, ожидающую в качестве аргумента указатель на double. А на машинах, где типы int и double имеют разный размер (это все известные 64-битные архитектуры) появится предупреждение при компиляции.

Одни процессоры предпочитают беззнаковые значения (ARM, PowerPC) и для них "char" по-умолчанию означает "unsigned char", другие под "char" подразумевают "signed char" (i386). Опять же, стандарт C это разрешает.

И обычно никаких проблем не возникнет, если только код не использует char вместо int или не пытается определить свойства литеры с помощью доморощенных арифметических выражений, вместо специальных макросов для классификации ctype символов:

```
char c;
if (c < 0) { ...}
```

Естественно, на машинах, где char == unsigned char, это условие никогда не будет выполнено (и gcc выдаст соответствующее предупреждение). Ещё один пример, который будет работать с signed char и не будет с unsigned char:

```
char c;
FILE *f;
while ((c = fgetc(f)) != EOF) { ...}
```

Явная попытка читать символы из файла f, пока он не закончится. Макрос EOF определяется как -1. Соответственно, пока переменная с способна хранить -1, всё нормально. Но на машинах, где char == unsigned char, конструкция просто не будет работать. Надо заметить, что fgetc() определяется как:

```
int fgetc(FILE *);
```

Возвращаемое целое число - либо EOF, либо значение, которое можно представить с помощью char.

Более хитроумный пример, взят из NetBSD: структура struct timeval (используемая в частности системным вызовом gettimeofday(2)) определялась следующим образом:

```
struct timeval {
       long  tv_sec;  /* seconds */
       long  tv_usec; /* and microseconds */
};
```

Тут сразу две ошибки. Во-первых, практически везде должна использоваться структура struct timespec, определяемая стандартом POSIX.1b и имеющая точность до наносекунд. Когда-нибудь так и будет. Во-вторых, использование типа long для поля tv_sec. Естественно, и стандарт POSIX это определяет, поле должно иметь тип time_t. К сожалению, NetBSD заботится о двоичной совместимости, что заставляет прикладывать большое количество усилий и поддерживать версионность множества функций. Из-за этого ошибки до сих пор не исправлены. Теперь рассмотрим следующий код:

```
struct timeval tv;
time(&tv.tv_sec);
```

На первый взгляд все нормально, на машинах i386 код действительно компилируется и работает. А на sparc64 не работает (компилятор gcc выдаст предупреждение). Что здесь не так? Версии NetBSD для sparc и для

sparc64 во многом основаны на одном и том же коде, а sparc64 обеспечивает двоичную совместимость, т.е. позволяет исполнять двоичный код sparc (32-битная архитектура). Чтобы достичь такой совместимости, тип time_t имеет одинаковый размер 32 бита и в sparc и в sparc64 (тип time_t синоним int на sparc64). В функцию передается указатель на long (64-битное значение), в то время, как функция ожидает указатель на 32-битное значение. Компилятор gcc выдаст лишь предупреждение, хотя программа будет работать совсем не так, как ожидалось — результат будет записан в 4 старших байта переменной типа long, по сути сдвинув значение времени влево на 32 бита. В младших 32 битах окажутся произвольные значения.

Целый класс проблем, связанных с указателями на целые значения различного размера, порождается функциями, принимающими в качестве аргумента строку форматирования (как в printf()). К счастью, сейчас gcc проверяет строки форматирования на соответствие передаваемым параметрам, так что этот класс проблем проявится на этапе компиляции. А раньше, неверный размер аргументов приводил к разрушению стека, падениям и появлению уязвимостей. Предположим, необходимо извлечь из входной последовательности значения типа time_t, заданные в виде десятичной строки. На машинах i386 это можно реализовать так:

```
time_t t;
FILE *f;
fscanf(f, "%ld", &t);
```

И конструкция будет работать. Однако, на sparc64 такой код испортит стек — fscanf() ожидает значение типа long (8 байт), а передаваемый адрес указывает лишь на 4 байтовое значение. Возможно вам повезёт и вы наткнётесь на проблему с выравниванием (об этом читайте ниже), а может вызов функции перезапишет 4 байта в стеке. Новая версия стандарта C определяет дополнительные модификаторы для типов-синонимов, появились “%zd” для типа size_t и “%td” для типа ptrdiff_t, но не существует простого способа использовать тип time_t в функциях scanf()/printf(). Приведённый выше код должен был выглядеть так:

```
time_t t;
long temp ;
FILE *f;
fscanf(f, "%ld", &temp);
t = temp;
```

С печатью немного попроще, достаточно простого приведения типов:

```
time_t t;
printf("%ld\n", (long)t) ;
```

Если вы используете целочисленные типы фиксированного размера, то для них стандарт определяет макросы — спецификаторы формата, так как не очевидно какому “родному” типу соответствуют типы фиксированного размера.

Поэтому вместо:

```
int64_t t;
printf("%lld\n", t); /* works on i386 */
```

Лучше использовать:

```
#include <inttypes.h>
int64_t t;
printf("%" PRId64 "\n", t); /* portable */
```

2.7 Типы, зависимые от архитектуры

Формально, код не может быть переносимым, если во внешнем представлении данных используются значения с плавающей точкой. К счастью, сегодня все распространённые архитектуры используют формат представления значения с плавающей точкой, описанный стандартом IEEE 754. Но машины, где это не так тоже существуют в природе — это VAX, старые Cray, многие мэйнфреймы и некоторые реализации архитектуры ARM. Процессоры alpha используют формат IEEE для значений с плавающей точкой, но дополнительно к этому имеют инструкции загрузки и сохранения, которые понимают значения с плавающей точкой в формате VAX (вероятно, используются только в OpenVMS).

Тяжело создать переносимый код, если значения с плавающей запятой используются в двоичных файлах или потоках передачи данных. Решением может быть использование текстовых строк, значений с фиксированной точкой или формата IEEE с заранее известным размером (это ограничит переносимость, но не существенно).

В IEEE стандарте определено несколько размеров для значений с плавающей точкой. Наиболее часто используемые из них — это 32-битные (float на i386) и 64-битные (double на i386). Некоторые 64-битные машины используют 128-битные значения с плавающей запятой, а стандарт C избегает жёсткой привязки типов данных к размеру. Некоторые (RISC) архитектуры определяют 128-битные форматы с плавающей запятой и инструкции для работы с ними, но реализации на самом деле их не поддерживают, перекладывая эту задачу на операционную систему и программную эмуляцию (или целиком от них избавляются на этапе компиляции). Таким примером может служить UltraSparc.

Возможно, что для внутренних расчётов свойства конкретной реализации чисел с плавающей точкой, не будут иметь никакого значения. Благо для большинства алгоритмов это не критично. Но если всё-таки, при каких-то граничных условиях это будет важно, проблема решается с помощью хорошо известных доработок. Так что реальные проблемы переносимости, не считая случая хранения или передачи двоичных данных, чаще всего не связаны с форматами чисел с плавающей точкой.

2.8 Выравнивание

Так как размеры типов данных отличаются, то отличаются и размеры структур и смещения полей структуры при портировании кода с одной архитектуры на другую. Для некоторых архитектур требуется или рекомендуется расположение данных в памяти по определенным адресам. Общее правило выравнивания звучит так: все данные должны быть выравнены по

адресам, кратным размеру этих данных. Например, 32-битные целые значения должны храниться по адресам кратным четырем. Некоторые процессоры используют еще более строгие правила выравнивания. В современных машинах i386 для инструкций SSE2 требуется выравнивание по границе 16 байт с целью повышения производительности.

Реакция на несоблюдение правил выравнивания различается на разных процессорах:

- для доступа к невыравненным данным требуется больше времени (i386);
- запрос к операционной системе на эмуляцию доступа (alpha);
- будут получены неверные результаты без каких-либо предупреждений (некоторые варианты ARM);
- будет выдан сигнал "ошибка на шине" (bus error) (sparc и большинство других процессоров RISC архитектуры).

Платформы, относящиеся к нарушению правил лояльно (i386), плохо подходят для тестирования кода и выявления проблем с выравниванием.

Обычно компилятор сам заботится о правильном выравнивании. Но опять же существуют исключения — структуры предназначенные для хранения данных на диске или передачи данных и использующие директивы компилятора для упаковки полей структуры. С такими структурами надо обращаться осторожно. Кроме проблем выравнивания, нужно не забывать и про порядок следования байт. Кстати, если при решении проблемы правильного порядка следования байт используется подход простой байтовой последовательности "байт-за-байтом", то это автоматически решает проблемы с выравниванием.

Как только вы начнёте выделывать с компилятором разные хитрые, нестандартные вещи, он будет уже неспособен позаботиться о выравнивании самостоятельно. Чаще всего это случается при операциях приведения типов указателей. Эта тема заслуживает отдельной главы.

2.9 Приведение типов указателей

```
void encode_packet(unsigned char *packet, size_t len, uint32_t tag)
{
      *((uint32_t*)packet) = htole32(tag);
      packet += sizeof tag;
      len -= sizeof tag;
      ...
}
```

Вот такой код появился в 2004 году. В чем проблема? Код работает на i386, значит с ним все в порядке?!

Как только указатель на unsigned char (не имеющий каких-либо ограничений по выравниванию) приводится к указателю на тип с другими (более жесткими) ограничениями по выравниванию, компилятор больше не может заботиться о выравнивании и вся ответственность ложится на программиста. А как тогда исправить код? Процессор с более строгими требованиями

к выравниванию (например sparc) не может обратиться по трём из каждых четырёх адресов, которые могут адресоваться с помощью указателя на unsigned char. Поэтому придется читать байт за байтом и из прочитанных байтов собирать 32-битное значение — так же как это делал макрос PUT_32BIT в одном из предыдущих примеров. Если заранее неизвестно, как выравнен указатель, придется применять этот метод, если же указатель гарантированно выравнен, то можно просто считать многобайтовое значение по этому адресу. Но учтите, компилятор не сможет принять решение за вас.

Правильный код будет выглядеть так:

```
void encode_packet(unsigned char *packet, size_t len, uint32_t tag)
{
      uint32_t le32tag = htole32(tag);
      memcpy(packet, &le32tag, sizeof le32tag);
      packet += sizeof le32tag;
      len -= sizeof tag;
      ...
}
```

Ещё один пример неверного приведения типов указателя, на этот раз в правой части выражения:

```
void decode_packet(unsigned char *packet, size_t len)
{
      struct header *h = (struct header)packet;
      switch (h->tag) {
            ...
      }
}
```

Адрес, хранящийся в packet, может не соответствовать ограничениям по выравниванию для структуры h. Стандарт C гарантирует что приведение типов не изменяет адрес, хранящийся в переменной — указателе. Таким образом приведение типов указателей не может решить проблемы с выравниванием.

Код можно улучшить с помощью функции memcpy():

```
void decode_packet(unsigned char *packet, size_t len)
{
      struct header h, *p = (struct header*)packet;
      memcpy(&h, p, sizeof h);
      switch (h.tag) {
            ...
      }
}
```

Но и этот код тоже не всегда будет работать правильно. Стандарт C разрешает компилятору думать, что указатель на struct header уже выравнен правильно, соответственно типу указателя. А в приведённом примере это вовсе не гарантированно. Компилятор может оптимизировать операцию memcpy() с помощью последовательных 64-битных инструкций загрузки/сохранения, вместо побайтового копирования. Это вызовет ошибку на шине (bus error).

Правильный код:

```
void decode_packet(unsigned char *packet, size_t len)
{
      struct header h;
      memcpy(&h, packet, sizeof h);
      switch (h.tag) {
            ...
      }
}
```

Важное правило: приведение типов в левой части выражения - зло. Для этого не может быть никаких оправданий.

Приведение типов указателей в правой части выражения тоже не приветствуется. Это может помешать компилятору выдать предупреждение о реально существующих проблемах или не позволить использовать оптимизацию. Стандарт C декларирует, что два указателя могут быть синонимами только в тех случаях, если они имеют один и тот-же тип (или его знаковые, беззнаковые варианты, варианты с квалификаторами), либо один из них указывает на char. Разрешается выполнить приведение к “правильному” типу объекта при присвоении через указатель, но компилятор будет считать, что все переменные типа, к которому выполняется приведение, могли измениться (см. примечание 4.2 в конце статьи). Приведение типов может помешать компилятору применить оптимизацию там, где она бы успешно применялась при нормально (без приведения типов) спроектированных интерфейсах.

2.10 Вызовы ioctl()

Функция ioctl() принимает указатель на void в качестве одного из своих аргументов и использует его как контейнер для любых других данных. Это не позволяет компилятору проверить аргументы на допустимость. Значение второго аргумента определяет тип передаваемых данных (третий аргумент). Несоответствие сложно отследить на этапе компиляции.

Некоторое время назад в исходных кодах X была обнаружена древняя ошибка: вызов ioctl() с командой FIONREAD, использующийся для определения возможности немедленного чтения из файлового дескриптора, принимал указатель на int в качестве третьего аргумента (согласно стандарту POSIX и коду NetBSD). Некоторые другие операционные системы, такие как windows и, вероятно, IRIX в 64-битном режиме, вместо этого используют указатель на unsigned long. Обсуждаемый кусок кода выглядел примерно так:

```
long arg;
ioctl(fd, FIONREAD, (char *)&arg);
return (int)arg;
```

На NetBSD/sparc64 это приводит к тому, что результат выполнения операции будет записан в 4 старших байта переменной arg, а в дальнейшем используются только 4 младших байта этой переменной. По случайному стечению обстоятельств такой код прекрасно работает на alpha - из-за другого порядка следования байт.

3 Заключение

NetBSD претендует на звание самой переносимой операционной системы в мире. Она работает на пятидесяти четырёх различных системных архитектурах, характеризующихся семнадцатью машинными архитектурами, которые в свою очередь охватывают пятнадцать процессорных семейств. Всё это достигается с использованием единой базы исходного кода, который проделал долгий путь, чтобы стать настолько переносимым. Но до сих пор в коде находят ошибки переносимости и, несмотря на большой опыт, такие ошибки появляются во вновь добавляемом коде. Переносимость - одна из основных целей проекта NetBSD.

Будет здорово, если все большее и большее число приложений последует такому примеру.

Чтобы достичь переносимости придется приложить определённые усилия, но если чётко осознавать возможные проблемы и способы их решения, то всё пройдет достаточно безболезненно. Надо только с самого начала разработки обращать внимание на переносимость. А теперь, все дружно отыскали систему NetBSD/sparc64 и, как только ваше программа заработает на основной целевой системе, портируйте её ещё и туда. ;–}

4 Примечания

4.1 Автор о названии

В названии (“Fighting the lemmings”) нет какого-то особого смысла. Раньше мы говорили “весь мир VAX”, а теперь весь мир — i386 и везде работает linux. Все авторы просто следуют за этим “леммингом — вожаком” и пишут (а ещё хуже — тестируют) код исключительно под i386/linux.

Возможно слово “fighting” (борьба) в названии не лучшим образом отражает суть, и “dealing” (поведение, иметь дело) или “cope” (справиться, совладать) было бы лучшим выбором.

4.2 О приведении типов указателей

В главе о приведении типов указателей попалась одна непонятная фраза: You are allowed to cast to the “right” type of an object when assigning via a pointer, so the compiler assumes all variables of the casted-to type might have changed. Вот комментарии автора по этому поводу:

«Скажем, у вас есть поток байтов, который включает в себя некоторый участок, и вы хотите обратиться к этому участку, как к структуре. Поток байтов не имеет выравнивания. Для указателя на байт используется char* или void*. Теперь, если привести тип указателя к типу struct* (указатель на вашу структуру), компилятор будет считать указатель допустимым, включая все требования по выравниванию, относящиеся к указателям на вашу структуру.

Это также справедливо в случае, если вы сперва выполнили приведение типа указателя, а потом использовали memcpy() для копирования содержимого структуры.

Однако, если вы выполните memcpy(), т.е. скопируете данные по указателю с неприведённым типом void*/char* в локальную типизированную

структуру, то компилятор не станет делать никаких предположений и всё будет работать нормально.»

Этот комментарий не до конца прояснил смысл фразы. Более понятно стало после комментария от gena2x. Речь идёт об особенностях оптимизации. Был приведён следующий пример:

```
$ cat test.c
#include <stdio.h>

void f(int *i, long *l) {
        printf("1.v=%ld\n", *l);
        *i = 11;
        printf("2.v=%ld\n", *l);
}

int main() {
        long b = 10;
        f((int*)&b, &b);
        printf("3.v=%ld\n", b);
}

$ gcc -O2 test.c -o test
$ ./test
1.v=10
2.v=10
3.v=11
```

Неожиданное значение “2. v=10”. Обратите внимание, при компиляции использовалась оптимизация -O2. Если оптимизацию не использовать, то вывод будет другим:

```
$ gcc test.c -o test
$ ./test
1.v=10
2.v=11
3.v=11
```

При использовании оптимизации компилятор “не заметил”, что значение *l изменилось. Если же строку “*i = 11;” заменить на “*(long *)i = 11;”, то компилятор сделает предположение, что могли измениться все значения по указателям, имеющим тип long * (тип к которому приводили), и соответственно при использовании *l будет получено уже новое значение (“2.v=11”).

4.3 Благодарности

Благодарю Martin Husemann, Петра Косых и gena2x за помощь в переводе.