# 20 ловушек переноса Си++ кода на 64-битную платформу

Автор: Андрей Карпов, Евгений Рыжков

Вашему вниманию предлагается статья, посвященная переносу программного кода 32-битных приложений на 64-битные системы. Статья составлена для программистов, использующих Си++, но может быть полезна всем, кто сталкивается с переносом приложений на другие платформы.

Нужно четко понимать, что новый класс ошибок, возникающий при написании 64-битных программ, не просто еще несколько новых некорректных конструкций, среди тысяч других. Это означает неминуемые сложности, с которыми столкнутся разработчики любой развивающейся программы. Данная статья поможет быть готовым к этим трудностям и покажет пути их преодоления.

Любая новая технология (как в программировании, так и в других областях) несет в себе помимо преимуществ, также и некоторые ограничения или даже проблемы использования этой технологии. Точно такая же ситуация сложилась и в области разработки 64-битного программного обеспечения. Мы все знаем о том, что 64-битное программное обеспечение - это следующий этап развития информационных технологий. Однако немногие программисты пока реально столкнулись с нюансами этой области, а именно разработки 64-битных программ.

Мы не будем задерживаться на преимуществах, которые открывает перед программистами переход на 64-битную архитектуру. Данной тематике посвящено большое количество публикаций, и читателю не составит труда их найти.

Целью этой статьи является подробный обзор тех проблем, с которыми может столкнуться разработчик 64-битных программ. В статье Вы познакомитесь:

- с типовыми ошибками программирования, проявляющими себя на 64-битных системах;
- с причинами, по которым эти ошибки проявляют себя (с соответствующими примерами);
- с методами устранения перечисленных ошибок;
- с обзором методик и средств поиска ошибок в 64-битных программах.

Приведенная информация позволит Вам:

- узнать отличия 32-битных и 64-битных систем;
- избежать ошибок при написании кода для 64-битных систем;
- ускорить процесс миграции 32-битного приложения на 64-битную архитектуру за счет существенного сокращения времени отладки и тестирования;
- более точно и обоснованно прогнозировать время переноса кода на 64-битную систему.

Для лучшего понимания изложенного материала, в статье приводится много примеров. Знакомясь с ними, Вы получите нечто большее суммы отдельных частей. Вы откроете дверь в мир 64-битных систем.

Для облегчения понимания дальнейшего текста вначале вспомним некоторые типы, с которыми мы можем столкнуться (см. таблица n1).

| Название типа | Размерность типа в битах (32-битная система) | Размерность типа в битах (64-битная система) | Описание |
|---|---|---|---|
| ptrdiff_t | 32 | 64 | Знаковый целочисленный тип, образующийся при вычитании двух указателей. Используется для хранения размеров. Иногда используется в качестве результата функции, возвращающей размер или -1 при возникновении ошибки. |
| size_t | 32 | 64 | Беззнаковый целочисленный тип. Результат оператора sizeof(). Служит для хранения размера или количества объектов. |
| intptr_tuintptr_tsize_tssize_tint_ptrdword_ptrи так далее | 32 | 64 | Целочисленные типы, способные хранить в себе значение указателя. |
| time_t | 32 | 64 | Время в секундах. |

Таблица n1. Описание некоторых целочисленных типов.

В тексте будет использоваться термин "memsize" тип. Под memsize-типом мы будем понимать любой простой целочисленный тип, способный хранить в себе указатель и меняющий свою размерность при изменении разрядности платформы с 32-бит на 64-бита. Примеры memsize-типов: size_t, ptrdiff_t, все указатели, intptr_t, int_ptr, dword_ptr.

Несколько слов следует уделить моделям данных, определяющим соотношения размеров фундаментальных типов для различных систем. В таблице n2 приведены модели данных, которые могут быть нам интересны.

| | ilp32 | lp64 | llp64 | ilp64 |
|---|---|---|---|---|
| char | 8 | 8 | 8 | 8 |
| short | 16 | 16 | 16 | 16 |
| int | 32 | 32 | 32 | 64 |
| long | 32 | 64 | 32 | 64 |
| long long | 64 | 64 | 64 | 64 |
| size_t | 32 | 64 | 64 | 64 |
| pointer | 32 | 64 | 64 | 64 |

Таблица n2. Модели 32-разрядных и 64-разрядных данных

По умолчанию в статье будет считаться, что перенос программ осуществляется с системы, имеющей модель данных ilp32, на системы с моделью данных lp64 или llp64.

И последнее: 64-битная модель в linux (lp64) и windows (llp64) имеет различие только в размерности типа long. Поскольку это их единственное отличие, то для обобщения изложения мы будем избегать использования типов long, unsigned long, и будем использовать типы ptrdiff_t, size_t.

Приступим к рассмотрению типовых ошибок, возникающих при переносе программ на 64-битную архитектуру.

# 1. Отключенные предупреждения

Во всех книгах, посвященных разработке качественного кода, рекомендуется выставлять уровень предупреждений, выдаваемых компилятором на как можно более высокий. Но на практике встречаются ситуации, когда для определенных частей проекта выставлен меньший уровень диагностики или вообще выключен. Обычно это очень старый код, который продолжает поддерживаться, но не модифицируется. Программисты, работающие в проекте, привыкли, что этот код работает, и закрывают глаза на его качество. Здесь и кроется опасность пропустить серьезные предупреждения компилятора при переносе программ на новую 64-битную систему.

При переносе приложения следует обязательно включить предупреждения для всего проекта, помогающие проверить код на совместимость и внимательно проанализировать их. Это может существенно сэкономить время при отладке проекта на новой архитектуре.

Если этого не сделать, то самые глупые и простые ошибки будут проявлять себя во всем своем многообразии. Вот простейший пример переполнения, который возникнет в 64-битной программе, если полностью игнорировать предупреждения:

unsigned char *array[50]; unsigned char size = sizeof(array); 32-bit system: sizeof(array) = 200 64-bit system: sizeof(array) = 400

# 2. Использование функций с переменным количеством аргументов

Классическим примером является некорректное использование функций printf, scanf и их разновидностей:

1) const char *invalidformat = "%u"; size_t value = size_max; printf(invalidformat, value);

2) char buf[9]; sprintf(buf, "%p", pointer);

В первом случае не учитывается, что тип size_t не эквивалентен типу unsigned на 64-битной платформе. Это приведет к выводу на печать некорректного результата, в случае если value > uint_max.

Во втором случае автор кода не учел, что размер указателя в будущем может составить более 32 бит. В результате на 64-битной архитектуре данный код приведет к переполнению буфера.

Некорректное использование функций с перемененным количеством параметров является распространенной ошибкой на всех архитектурах, а не только 64-битных. Это связано с принципиальной опасностью использования данных конструкций языка Си++. Общепринятой практикой является отказ от них и использование безопасных методик

программирования. Мы настоятельно рекомендуем модифицировать код и использовать безопасные методы. Например, можно заменить printf на cout, a sprintf на boost::format или std::stringstream.

Если Вы вынуждены поддерживать код, использующий функции типа sscanf, то в формате управляющих строк можно использовать специальные макросы, раскрывающиеся в необходимые модификаторы для различных систем. Пример:

// pr_sizet on win64 = "i" // pr_sizet on win32 = "" // pr_sizet on linux64 = "l" // ... size_t u; scanf("%" pr_sizet "u", &u);

# 3. Магические константы

В некачественном коде часто встречаются магические числовые константы, наличие которых опасно само по себе. При миграции кода на 64-битную платформу эти константы могут сделать код неработоспособным, если участвуют в операциях вычисления адреса, размера объектов или в битовых операциях.

В таблице n3 перечислены основные магические константы, которые могут влиять на работоспособность приложения на новой платформе.

| Значение | Описание |
|---|---|
| 4 | Количество байт в типе |
| 32 | Количество бит в типе |
| 0x7fffffff | Максимальное значение 32-битной знаковой переменной. Маска для обнуления старшего бита в 32-битном типе. |
| 0x80000000 | Минимальное значение 32-битной знаковой переменной. Маска для выделения старшего бита в 32-битном типе. |
| 0xffffffff | Максимальное значение 32-битной переменной. Альтернативная запись -1 в качестве признака ошибки. |

Таблица n3. Основные магические значения, опасные при переносе приложений с 32-битной на 64-битную платформу.

Следует внимательно изучить код на предмет наличия магических констант и заменить их безопасными константами и выражениями. Для этого можно использовать оператор sizeof(), специальные значения из , и так далее.

Приведем несколько ошибок, связанных с использованием магических констант. Самой распространенной является запись в виде числовых значений размеров типов:

1) size_t arraysize = n * 4; intptr_t *array = (intptr_t *)malloc(arraysize);

2) size_t values[array_size]; memset(values, array_size * 4, 0);

3) size_t n, newexp; n = n >> (32 - newexp);

Во всех случаях, предполагаем, что размер используемых типов всегда равен 4 байта. Исправление кода заключается в использовании оператора sizeof():

1) size_t arraysize = n * sizeof(intptr_t); intptr_t *array = (intptr_t *)malloc(arraysize);

2) size_t values[array_size]; memset(values, array_size * sizeof(size_t), 0);

или

memset(values, sizeof(values), 0); //preferred alternative

3) size_t n, newexp; n = n >> (char_bit * sizeof(n) - newexp);

Иногда может потребоваться специфическая константа. В качестве примера мы возьмем значение size_t, где все биты кроме 4 младших должны быть заполнены единицами. В 32-битной программе эта константа может быть объявлена следующим образом:

// constant `1111..110000` const size_t m = 0xfffffff0u;

Это некорректный код в случае 64-битной системы. Такие ошибки очень неприятны, так как запись магических констант может быть осуществлена различными способами и их поиск достаточно трудоемок. К сожалению, нет никаких других путей, кроме как найти и исправить этот код, используя директиву #ifdef или специальный макрос.

#ifdef _win64 #define const3264(a) (a##i64) #else #define const3264(a) (a) #endif const size_t m = ~const3264(0xfu);

Иногда в качестве кода ошибки или другого специального маркера используют значение "-1", записывая его как "0xffffffff". На 64-битной платформе записанное выражение некорректно и следует явно использовать значение -1. Пример некорректного кода, использующего значение 0xffffffff как признак ошибки:

#define invalid_result (0xffffffffu) size_t mystrlen(const char *str) { if (str == null) return invalid_result; ... return n; } size_t len = mystrlen(str); if (len == (size_t)(-1)) showerror();

На всякий случай уточним Ваше понимание, чему с вашей точки зрения равно значение "(size_t)(-1)" на 64-битной платформе. Можно ошибиться, назвав значение 0x00000000ffffffffu. Согласно правилам языка Си++ сначала значение -1 преобразуется в знаковый эквивалент большего типа, а затем в беззнаковое значение:

int a = -1; // 0xffffffffi32 ptrdiff_t b = a; // 0xffffffffffffffffi64 size_t c = size_t(b); // 0xffffffffffffffffui64

Таким образом, "(size_t)(-1)" на 64-битной архитектуре представляется значением 0xffffffffffffffffui64, которое является максимальным значением для 64-битного типа size_t.

Вернемся к ошибке с invalid_result. Использование константы 0xffffffffu приводит к невыполнению условия "len == (size_t)(-1)" в 64-битной программе. Наилучшее решение заключается в изменении кода так, чтобы специальных маркерных значений не требовалось. Если по какой-то причине Вы не можете от них отказаться или считаете нецелесообразным существенные правки кода, то просто используйте честное значение -1.

#define invalid_result (size_t(-1)) ...

# 4. Хранение в double целочисленных значений

Тип double, как правило, имеет размер 64-бита, и совместим со стандартом ieee-754 на 32-битных и 64-битных системах. Некоторые программисты используют тип double для

хранения и работы с целочисленными типами:

size_t a = size_t(-1); double b = a; --a; --b; size_t c = b; // x86: a == c // x64: a != c

Данный пример еще можно пытаться оправдывать на 32-битной системе, так как тип double имеет 52 значащих бит и способен без потерь хранить 32-битное целое значение. Но при попытке сохранить в double 64-битное целое число точное значение может быть потеряно (см. рисунок 1).

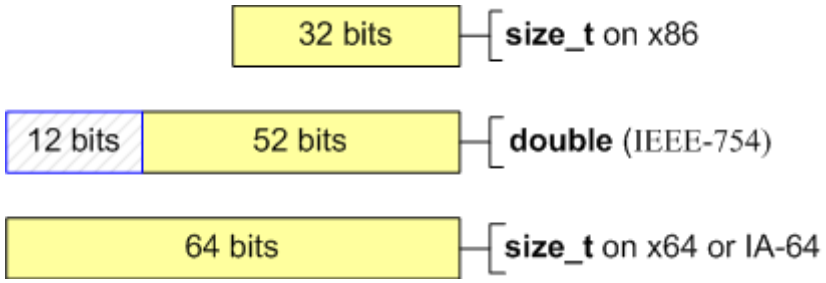



*Рисунок 1. Количество значащих битов в типах size_t и double.*

Возможно, приближенное значение вполне применимо в Вашей программе, но на всякий случай хочется сделать предупреждение о потенциальных эффектах на новой архитектуре. И в любом случае не рекомендуется смешивать целочисленную арифметику и арифметику с плавающей точкой.

# 5. Операции сдвига

Операции сдвига при невнимательном использовании, могут принести много неприятностей во время перехода от 32-битной к 64-битной системе. Начнем с примера функции, выставляющей в переменной типа memsize, указанный вами бит в 1:

ptrdiff_t setbitn(ptrdiff_t value, unsigned bitnum) { ptrdiff_t mask = 1 << bitnum; return value | mask; }

Приведенный код работоспособен на 32-битной архитектуре и позволяет выставлять биты с номерами от 0 до 31. После переноса программы на 64-битную платформу возникнет необходимость выставлять биты от 0 до 63. Как Вы думаете, какое значение вернет следующий вызов функции setbitn(0, 32)? Если Вы думаете, что 0x100000000, то авторы рады, что не зря подготовили эту статью. Вы получите 0.

Обратите внимание, что "1" имеет тип int и при сдвиге на 32 позиции произойдет переполнение, как показано на рисунке 2.

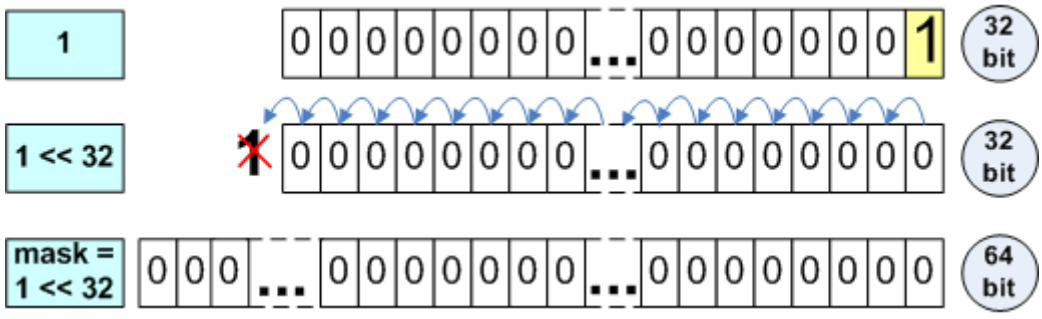



*Рисунок 2. Вычисление выражения "ptrdiff_t mask = 1 << bitnum".*

Для исправления кода необходимо сделать константу "1" того же типа, что и переменная mask.

ptrdiff_t mask = ptrdiff_t(1) << bitnum;

или

ptrdiff_t mask = const3264(1) << bitnum;

Еще один вопрос. Чему будет равен результат вызова неисправленной функции setbitn(0, 31)? Правильный ответ 0xffffffff80000000. Результатом выражения 1 << 31 является отрицательное число -2147483648. Это число представляется в 64-битной целой переменной как 0xffffffff80000000. Следует помнить и учитывать эффекты сдвига значений различных типов. Для лучшего понимания и наглядности изложенной информации в таблице n4 приведен ряд интересных выражений со сдвигами в 64-битной системе.

| Выражение | Результат (dec) | Результат (hex) |
|---|---|---|
| ptrdiff_t result; result = 1 << 31; | -2147483648 | 0xffffffff80000000 |
| result = ptrdiff_t(1) << 31; | 2147483648 | 0x0000000080000000 |
| result = 1u << 31; | 2147483648 | 0x0000000080000000 |
| result = 1 << 32; | 0 | 0x0000000000000000 |
| result = ptrdiff_t(1) << 32; | 4294967296 | 0x0000000100000000 |

Таблица n4. Выражения со сдвигами и результаты в 64-битной системе.

# 6. Упаковка указателей

Большое количество ошибок при мигрировании на 64-битные системы связано с изменением размера указателя по отношению к размеру обычных целых. В среде с моделью данных ilp32 обычные целые и указатели имеют одинаковый размер. К сожалению, 32-битный код повсеместно опирается на это предположение. Указатели часто приводятся к int, unsigned int и другим неподходящим типам для выполнения адресных расчетов.

Следует четко помнить, что для целочисленного представления указателей следует использовать только memsize типы. Предпочтение, на наш взгляд, следует отдавать типу uintptr_t, так как он лучше выражает намерения и делает код более переносимым, предохраняя его от изменений в будущем.

Рассмотрим два небольших примера.

1) char *p; p = (char *) ((int)p & pageoffset);

2) dword tmp = (dword)malloc(arraysize); ... int *ptr = (int *)tmp;

Оба примера не учитывают, что размер указателя может отличаться от 32 бит. Используется явное приведение типа, отбрасывающее старшие биты в указателе, что является явной ошибкой на 64-битной системе. Исправленные варианты, использующие для упаковки указателей целочисленные memsize типы (intptr_t и dword_ptr), приведены ниже:

1) char *p; p = (char *) ((intptr_t)p & pageoffset);

2) dword_ptr tmp = (dword_ptr)malloc(arraysize); ... int *ptr = (int *)tmp;

Опасность двух рассмотренных примеров в том, что сбой в программе может быть

обнаружен спустя очень большой промежуток времени. Программа может совершенно корректно работать с небольшим объемом данных на 64-битной системе, пока обрабатываемые адреса находятся в пространстве первых четырех гигабайт памяти. А затем, при запуске программы на больших производственных задачах, произойдет выделение памяти за пределами этой области. Рассмотренный в примерах код, обрабатывая указатель на объект вне данной области, приведет к неопределенному поведению программы.

Следующий приведенный код не будет таиться и проявит себя при первом выполнении:

void getbufferaddr(void **retptr) { ... // access violation on 64-bit system *retptr = p; } unsigned bufaddress; getbufferaddr((void **)&bufaddress);

Исправление также заключается в выборе типа, способного вмещать в себя указатель.

uintptr_t bufaddress; getbufferaddr((void **)&bufaddress); //ok

Бывают ситуации, когда упаковка указателя в 32-битный тип просто необходима. В основном такие ситуации возникают при необходимости работы со старыми api функциями. Для таких случаев следует прибегнуть к специальным функциям, таким как longtointptr, ptrtoulong и так далее.

Резюмируя, хочется заметить, что плохим стилем будет упаковка указателя в типы, всегда равные 64-битам. Показанный ниже код вновь придется исправлять с приходом 128-битных систем:

pvoid p; // bad style. the 128-bit time will come. __int64 n = __int64(p); p = pvoid(n);

# 7. memsize-типы в объединениях

Особенностью объединения является то, что для всех элементов - членов объединения выделяется одна и та же область памяти, то есть они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Следует внимательно отнестись к объединениям, имеющим в своем составе указатели и другие члены типа memsize.

Когда возникает необходимость работать с указателем как с целым числом, иногда удобно воспользоваться объединением, как показано в примере, и работать с числовым представлением типа без использования явных приведений:

union ptrnumunion { char *m_p; unsigned m_n; } u; u.m_p = str; u.m_n += delta;

Данный код корректен на 32-битных системах и некорректен на 64-битных. Изменяя член m_n на 64-битной системе, мы работаем только с частью указателя m_p. Следует использовать тип, который будет соответствовать размеру указателя:

union ptrnumunion { char *m_p; size_t m_n; //type fixed } u;

Другое частое использование объединения заключается в представлении одного члена, набором других более мелких. Например, нам может потребоваться разбить значение типа size_t на байты для реализации табличного алгоритма подсчета количества нулевых битов в байте:

union sizettobytesunion { size_t value; struct { unsigned char b0, b1, b2, b3; } bytes; } u; sizettobytesunion u; u.value = value; size_t zerobitsn = translatetable[u.bytes.b0] + translatetable[u.bytes.b1] + translatetable[u.bytes.b2] + translatetable[u.bytes.b3];

Здесь допущена принципиальная алгоритмическая ошибка, заключающаяся в

предположении, что тип size_t состоит из 4 байт. Возможность автоматического поиска алгоритмических ошибок пока вряд ли возможна, но мы можем осуществить поиск всех объединений и проверить наличие в них memsize типов. Найдя такое объединение, мы можем обнаружить алгоритмическую ошибку и переписать код следующим образом.

union sizettobytesunion { size_t value; unsigned char bytes[sizeof(value)]; } u; sizettobytesunion u; u.value = value; size_t zerobitsn = 0; for (size_t i = 0; i != sizeof(bytes); ++i) zerobitsn += translatetable[bytes[i]];

# 8. Изменение типа массива

Иногда в программах необходимо (или просто удобно) представлять элементы массива в виде элементов другого типа. Опасное и безопасное приведение типов представлено в следующем коде:

int array[4] = { 1, 2, 3, 4 }; enum enumbers { zero, one, two, three, four }; //safe cast (for msvc2005) enumbers *enumptr = (enumbers *)(array); cout << enumptr[1] << " "; //unsafe cast size_t *sizetptr = (size_t *)(array); cout << sizetptr[1] << endl; //output on 32-bit system: 2 2 //output on 64 bit system: 2 17179869187

Как видите, результат вывода программы отличается в 32-битном и 64-битном варианте. На 32-битной системе доступ к элементам массива осуществляется корректно, так как размеры типов size_t и int совпадают, и мы видим вывод "2 2".

На 64-битной системе мы получили в выводе "2 17179869187", так как именно значение 17179869187 находится в 1-ом элементе массива sizetptr (см. рисунок 3). В некоторых случаях именно такое поведение и бывает нужно, но обычно это является ошибкой.

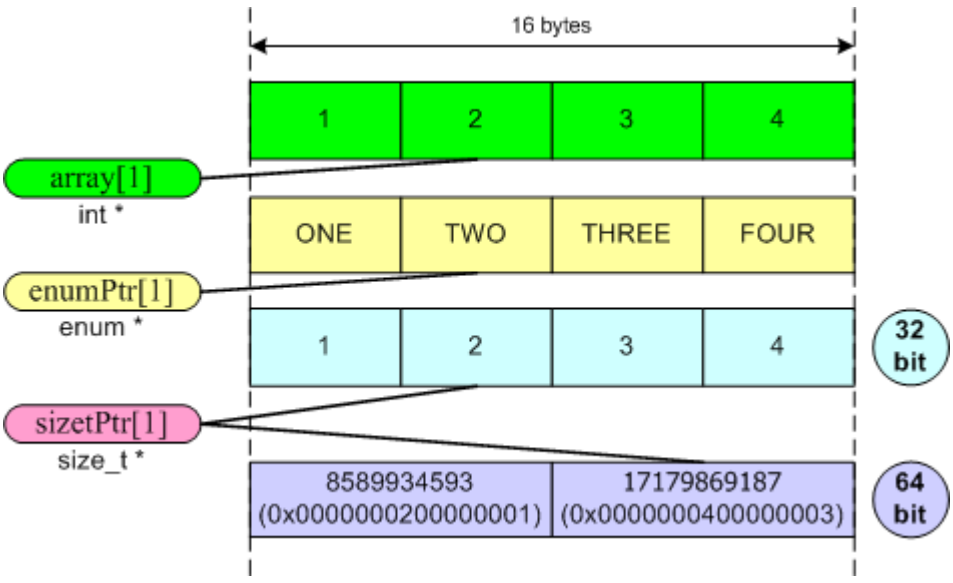



*Рисунок 3. Расположение элементов массивов в памяти.*

Исправление описанной ситуации заключается в отказе от опасных приведений типов путем модернизации программы. Другим вариантом является создание нового массива и копирование в него значений из исходного массива.

# 9. Виртуальные функции с аргументами типа memsize

Если у Вас в программе имеются большие иерархии наследования классов с виртуальными функциями, то существует вероятность использования по невнимательности аргументов различных типов, но которые фактически совпадают на 32-битной системе. Например, в базовом классе Вы используете в качестве аргумента виртуальной функции тип size_t, а в наследнике - тип unsigned. Соответственно, на 64-битной системе этот код будет некорректен.

Такая ошибка не обязательно кроется в сложных иерархиях наследования, и вот один из примеров:

class cwinapp { ... virtual void winhelp(dword_ptr dwdata, uint ncmd); }; class csampleapp : public cwinapp { ... virtual void winhelp(dword dwdata, uint ncmd); };

Проследим жизненный цикл разработки некоторого приложения. Пусть первоначально оно разрабатывалось под microsoft visual c++ 6.0., когда функция winhelp в классе cwinapp имела следующий прототип:

virtual void winhelp(dword dwdata, uint ncmd = help_context);

Совершенно верно было осуществить перекрытие виртуальной функции в классе csampleapp, как показано в примере. Затем проект был перенесен в microsoft visual c++ 2005, где прототип функции в классе cwinapp претерпел изменения, заключающиеся в смене типа dword на тип dword_ptr. На 32-битной системе программа продолжит совершенно корректно работать, так как здесь типы dword и dword_ptr совпадают. Неприятности проявят себя при компиляции данного кода под 64-битную платформу. Получатся две функции с одинаковыми именами, но с различными параметрами, в результате чего перестанет вызываться пользовательский код.

Исправление заключается в использовании одинаковых типов в соответствующих виртуальных функциях.

class csampleapp : public cwinapp { ... virtual void winhelp(dword_ptr dwdata, uint ncmd); };

# 10. Сериализация и обмен данными

Важным элементом переноса программного решения на новую платформу является преемственность к существующим протоколам обмена данных. Необходимо обеспечить чтение существующих форматов проектов, осуществлять обмен данными между 32-битными и 64-битными процессами и так далее.

В основном, ошибки данного рода заключаются в сериализации memsize типов и операциях обмена данными с их использованием:

1) size_t pixelcount; fread(&pixelcount, sizeof(pixelcount), 1, infile);

2) __int32 value_1; ssize_t value_2; inputstream >> value_1 >> value_2;

3) time_t time; packtobuffer(memorybuf, &time, sizeof(time));

Во всех приведенных примерах имеются ошибки двух видов: использование типов непостоянной размерности в бинарных интерфейсах и игнорирование порядка байт.

**Использование типов непостоянной размерности.** Недопустимо использование типов, которые меняют свой размер в зависимости от среды разработки, в бинарных интерфейсах обмена данными. В языке Си++ все типы не имеют четкого размера и, следовательно, их все невозможно использовать для этих целей. Поэтому создатели средств разработки и сами программисты создают типы данных, имеющие строгий размер, такие как __int8, __int16,

int32, word64 и так далее.

Использование подобных типов обеспечивает переносимость данных между программами на различных платформах, хотя и требует дополнительных усилий. Три показанных примера написаны неаккуратно, что даст о себе знать при смене разрядности некоторых типов данных с 32-бит до 64-бит. Учитывая необходимость поддержки старых форматов данных, исправление может выглядеть следующим образом:

1) size_t pixelcount; __uint32 tmp; fread(&tmp, sizeof(tmp), 1, infile); pixelcount = static_cast(tmp);

2) __int32 value_1; __int32 value_2; inputstream >> value_1 >> value_2;

3) time_t time; __uint32 tmp = static_cast<__uint32>(time); packtobuffer(memorybuf, &tmp, sizeof(tmp));

Но приведенный вариант исправления может являться не лучшим. При переходе на 64-битную систему программа может обрабатывать большее количество данных, и использование в данных 32-битных типов может стать существенным препятствием. В таком случае, можно оставить старый код для совместимости со старым форматом данных, исправив некорректные типы. И реализовать новый бинарный формат данных уже с учетом допущенных ошибок. Еще одним вариантом может стать отказ от бинарных форматов и переход на текстовый формат или другие форматы, предоставляемые различными библиотеками.

**Игнорирование порядка байт (byte order)**. Даже после внесения исправлений, касающихся размеров типа, Вы можете столкнуться с несовместимостью бинарных форматов. Причина кроется в ином представлении данных. Наиболее часто это связано с другой последовательностью байт.

Порядок байт - метод записи байтов многобайтовых чисел (см. также рисунок 4). Порядок от младшего к старшему (англ. little-endian) - запись начинается с младшего и заканчивается старшим. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами. Порядок от старшего к младшему (англ. big-endian) - запись начинается со старшего и заканчивается младшим. Этот порядок является стандартным для протоколов tcp/ip. Поэтому, порядок байтов от старшего к младшему часто называют сетевым порядком байтов (англ. network byte order). Этот порядок байт используется процессорами motorola 68000, sparc.

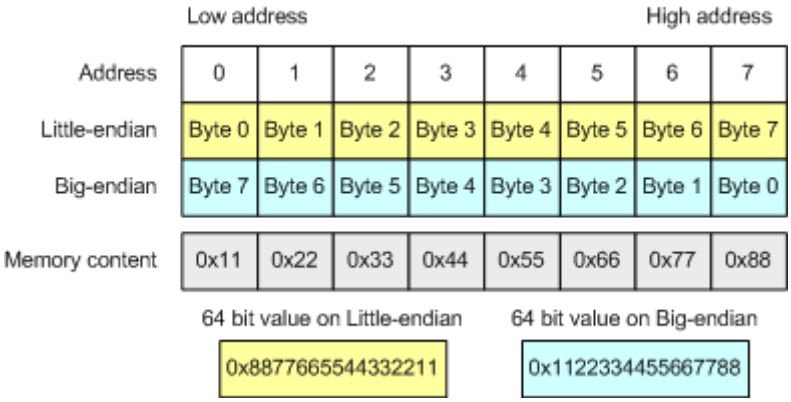



*Рисунок 4. Порядок байт в 64-битном типе на little-endian и big-endian системах.*

Разрабатывая бинарный интерфейс или формат данных, следует помнить о последовательности байт. А если 64-битная система, на которую Вы переносите 32-битное приложение, имеет иную последовательность байт, то Вы просто будете вынуждены учесть это в своем коде. Для преобразования между сетевым порядком байт (big-endian) и порядком байт (little-endian), можно использовать функции htonl(), htons(), bswap_64, и так далее.

# 11. Битовые поля

Если Вы используете битовые поля, то необходимо учитывать, что использование memsize типов повлечет изменение размеров структур и выравнивания. Например, приведенная далее структура будет иметь размер 4 байта на 32-битной системе и 8 байт на 64-битной системе:

struct mystruct { size_t r : 5; };

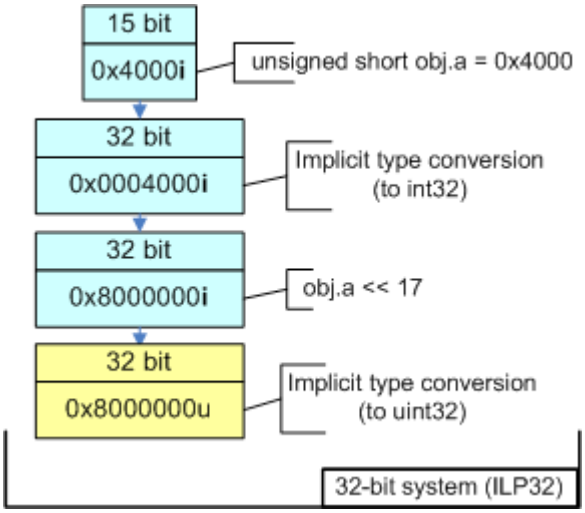
Но на этом ваша внимательность к битовым полям ограничиваться не должна. Рассмотрим тонкий пример:

struct bitfieldstruct { unsigned short a:15; unsigned short b:13; }; bitfieldstruct obj; obj.a = 0x4000; size_t addr = obj.a << 17; //sign extension printf("addr 0x%ix\n", addr); //output on 32-bit system: 0x80000000 //output on 64-bit system: 0xffffffff80000000

Обратите внимание, если приведенный пример скомпилировать для 64-битной системы, то в выражении "addr = obj.a << 17;" будет присутствовать знаковое расширение, несмотря на то, что обе переменные addr и obj.a являются беззнаковыми. Это знаковое расширение обусловлено правилами приведения типов, которые применяются следующим образом (см. также рисунок 5):

Член структуры obj.a преобразуется из битового поля типа unsigned short в int. Мы получаем тип int, а не unsigned int из-за того, что 15-битное поле помещается в 32-битное знаковое целое.

Выражение "obj.a << 17" имеет тип int, но оно преобразуется в ptrdiff_t и затем в size_t, перед тем как будет присвоено переменной addr. Знаковое расширение происходит в момент совершения преобразования из int в ptrdiff_t.



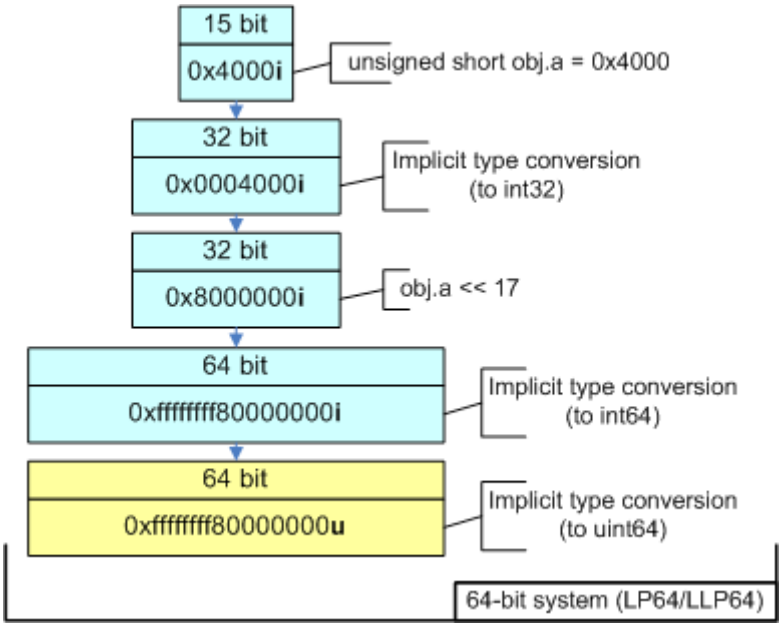



*Рисунок 5. Вычисление выражения на различных системах.*

Так что будьте внимательны при работе с битовыми полями. Для предотвращения описанной ситуации в нашем примере нам достаточно явно привести obj.a к типу size_t.

... size_t addr = size_t(obj.a) << 17; printf("addr 0x%ix\n", addr); //output on 32-bit system: 0x80000000 //output on 64-bit system: 0x80000000

# 12. Адресная арифметика с указателями

Пример первый:

unsigned short a16, b16, c16; char *pointer; : pointer += a16 * b16 * c16;

Данный пример корректно работает с указателями, если значение выражения "a16 * b16 * c16" не превышает uint_max (4gb). Такой код мог всегда корректно работать на 32-битной платформе, так как программа никогда не выделяла массивов больших размеров. На 64-битной архитектуре размер массива превысил uint_max элементов. Допустим, мы хотим сдвинуть значение указателя на 6.000.000.000 байт, и поэтому переменные a16, b16 и c16 имеют значения 3000, 2000 и 1000 соответственно. При вычислении выражения "a16 * b16 * c16" все переменные, согласно правилам языка Си++, будут приведены к типу int, а уже затем будет произведено их умножение. В ходе выполнения умножения произойдет переполнение. Некорректный результат выражения будет расширен до типа ptrdiff_t и произойдет некорректное вычисление указателя.

Следует старательно избегать возможных переполнений в арифметике с указателями. Для этого лучше всего использовать memsize типы или явное приведение типов в выражениях, где присутствуют указатели. Используя явное приведение типов, мы можем переписать наш код следующим образом:

short a16, b16, c16; char *pointer; : pointer += static_cast(a16) * static_cast(b16) * static_cast(c16);

Если Вы думаете, что злоключения ждут неаккуратные программы только на больших объемах данных, то мы вынуждены Вас огорчить. Рассмотрим интересный код для работы с массивом, содержащим всего 5 элементов. Второй пример работоспособен в 32-битном варианте и не работоспособен в 64-битном:

int a = -2; unsigned b = 1; int array[5] = { 1, 2, 3, 4, 5 }; int *ptr = array + 3; ptr = ptr + (a + b); //invalid pointer value on 64-bit platform printf("%i\n", *ptr); //access violation on 64-bit platform

Давайте проследим, как происходит вычисление выражения "ptr + (a + b)":

- Согласно правилам языка Си++ переменная a типа int приводится к типу unsigned.
- Происходит сложение a и b. В результате мы получаем значение 0xffffffff типа unsigned.

Затем происходит вычисление выражения "ptr + 0xffffffffu", но что из этого выйдет, будет зависеть от размера указателя на данной архитектуре. Если сложение будет происходить в 32-битной программе, то данное выражение будет эквивалентно "ptr - 1" и мы успешно распечатаем число 3.

В 64-битной программе к указателю честным образом прибавится значение 0xffffffffu, в результате чего указатель окажется далеко за пределами массива. И при доступе к элементу по данному указателю нас ждут неприятности.

Для предотвращения показанной ситуации, как и в первом случае, рекомендуем использовать в арифметике с указателями только memsize-типы. Два варианта исправления кода:

ptr = ptr + (ptrdiff_t(a) + ptrdiff_t(b));

ptrdiff_t a = -2; size_t b = 1; ... ptr = ptr + (a + b);

Вы можете возразить и предложить следующий вариант исправления:

int a = -2; int b = 1; ... ptr = ptr + (a + b);

Да, такой код будет работать, но он плох по ряду причин:

- Он будет приучать к неаккуратной работе с указателями. Через некоторое время Вы можете забыть нюансы и по ошибке вновь сделать одну из переменных типа unsigned.
- Использование не memsize-типов совместно с указателями потенциально опасно. Допустим, что в выражении с указателем участвует переменная delta типа int. И это выражение совершенно корректно. Но ошибка может укрыться в вычислении самой переменной delta, так как 32-бит может не хватить для необходимых вычислений, при работе с большими массивами данных. Использование memsize-типа для переменной delta автоматически устраняет такую опасность.

# 13. Индексация массивов

Данная разновидность ошибок выделена для лучшей структуризации изложения, так как индексация в массивах с использованием квадратных скобок - это всего лишь иная запись адресной арифметики, рассмотренной выше.

В программировании на языке Си, а затем и Си++ сложилась практика использования в конструкциях следующего вида переменные типа int/unsigned:

unsigned index = 0; while (mybignumberfield[index] != id) index++;

Но время идет, и все меняется. И вот теперь пришло время сказать: "Больше так не делайте! Используйте для индексации (больших) массивов только memsize-типы."

Приведенный код не сможет обработать в 64-битной программе массив, содержащий более uint_max элементов. После доступа к элементу с индексом uint_max произойдет переполнение переменной index и мы получим вечный цикл.

Чтобы окончательно убедить Вас в необходимости использования только memsize типов для индексации и в выражениях адресной арифметики, приведем последний пример.

class region { float *array; int width, height, depth; float region::getcell(int x, int y, int z) const; ... }; float region::getcell(int x, int y, int z) const { return array[x + y * width + z * width * height]; }

Данный код взят из реальной программы математического моделирования, в которой важным ресурсом является объем оперативной памяти, и возможность на 64-битной архитектуре использовать более 4 гигабайт памяти существенно увеличивает вычислительные возможности. В программах данного класса для экономии памяти часто используют одномерные массивы, осуществляя работу с ними как с трехмерными массивами. Для этого существуют функции, аналогичные getcell, обеспечивающие доступ к необходимым элементам. Но приведенный код будет корректно работать только с массивами, содержащими менее int_max элементов. Причина - использование 32-битных типов int для вычисления индекса элемента.

Программисты часто допускают ошибку, пытаясь исправить код следующим образом:

float region::getcell(int x, int y, int z) const { return array[static_cast(x) + y * width + z * width * height]; }

Они знают, что по правилам языка Си++ выражение для вычисления индекса будет иметь тип ptrdiff_t и надеются за счет этого избежать переполнения. Но переполнение может произойти внутри подвыражения "y * width" или "z * width * height", так как для их вычисления по-прежнему используется тип int.

Если Вы хотите исправить код, не изменяя типов переменных, участвующих в выражении, то Вы можете явно привести каждую переменную к memsize типу:

float region::getcell(int x, int y, int z) const { return array[ptrdiff_t(x) + ptrdiff_t(y) * ptrdiff_t(width) + ptrdiff_t(z) * ptrdiff_t(width) * ptrdiff_t(height)]; }

Другое решение - изменить типы переменных на memsize тип:

typedef ptrdiff_t tcoord; class region { float *array; tcoord width, height, depth; float region::getcell(tcoord x, tcoord y, tcoord z) const; ... }; float region::getcell(tcoord x, tcoord y, tcoord z) const { return array[x + y * width + z * width * height]; }

# 14. Смешанное использование простых целочисленных типов и memsize-типов

Смешанное использование memsize- и не memsize-типов в выражениях может приводить к некорректным результатам на 64-битных системах и быть связано с изменением диапазона входных значений. Рассмотрим ряд примеров:

size_t count = bigvalue; for (unsigned index = 0; index != count; ++index) { ... }

Это пример вечного цикла, если count > uint_max. Предположим, что на 32-битных системах этот код работал с диапазоном менее uint_max итераций. Но 64-битный вариант программы может обрабатывать больше данных и ему может потребоваться большее количество итераций. Поскольку значения переменной index лежат в диапазоне [0..uint_max], то условие "index != count" никогда не выполнится, что и приводит к бесконечному циклу.

Другая частая ошибка - запись выражений следующего вида:

int x, y, z; intptr_t sizevalue = x * y * z;

Ранее уже рассматривались подобные примеры, когда при вычислении значений с использованием не memsize-типов происходило арифметическое переполнение. И конечный результат был некорректен. Поиск и исправление приведенного кода осложняется тем, что компиляторы, как правило, не выдают на него никаких предупреждений. С точки зрения языка Си++ это совершенно корректная конструкция. Происходит умножение нескольких переменных типа int, после чего результат неявно расширяется до типа intptr_t и происходит присваивание.

Приведем небольшой код, показывающий опасность неаккуратных выражений со смешанными типами (результаты получены с использованием microsoft visual c++ 2005, 64-битный режим компиляции):

int x = 100000; int y = 100000; int z = 100000; intptr_t size = 1; // result: intptr_t v1 = x * y * z; // -1530494976 intptr_t v2 = intptr_t(x) * y * z; // 1000000000000000 intptr_t v3 = x * y * intptr_t(z); // 141006540800000 intptr_t v4 = size * x * y * z; // 1000000000000000 intptr_t v5 = x * y * z * size; // -1530494976 intptr_t v6 = size * (x * y * z); // -1530494976 intptr_t v7 = size * (x * y) * z; // 141006540800000 intptr_t v8 = ((size * x) * y) * z; // 1000000000000000 intptr_t v9 = size * (x * (y * z)); // -1530494976

Необходимо, чтобы все операнды в подобных выражениях были заранее приведены к типу большей разрядности. Помните, что выражение вида

intptr_t v2 = intptr_t(x) * y * z;

вовсе не гарантирует правильный результат. Оно гарантирует только то, что выражение "intptr_t(x) * y * z" будет иметь тип intptr_t. Правильный результат, показанный этим выражением в примере, не более чем везение, обусловленное конкретной версией компилятора и фазой Луны.

Порядок вычисления выражения с операторами одинакового приоритета 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.

Следовательно, если результатом выражения должен являться memsize-тип, то в выражении должны участвовать только memsize-типы. Или элементы, приведенные к memsize-типам. Правильный вариант:

intptr_t v2 = intptr_t(x) * intptr_t(y) * intptr_t(z); // ok!

**ПРИМЕЧАНИЕ**

Примечание. Если у Вас много целочисленных вычислений и контроль над переполнениями для Вас является важной задачей, то мы предлагаем обратить Ваше внимание на класс safeint, реализацию и описание которого можно найти в msdn.

Смешанное использование типов может проявляться и в изменении программной логики:

ptrdiff_t val_1 = -1; unsigned int val_2 = 1; if (val_1 > val_2) printf ("val_1 is greater than val_2\n"); else printf ("val_1 is not greater than val_2\n"); //output on 32-bit system: "val_1 is greater than val_2" //output on 64-bit system: "val_1 is not greater than val_2"

На 32-битной системе переменная val_1 согласно правилам языка Си++ расширялась до типа unsigned int и становилась значением 0xffffffffu. В результате условие "0xffffffffu > 1"

выполнялось. На 64-битной системе наоборот расширяется переменная val_2 до типа ptrdiff_t. В этом случае уже проверяется выражение "-1 > 1". На рисунке 6 схематично отображены происходящие преобразования.

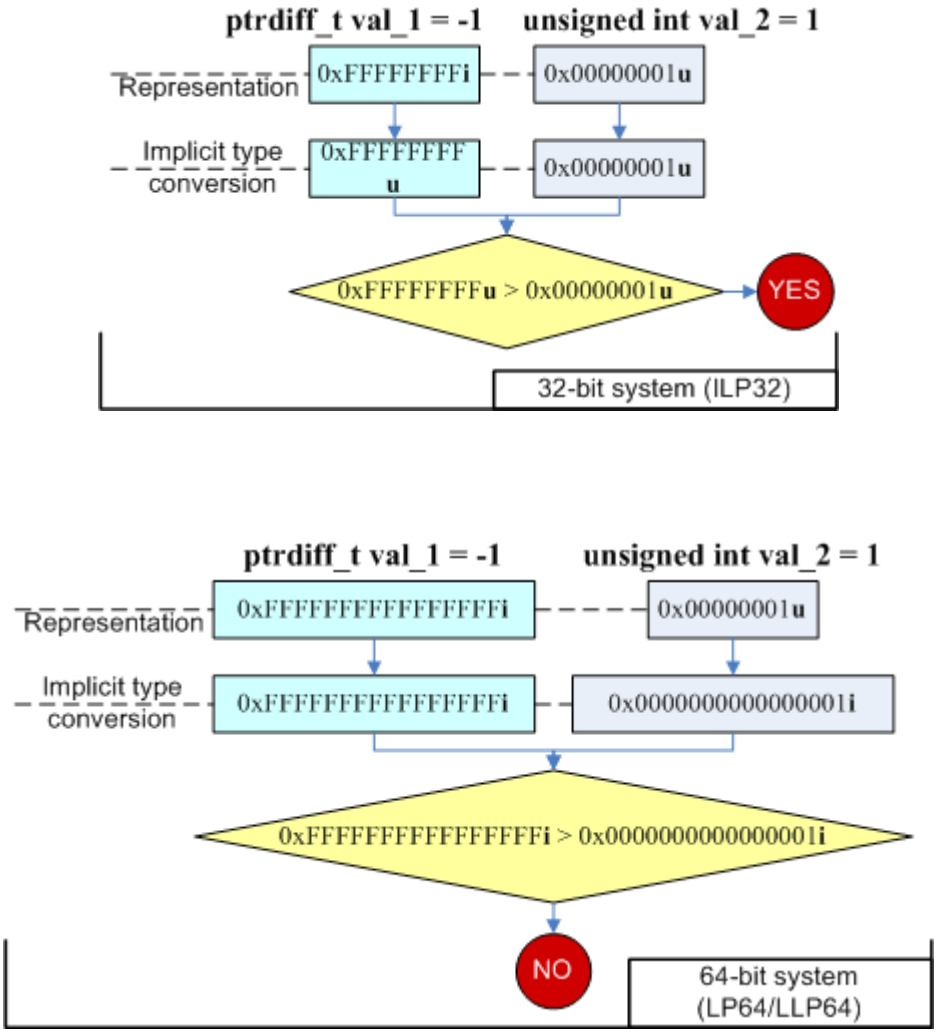



*Рисунок 6. Преобразования, происходящие в выражении.*

Если Вам необходимо вернуть прежнее поведение кода - следует изменить тип переменной val_2:

ptrdiff_t val_1 = -1; size_t val_2 = 1; if (val_1 > val_2) printf ("val_1 is greater than val_2\n"); else printf ("val_1 is not greater than val_2\n");

# 15. Неявные приведения типов при использовании функций

Рассматривая предыдущий класс ошибок, связанный со смешиванием простых целочисленных типов и memsize-типов, мы рассматривали только простые выражения. Но аналогичные проблемы могут проявиться и при использовании других конструкций языка Си++:

extern int width, height, depth; size_t getindex(int x, int y, int z) { return x + y * width + z * width *

height; } ... myarray[getindex(x, y, z)] = 0.0f;

В случае работы с большими массивами (более int_max элементов) данный код будет вести себя некорректно, и мы будем адресоваться не к тем элементам массива myarray, к которым рассчитываем. Несмотря на то, что мы возвращаем значение типа size_t, выражение "x + y * width + z * width * height" вычисляется с использованием типа int. Мы думаем, Вы уже догадались, что исправленный код будет выглядеть следующим образом:

extern int width, height, depth; size_t getindex(int x, int y, int z) { return (size_t)(x) + (size_t)(y) * (size_t)(width) + (size_t)(z) * (size_t)(width) * (size_t)(height); }

В следующем примере, у нас вновь смешивается memsize-тип (указатель) и простой тип unsigned:

extern char *begin, *end; unsigned getsize() { return end - begin; }

Результат выражения "end - begin" имеет тип ptrdiff_t. Поскольку функция возвращает тип unsigned, то происходит неявное приведение типа, при котором старшие биты результата теряются. Таким образом, если указатели begin и end ссылаются на начало и конец массива, по размеру большего uint_max (4gb), то функция вернет некорректное значение.

И еще один пример. На этот раз рассмотрим не возвращаемое значение, а формальный аргумент функции:

void foo(ptrdiff_t delta); int i = -2; unsigned k = 1; foo(i + k);

Этот код не напоминает Вам пример с некорректной арифметикой указателей, рассмотренный ранее? Да, здесь происходит то же самое. Некорректный результат возникает при неявном расширении фактического аргумента, имеющего значение 0xffffffff и тип unsigned, до типа ptrdiff_t.

# 16. Перегруженные функции

При переносе 32-битных программ на 64-битную платформу может наблюдаться изменение логики ее работы, связанное с использованием перегруженных функций. Если функция перекрыта для 32-битных и 64-битных значений, то обращение к ней с аргументом типа memsize будет транслироваться в различные вызовы на различных системах. Этот прием может быть полезен, как, например, в приведенном коде:

static size_t getbitcount(const unsigned __int32 &) { return 32; } static size_t getbitcount(const unsigned __int64 &) { return 64; } size_t a; size_t bitcount = getbitcount(a);

Но такое изменение логики хранит в себе опасность. Представьте себе программу, где для каких-то целей используется класс для организации стека. Особенность этого класса в том, что он позволяет хранить значение различных типов:

class mystack { ... public: void push(__int32 &); void push(__int64 &); void pop(__int32 &); void pop(__int64 &); } stack; ptrdiff_t value_1; stack.push(value_1); ... int value_2; stack.pop(value_2);

Неаккуратный программист помещал и затем выбирал из стека значения различных типов (ptrdiff_t и int). На 32-битной системе их размеры совпадали, все замечательно работало. Когда в 64-битной программе изменился размер типа ptrdiff_t, то в стек стало попадать больше байт, чем затем извлекаться.

Думаем, что Вам понятен данный класс ошибок, и как внимательно следует относиться к вызову перегруженных функций, передавая фактические аргументы типа memsize.

# 17. Выравнивание данных

Процессоры работают эффективнее, когда имеют дело с правильно выровненными данными. Как правило, 32-битный элемент данных должен быть выровнен по границе, кратной 4 байт, а 64-битный элемент - по границе 8 байт. Попытка работать с не выровненными данными на процессорах ia-64 (itanium), как показано в следующем примере, приведет к возникновению исключения:

#pragma pack (1) // also set by key /zp in msvc struct alignsample { unsigned size; void *pointer; } object; void foo(void *p) { object.pointer = p; // alignment fault }

Если Вы вынуждены работать с не выровненными данными на itanium, то следует явно указать это компилятору. Например, воспользоваться специальным макросом unaligned:

#pragma pack (1) // also set by key /zp in msvc struct alignsample { unsigned size; void *pointer; } object; void foo(void *p) { *(unaligned void *)&object.pointer = p; //very slow }

Такое решение неэффективно, так как доступ к не выровненным данным будет происходить в несколько раз медленнее. Лучшего результата можно достичь, располагая в 64-битные элементы данных до 32,16 и 8-битных элементов.

На архитектуре x64, при обращении к не выровненным данным, исключения не возникает, но их также следует избегать. Во-первых, из-за существенного замедления скорости доступа к таким данным, а во-вторых, из-за высокой вероятности переноса программы в будущем на платформу ia-64.

Рассмотрим еще один пример кода, не учитывающий выравнивание данных:

struct mypointersarray { dword m_n; pvoid m_arr[1]; } object; ... malloc( sizeof(dword) + 5 * sizeof(pvoid) ); ...

Если мы хотим выделить объем памяти, необходимый для хранения объекта типа mypointersarray, содержащего 5 указателей, то мы должны учесть, что начало массива m_arr будет выровнено по границе 8 байт. Расположение данных в памяти на разных системах (win32/win64) показано на рисунке 7.

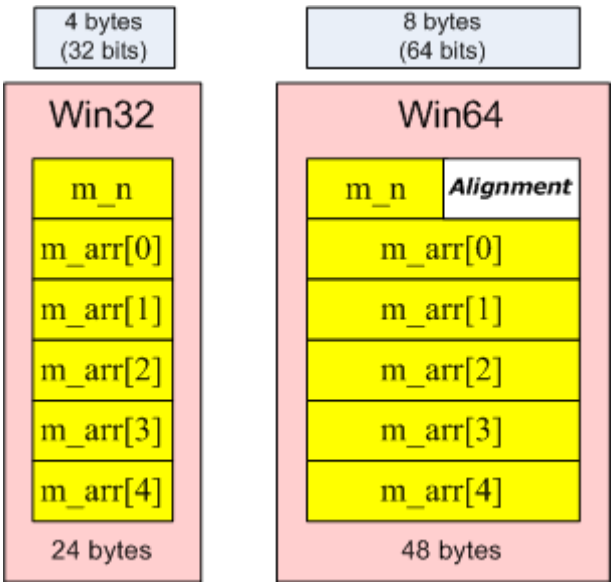



*Рисунок 7. Выравнивание данных в памяти на системах win32 и win64*

Корректный расчет размера должен выглядеть следующим образом:

struct mypointersarray { dword m_n; pvoid m_arr[1]; } object; ... malloc( field_offset(struct

mypointersarray, m_arr) + 5 * sizeof(pvoid) ); ...

В приведенном коде мы узнаем смещение последнего члена структуры и суммируем это смещение с его размером. Смещение члена структуры или класса можно узнать с использованием макроса offsetof или field_offset.

Всегда используйте эти макросы для получения смещения в структуре, не опираясь на Ваше знание размеров типов и выравнивания. Пример кода с правильным вычислением адреса члена структуры:

struct tfoo { dword_ptr whatever; int value; } object; int *valueptr = (int *)((size_t)(&object) + offsetof(tfoo, value)); // ok

# 18. Исключения.

Генерирование и обработка исключений с участием целочисленных типов не является хорошей практикой программирования на языке Си++. Для этих целей следует использовать более информативные типы, например классы, производные от классов std::exception. Но иногда все-таки приходится работать с менее качественным кодом, таким как показано ниже:

char *ptr1; char *ptr2; try { try { throw ptr2 - ptr1; } catch (int) { std::cout << "catch 1: on x86" << std::endl; } } catch (ptrdiff_t) { std::cout << "catch 2: on x64" << std::endl; }

Следует тщательно избегать генерирования или обработку исключений с использованием memsize-типов, так как это чревато изменением логики работы программы. Исправление данного кода может заключаться в замене "catch (int)" на "catch (ptrdiff_t)". А более правильным решением будет использование специального класса для передачи информации о возникшей ошибке.

# 19. Использование устаревших функций и предопределенных констант.

Разрабатывая 64-битное приложение, помните об изменениях среды, в которой оно теперь будет выполняться. Часть функций станут устаревшими, их будет необходимо изменить на обновленные варианты. Примером такой функции в ОС windows будет getwindowlong. Обратите внимание на константы, относящиеся к взаимодействию со средой, в которой выполняется программа. В windows подозрительными будут являться строки, содержащие "system32" или "program files".

# 20. Явные приведения типов

Будьте аккуратны с явными приведениями типов. Они могут изменить логику выполнения программы при изменении разрядности типов или спровоцировать потерю значащих битов. Привести типовые примеры ошибок, связанных с явным приведением типов сложно, так как они очень разнообразны и специфичны для разных программ. С некоторыми из ошибками, связанными с явным приведением типов, Вы уже познакомились ранее.

# Диагностика ошибок

Диагностика ошибок, возникающих при переносе 32-битных программ на 64-битные системы, является непростой задачей. Перенос недостаточно качественного кода, разработанного без учета особенностей других архитектур, может потребовать много времени и усилий. Поэтому уделим немного внимания описанию методик и средств, которые смогут упростить эту задачу.

# Юнит-тестирование

Юнит-тестирование (англ. *unit test*) давно завоевало заслуженное уважение среди программистов. Юнит-тесты помогут проверить корректность программы после переноса на новую платформу. Но тут есть одна тонкость, о которой Вы должны помнить.

Юнит-тестирование может не позволить Вам проверить новые диапазоны входных значений, которые становятся доступны на 64-битных системах. Юнит-тесты классически разрабатываются таким образом, чтобы по возможности проходить за минимальное время. И та функция, которая обычно работает с массивом размером в десятки мегабайт, в юнит-тестах, скорее всего, будет обрабатывать десятки килобайт. Это обоснованно, так как эта функция в тестах может вызваться много раз с различными наборами входных значений. Но вот перед нами 64-битный вариант программы. И рассматриваемая функция теперь обрабатывает уже более 4 гигабайт данных. Соответственно, возникает необходимость увеличения входного размера массива и в тестах до размеров более 4 гигабайт. Проблема в том, что время прохождения тестов в таком случае увеличится на несколько порядков.

Поэтому, модифицируя наборы тестов, помните о компромиссе между скоростью выполнения юнитестов и полнотой проверок. К счастью, убедиться в работоспособности Ваших приложений могут помочь другие методики.

# Просмотр кода

Просмотр кода (англ. *code review*) - самая лучшая методика поиска ошибок и улучшения кода. Совместный тщательный просмотр кода может полностью избавить программу от ошибок, связанных с особенностями разработки 64-битных приложений. Естественно, вначале следует узнать, какие именно ошибки следует искать, иначе просмотр может не дать положительных результатов. Для этого необходимо заранее ознакомиться с этой и другими статьями, посвященными переносу программ с 32-битных систем на 64-битные. Ряд интересных ссылок по данной тематике Вы можете найти в конце статьи.

Но у этого подхода к анализу исходного кода есть один существенный недостаток. Он требует очень большого количества времени, из-за чего практически неприменим на больших проектах.

Компромиссом является использование статических анализаторов. Статический анализатор можно рассматривать как автоматизированную систему просмотра кода, где для программиста создается выборка потенциально опасных мест для проведения им дальнейшего анализа.

Но в любом случае желательно провести несколько просмотров кода, с целью совместного обучения команды поиску новых разновидностей ошибок, проявляющих себя на 64-битных системах.

# Встроенные средства компиляторов

Часть задач с поиском дефектного кода позволяют решать компиляторы. В них часто бывают встроены различные механизмы для диагностики рассматриваемых нами ошибок. Например, в microsoft visual c++ 2005 Вам могут быть полезны следующие ключи: /wp64, /wall, а в sunstudio c++ ключ -xport64.

К сожалению, предоставляемые ими возможности часто недостаточны и не стоит полагаться только на них. Но в любом случае крайне рекомендуется включить соответствующие опции компилятора для диагностики ошибок в 64-битном коде.

# Статические анализаторы

Статические анализаторы - прекрасное средство повышения качества и надежности программного кода. Основная сложность, связанная с использованием статических анализаторов заключается в том, что они генерируют довольно много ложных сообщений о потенциальных ошибках. Программисты, будучи по натуре ленивыми, используют этот аргумент, чтобы так или иначе не заниматься исправлением найденных ошибок. В microsoft эта проблема решается безусловным внесением обнаруженных ошибок в bug tracking систему. Тем самым у программиста не остается выбора между исправлением кода и попытками избежать этого.

Мы считаем, что такие жесткие правила оправданы. Выигрыш от качественного кода существенно покрывает издержки времени на статический анализ и соответствующую модификацию кода. Выигрыш достигается за счет облегчения поддержки кода и уменьшения сроков отладки и тестирования.

Статические анализаторы могут с успехом использоваться для диагностики многих из рассмотренных в статье классов ошибок.

Авторам известны 3 статических анализатора, которые заявляют о наличии средств диагностирования ошибок, связанных с переносом программ на 64-битные системы. Хотим сразу предупредить, что мы можем заблуждаться по поводу возможностей, которыми они обладают, тем более что это развивающиеся продукты и новые версии могут иметь большую функциональность.

1. gimpel software pc-lint (http://www.gimpel.com). Данный анализатор обладает широким списком поддерживаемых платформ и является статическим анализатором общего назначения. Он позволяет выявлять ошибки при переносе программ на архитектуру с моделью данных lp64. Преимуществом является возможность построения жесткого контроля над преобразованиями типов. К недостаткам можно отнести отсутствие среды, но это можно исправить, используя стороннюю оболочку riverblade visual lint.
2. parasoft c++test (http://www.parasoft.com/). Другой известный статический анализатор общего назначения. Также существует под большое количество аппаратных и программных платформ. Имеет встроенную среду, существенно облегчающую работу и настройку правил анализа. Как и pc-lint он рассчитан на модель данных lp64.
3. viva64 (http://www.viva64.com). В отличие от других анализаторов рассчитан на модель данных windows (llp64). Интегрируется в среду разработки visual studio 2005. Предназначен только для диагностики проблем, связанных с переносом программ на 64-битные системы, что существенно упрощает его настройку.