20 issues of porting C++ code to the 64-bit platform


Andrey Karpov (/en/b/a/andrey-karpov)
Articles: 327


Evgeniy Ryzhkov (/en/b/a/evgeniy-ryzhkov)
Articles: 109

01.03.2007

m/Code_Analysis)

)urner.com/viva64-

Contents

Abstract

Program errors occurring while porting C++ code from 32-bit platforms to 64-bit are observed. Examples of the incorrect code, and the ways to correct it are given. Methods and means of the code analysis which allow diagnosis of the errors discussed, are listed.

This article contains various examples of 64-bit errors. However, we have learned of many more examples, and types of errors, since we started writing the article, which are not included here. Please see the article "A Collection of Examples of 64-bit Errors in Real Programs (/en/a/0065/)" which covers defects in 64-bit programs we know of most. We also recommend that you study the course "Lessons on development of 64-bit C/C++ applications (/en/l/)", where we describe the methodology for creating correct 64-bit code, and searching for all types of defects using the Viva64 code analyzer.

Introduction

This article describes the process of porting a 32-bit application to 64-bit (/en/t/0001/) systems. The article is written for programmers who use C++ but it may also be useful for all who face the problem of porting applications onto other platforms. The authors of the article are experts in the field of porting applications to 64-bit systems, and the developers of Viva64 (/en/viva64-tool/) tool, which facilitates the searching of errors in 64-bit applications.

One should understand that the new class of errors, which appear while developing 64-bit programs, is not just some new incorrect constructions among thousands of others. These are inevitable difficulties which the developers of any developing program will face. This article will help you to prepare for such difficulties, and will show ways to overcome them. Besides the advantages, any new technologies (in programming and other spheres as well) carry with them some limitations, and problems can be encountered when using these new technologies. The same situation can be observed in the sphere of 64-bit software developing. We all know that 64-bit software is the next step in information technologies development. But in reality, only a few programmers have faced the nuances of this sphere, and developing 64-bit programs in particular.

We won't dwell on the advantages which the use of 64-bit architecture provides. There are a lot of publications devoted to this theme, and the reader can find them easily.

The aim of this article, is to observe thoroughly the problems which can be faced by a developer of 64-bit programs. In this article you will learn about:

- typical programming errors which occur on 64-bit systems;
- the causes of these errors, with the corresponding examples;
- methods of error correction;
- review of methods and means of searching errors in 64-bit programs.

The information given will allow you to:

- find out the differences between 32-bit and 64-bit systems;
- avoid errors while writing code for 64-bit systems;
- speed up the process of migrating a 32-bit application to a 64-bit architecture via reducing the amount of time necessary for debugging and testing;
- forecast the amount of time necessary to port the code to the 64-bit system more accurately and seriously.

This article contains a lot of examples which you should try in the programming environment for better a better understanding of their functions. Going into them will give you more than just a set of separate elements. You will open the door into the world of 64-bit systems.

To make the following text easier to understand, let's remember some types we can face. (see table N1).

Type name	Type size (32-bit system)	Type size (64-bit system)	Description
ptrdiff_t	32	64	Signed integer type which appears after subtraction of two pointers. This type is used to keep memory sizes. Sometimes it is used as the result of function returning size or -1 if an error occurs.
size_t	32	64	Unsigned integer type. Data of this bype is returned by the sizeof() operator. This type is used to keep size or number of objects.
intptr_t, uintptr_t, SIZE_T, SSIZE_T, INT_PTR, DWORD_PTR, etc	32	64	Integer types capable to keep pointer value.
time_t	32	64	Amount of time in seconds.

Table N1. Description of some integer types.

We'll use term "**memsize**" type in the text. This term means any simple integer type which is capable of keeping a pointer, and changes its size according to the change of platform from 32-bit to 64-bit. For example, **memsize** types are: size_t, ptrdiff_t, **all pointers**, intptr_t, INT_PTR, DWORD_PTR.

We should also mention the data models which determine the correspondent sizes of fundamental types for different systems. Table N2 contains data models which may interest us.

	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

Table N2. 32-bit and 64-bit data models.

In this article we'll assume that the program will be ported from a system with the ILP32 data model to systems with LP64 (/en/t/0028/) or LLP64 (/en/t/0026/) data model.

And finally, the 64-bit model in Linux (LP64) differs from that in Windows (LLP64) only in the size of **long** type. Since it is their only difference, we'll avoid using long, unsigned long types, and will use ptrdiff_t, size_t types to generalize the article.

Let us observe the type errors which occur while porting programs on the 64-bit architecture.

1. Disabled warnings

All books on high-quality code development recommend you set the level of warnings shown by the compiler to the highest possible value. But there are situations in practice when the diagnosis level for some project parts is lower, or the diagnosis may even be disabled altogether. As a rule it is very old code which is supported but not modified. Programmers who work over the project are used to the fact that this code works, and don't take its quality into consideration. Thus, one can miss serious warnings produced by the compiler while porting programs on the new 64-bit system.

While porting an application you should always turn on warnings for the entire project. This will help you check the compatibility of the code, and analyze the code thoroughly. This approach can help to save you a lot of time while debugging the project on the new architecture.

If we don't do this, we will face the simplest and stupidest errors in all their variety. Here is a simple example of overflow which occurs in a 64-bit program if we ignore warnings completely.

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

2. Use of functions with a variable number of arguments

The typical example is the incorrect use of **printf**, **scanf** functions and their variants:

```
1) const char *invalidFormat = "%u";
   size_t value = SIZE_MAX;
   printf(invalidFormat, value);
2) char buf[9];
   sprintf(buf, "%p", pointer);
```

In the first case, it is not taken into account that size_t type is not equivalent to unsigned type on the 64-bit platform. As a result, it will cause printing of an incorrect result if value > UINT_MAX.

In the second case the developer didn't take into account that the pointer size may become more than 32-bit in future. As the result this code will cause buffer overflow on the 64-bit architecture.

Incorrect use of functions with a variable number of arguments, is a typical error on all architectures, not only on 64-bit. This is related to the fundamental danger of the use of the given C++ language constructions. The common practice is to refuse them and use safe programming methods.

We strongly recommend you modify the code and use safe methods. For example, you may replace **printf** with **cout**, and **sprintf** with **boost::format** or **std::stringstream**.

If you have to maintain a code which uses functions of **sscanf** type, in the control lines format we can use special macros which turn into necessary modifiers for different systems. For example:

```
// PR_SIZET on Win64 = "I"
// PR_SIZET on Win32 = ""
// PR_SIZET on Linux64 = "l"
// ...
size_t u;
scanf("%" PR_SIZET "u", &u);
```

3. Magic numbers

Low-quality code often contains magic numbers, the mere presence of which is dangerous. During the migration of the code onto the 64-bit platform, these magic numbers may make the code inefficient if they participate in calculation of address, object size or bit operations.

Table N3 contains basic magic numbers which may influence the workability of an application on a new platform.

Value	Description
4	Number of bytes in a pointer type
32	Number of bits in a pointer type
0x7fffffff	The maximum value of a 32-bit signed variable. Mask for zeroing of the high bit in a 32-bit type.
0x80000000	The minimum value of a 32-bit signed variable. Mask for allocation of the high bit in a 32-bit type.
0xffffffff	The maximum value of a 32-bit variable. An alternative record -1 as an error sign.

Table N3. Basic magic numbers which can be dangerous during the port of applications from 32-bit to 64-bit platform.

You should study the code thoroughly in search of magic numbers and replace them with safe numbers and expressions. To do so you can use **sizeof()** operator, special values from <limits.h>, <inttypes.h> etc.

Let's take a look at some errors related to the use of magic numbers. The most frequent is using numbers to store type sizes.

```
1) size_t ArraySize = N * 4;
   intptr_t *Array = (intptr_t *)malloc(ArraySize);
2) size_t values[ARRAY_SIZE];
   memset(values, 0, ARRAY_SIZE * 4);
3) size_t n, newexp;
   n = n >> (32 - newexp);
```

Let's assume that in all cases the size of the types used is always 4 bytes. To make the code correct, we should use the **sizeof**() operator.

```
1) size_t ArraySize = N * sizeof(intptr_t);
   intptr_t *Array = (intptr_t *)malloc(ArraySize);
2) size_t values[ARRAY_SIZE];
   memset(values, 0, ARRAY_SIZE * sizeof(size_t));
```

or

```
   memset(values, 0, sizeof(values)); //preferred alternative
3) size_t n, newexp;
   n = n >> (CHAR_BIT * sizeof(n) - newexp);
```

Sometimes we may need a specific number. As an example let's take the size_t where all the bits except 4 low bits must be filled with ones. In a 32-bit program this number may be declared in the following way.

```
// constant '1111..110000'
const size_t M = 0xFFFFFFF0u;
```

This code is incorrect for a 64-bit system. Such errors are very unpleasant because the recording of magic numbers may be carried out in different ways, and the search for them is very laborious. Unfortunately, there is no other way except to find and correct this code using #ifdef or a special macro.

```
#ifdef _WIN64
  #define CONST3264(a) (a##i64)
#else
  #define CONST3264(a)  (a)
#endif
const size_t M = ~CONST3264(0xFu);
```

Sometimes as an error code or other special marker "-1" value is used, which is written as "0xffffffff". On the 64-bit platform the recorded expression is incorrect, and we should use the "-1" value explicitly. Here is an example of an incorrect code using 0xffffffff value as an error sign.

```
#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();
```

To be on the safe side, let's make sure that you know clearly what the result of "(size_t)(-1)" value is on the 64-bit platform. You may make a mistake saying value 0x00000000FFFFFFFFu. According to C++ rules -1 value turns into a signed equivalent of a higher type and then into an unsigned value:

```
int a = -1;           // 0xFFFFFFFFi32
ptrdiff_t b = a;      // 0xFFFFFFFFFFFFFFFFi64
size_t c = size_t(b); // 0xFFFFFFFFFFFFFFFFui64
```

Thus "(size_t)(-1)" on the 64-bit architecture is represented by 0xFFFFFFFFFFFFFFFFui64 value which is the highest value for the 64-bit size_t type.

Let's return to the error with **INVALID_RESULT**. The use of the number 0xFFFFFFFFu causes execution failure of "len == (size_t)(-1)" condition in a 64-bit program. The best solution is to change the code in such a way that it doesn't need special marker values. If you need to use them for some reason or consider this suggestion unreasonable, to correct the code fundamentally just use fair value -1.

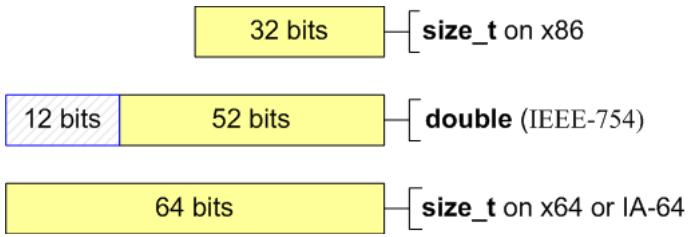
```
#define INVALID_RESULT (size_t(-1))
...
```

4. Storing integers in double type

Double type as a rule, has 64 bits size and is compatible with IEEE-754 standard on 32-bit and 64-bit systems. Some programmers use **double** type for storing of, and working with, integer types.

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

The given example can be justified on a 32-bit system, as double type has 52 significant bits and is capable of storing a 32-bit integer value without loss. But while trying to store a 64-bit integer in double the exact value can be lost (see picture 1).



Picture 1. The number of significant bits in size_t and double types.

It is possible that an approximate value can be used in your program, but to be on the safe side we'd like to warn you about possible effects on the new architecture. In any case it is not recommended to mix integer arithmetic with floating-point arithmetic.

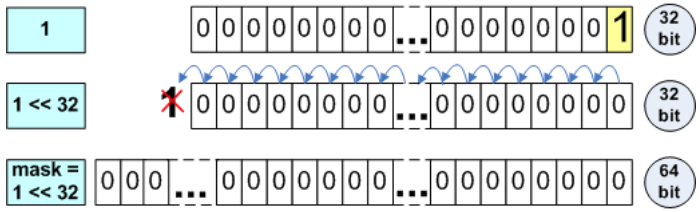
5. Bit shifting operations

Bit shifting operations can cause a lot of problems during the port from the 32-bit system to the 64-bit one if proper attention is not paid. Let's begin with an example of a function which defines the bit you've chosen as 1 in a variable of **memsize** type.

```
ptrdiff_t SetBitN(ptrdiff_t value, unsigned bitNum) {
  ptrdiff_t mask = 1 << bitNum;
  return value | mask;
}
```

The given code works only on the 32-bit architecture, and allows the definition of bits with numbers from 0 to 31. After the program is ported to a 64-bit platform, it becomes necessary to define bits from 0 to 63. What value will the SetBitN(0, 32) call return? If you think that the value is 0x100000000, the author is glad because he hasn't prepared this article in vain. You'll get 0.

Pay attention to the fact that "1" has int type and during the shift on 32 positions, an overflow will occur as it is shown in picture 2.



(/media/images/content/b/20_issues_of_porting_C++_code_on_the_64-bit_platform/image3.png)

Picture 2. Mask value calculation.

To correct the code, it is necessary to make the constant "1" of the same type as the variable **mask**.

```
ptrdiff_t mask = ptrdiff_t(1) << bitNum;
```

or

```
ptrdiff_t mask = CONST3264(1) << bitNum;
```

One more question. What will be the result of the uncorrected function SetBitN(0, 31) call? The right answer is 0xffffffff80000000. The result of 1 << 31 expression is negative number -2147483648. This number is formed in a 64-bit integer variable as 0xffffffff80000000. You should keep in mind, and take into consideration, the effects of the shift of values of different types. To make you understand the stated information better table N4 contains interesting expressions with shifts on the 64-bit system.

Expression	Result (Dec)	Result (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

Table N4. Expressions with shifts and results on a 64-bit system.

6. Storing of pointer addresses

Many errors during the migration on 64-bit systems are related to the change of a pointer size in relation to the size of usual integers. Usual integers and pointers have the same size in an environment with the ILP32 data model. Unfortunately, the 32-bit code is based on this supposition everywhere. Pointers are often cast to int, unsigned int and other types improper to fulfill address calculations.

One should use only **memsize** types for the integer form of pointers. The uintptr_t type is more preferable since it shows programmer's intentions more clearly, and makes the code more portable, saving it from future changes

Let's take a look at two small examples.

```
1) char *p;
   p = (char *) ((int)p & PAGEOFFSET);
2) DWORD tmp = (DWORD)malloc(ArraySize);
   ...
   int *ptr = (int *)tmp;
```

Both examples do not take into account that the pointer size may differ from 32-bits. They use explicit type conversion which truncates high bits in the pointer, and this is an error on the 64-bit system. Here are the corrected versions which use integer **memsize** types intptr_t and DWORD_PTR to store pointer addresses:

```
1) char *p;
   p = (char *) ((intptr_t)p & PAGEOFFSET);
2) DWORD_PTR tmp = (DWORD_PTR)malloc(ArraySize);
   ...
   int *ptr = (int *)tmp;
```

The two examples studied are dangerous because the program failure may be found much later. The program may work perfectly with a small data on a 64-bit system, while the truncated addresses are located in the first 4 Gb of memory. Then, on launching the program for large production aims, there will be the memory allocation out of first 4 Gb. The code given in the examples will cause an undefined behavior of the program on the object out of first 4 Gb while processing the pointer.

The following code won't hide, and will show up, at the first execution.

```
void GetBufferAddr(void **retPtr) {
  ...
  // Access violation on 64-bit system
  *retPtr = p;
}
unsigned bufAddress;
GetBufferAddr((void **)&bufAddress);
```

The correction is also in the choice of the type capable to store the pointer.

```
uintptr_t bufAddress;
GetBufferAddr((void **)&bufAddress); //OK
```

There are situations when storing of a pointer address into a 32-bit type is just necessary. For the most part, such situations appear when it is necessary to work with old API functions. For such cases, one should resort to special functions LongToIntPtr, PtrToUlong etc.

I'd like to stress, that it would be bad practice to store a pointer address into types which are always equal to 64-bits. One will have to correct the code shown again when 128-bit systems will appear.

```
PVOID p;
// Bad style. The 128-bit time will come.
__int64 n = __int64(p);
p = PVOID(n);
```

7. Memsize types in unions

The peculiarity of a union, is that for all members of the union, the same memory area is allocated; that is to say, they overlap. Although the access to this memory area is possible with the use of any of the elements, the element for this aim should be chosen so that the result won't be meaningless.

One should pay attention to the unions which contain pointers and other members of **memsize** type.

When there is a necessity to work with a pointer as an integer, sometimes it is convenient to use the union as it is shown in the example, and work with the numeric form of the type without using explicit conversions.

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

This code is correct on 32-bit systems and is incorrect on 64-bit ones. When changing the **m_n** member on a 64-bit system, we work only with a part of the **m_p**. We should use the type which will correspond to the pointer size.

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

Another frequent use of the union is the presentation of one member as a set of other smaller ones. For example, we may need to split a value of size_t type into bytes to carry out the table algorithm of calculation of the number of zero bits in a byte.

```
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];
```

Here is a fundamental algorithmic error which consists in the supposition that size_t type consists of 4 bytes. The possibility of the automatic search of algorithmic errors is hardly possible, but we can provide the search of all the unions, and check the presence of **memsize** types in them. Having found such a union we can find an algorithmic error and rewrite the code in the following way.

```
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. Changing an array type

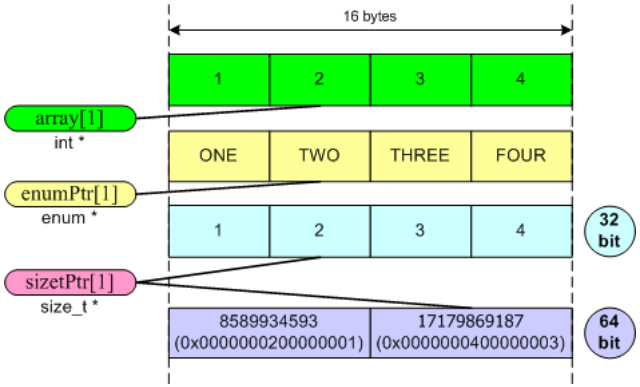
Sometimes it is necessary (or just convenient) to present array items as elements of a different type. Dangerous and safe type conversions are shown in the following code.

```
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
```

As you can see the program output is different in 32-bit and 64-bit systems. On the 32-bit system, the access to the array items is fulfilled correctly because sizes of size_t and int coincide and we see "2 2".

On a 64-bit system we got "2 17179869187" in the output because the 17179869187 value is located in the first item of **sizetPtr** array (see picture 3). In some cases we need this very behavior but usually it is an error.



(/media/images/content/b/20_issues_of_porting_C++_code_on_the_64-bit_platform/image5.png)

Picture 3. Arrangement of array items in memory.

The fix for the described situation, is rejecting dangerous type conversions by modernizing the program. Another variant is to create a new array and copy values of the original one into it.

9. Virtual functions with arguments of memsize type

If there are large derived class graphs with virtual functions in your program, there is a risk in using arguments of different types inattentively. However, these types actually coincide on the 32-bit system. For example, in the base class you use size_t type as an argument of a virtual function and in the derived class you use the unsigned type. So, this code will be incorrect on a 64-bit system.

But an error like this doesn't necessarily hide in large derived class graphs and here is one of the examples.

```
class CWinApp {
  ...
  virtual void WinHelp(DWORD_PTR dwData, UINT nCmd);
};
class CSampleApp : public CWinApp {
  ...
  virtual void WinHelp(DWORD dwData, UINT nCmd);
};
```

Let's follow the life-cycle of the development of some applications. Imagine that first it was being developed for Microsoft Visual C++ 6.0 when **WinHelp** function in **CWinApp** class had the following prototype:

```
virtual void WinHelp(DWORD dwData, UINT nCmd = HELP_CONTEXT);
```

It was absolutely correct to carry out an overlap of the virtual function in **CSampleApp** class as it is shown in the example. Then the project was ported into Microsoft Visual C++ 2005 where the function prototype in **CWinApp** class had undergone some changes which consisted of the replacement of DWORD type with DWORD_PTR type. On the 32-bit system the program will work perfectly, since DWORD and DWORD_PTR types coincide. Troubles will appear during the compilation of the given code for a 64-bit platform. We'll get two functions with the same name but different parameters and as a result the user's code won't be executed.

The correction is in the use of the same types in the corresponding virtual functions.

```
class CSampleApp : public CWinApp {
  ...
  virtual void WinHelp(DWORD_PTR dwData, UINT nCmd);
};
```

10. Serialization and data exchange

An important point during the port of a software solution on a new platform, is succession to the existing data exchange protocol. It is necessary to read existing projects formats, in order to carry out the data exchange between 32-bit and 64-bit processes, etc.

For the most part, errors of this kind are in the serialization of **memsize** types and data exchange operations using them.

```
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));
```

In all the given examples there are errors of two kinds: the use of types of volatile size in binary interfaces and ignoring the byte order.

The use of types of volatile size

It is unacceptable to use types which change their size depending on the development environment in binary interfaces of data exchange. In C++ language the types don't all have distinct sizes, and consequently it is not possible to use them all for these purposes. That's why the developers of the development environments and programmers themselves create data types which have an exact size such as __int8, __int16, INT32, word64 etc.

The usage of such types provides data portability between programs on different platforms, although it needs the usage of odd ones. The three shown examples are written inaccurately, and this will show up on the changing of the capacity of some data types from 32-bit to 64-bit. Taking into account the necessity to support old data formats, the correction may look as follows:

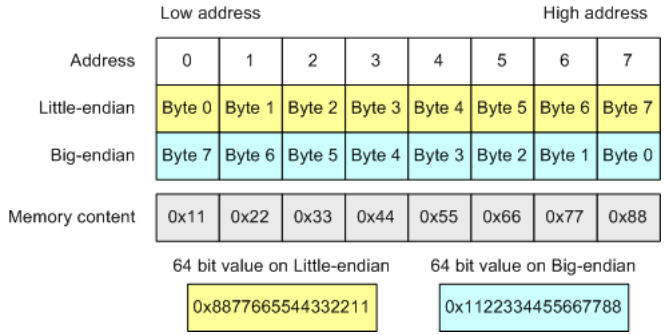
```
1) size_t PixelCount;
   __uint32 tmp;
   fread(&tmp, sizeof(tmp), 1, inFile);
   PixelCount = static_cast<size_t>(tmp);
2) __int32 value_1;
   __int32 value_2;
   inputStream >> value_1 >> value_2;
3) time_t time;
   __uint32 tmp = static_cast<__uint32>(time);
   PackToBuffer(MemoryBuf, &tmp, sizeof(tmp));
```

But the given version of correction may not be the best. During the port on the 64-bit system the program may process a large amount of data, and the use of 32-bit types in the data may become a serious problem. In this case we may leave the old code for compatibility with the old data format having corrected the incorrect types, and fulfill the new binary data format taking into account the errors made. One more variant is to refuse binary formats and take text format or other formats provided by various libraries.

Ignoring of the byte order

Even after the correction of volatile type sizes, you may face the incompatibility of binary formats. The reason is different data presentation. Most frequently it is related to a different byte order.

The byte order is a method of recording bytes of multibyte numbers (see also picture 4). The little-endian order means that the recording starts with the lowest byte and ends with the highest one. This recording order was acceptable for memory of PCs with x86-processors. The big-endian order - the recording starts with the highest byte and ends with the lowest one. This order is a standard for TCP/IP protocols. That's why the big-endian byte order is often called the network byte order. This byte order is used by the Motorola 68000, SPARC processors.



Picture 4. Byte order in a 64-bit type on little-endian and big-endian systems.

While developing the binary interface or data format you should keep the byte order in mind. If the 64-bit system on which you are porting a 32-bit application has a different byte order you'll just have to take it into account in your code. For conversion between the big-endian and the little-endian byte orders you may use functions htonl(), htons(), bswap_64, etc.

11. Bit fields

If you use bit fields, you should keep in mind that the use of **memsize** types will cause a change in structure size, and alignment. For example, the structure shown next will have 4 bytes size on the 32-bit system and 8 bytes size on a 64-bit one.

```
struct MyStruct {
  size_t r : 5;
};
```

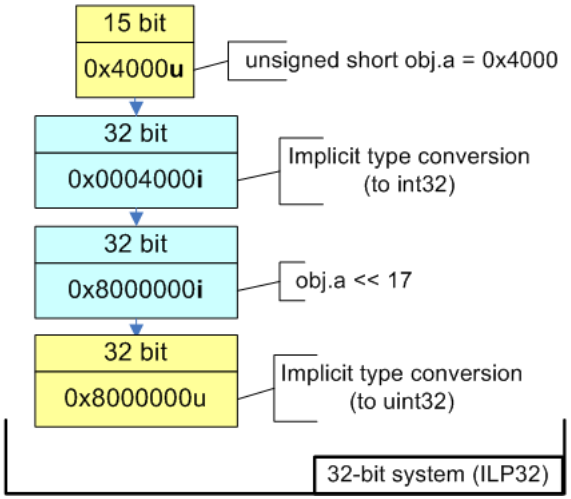
But our attention to bit fields is not limited by that. Let's take a delicate example.

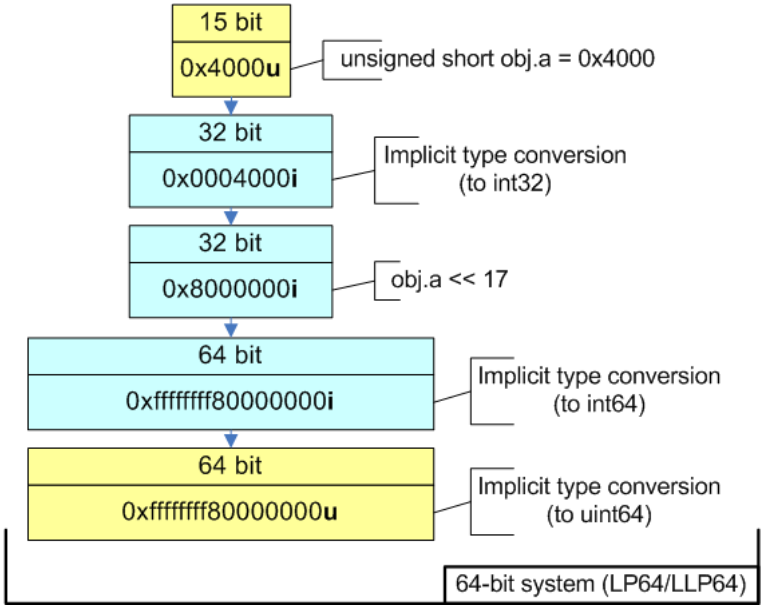
```
struct BitFieldStruct {
  unsigned short a:15;
  unsigned short b:13;
};
BitFieldStruct obj;
obj.a = 0x4000;
size_t addr = obj.a << 17; //Sign Extension
printf("addr 0x%Ix\n", addr);

//Output on 32-bit system: 0x80000000
//Output on 64-bit system: 0xffffffff80000000
```

Pay attention to the fact that if you compile the example for a 64-bit system, there is a sign extension in "addr = obj.a << 17;" expression, in spite of the fact that both variables, **addr** and **obj.a**, are unsigned. This sign extension is caused by the rules of type conversion which are used in the following way (see also picture 5):

- A member of **obj.a** is converted from a bit field of unsigned short type into int. We get int type and not unsigned int because the 15-bit field can be located in the 32-bit signed integer.
- "obj.a << 17" expression has int type but it is converted into ptrdiff_t and then into size_t before it will be assigned to variable **addr**. The sign extension occurs during the conversion from int into ptrdiff_t.





Picture 5. Expression calculation on different systems.

Therefore you should be attentive while working with bit fields. To avoid the described effect in our example we can simply use explicit conversion from **obj.a** type to size_t type.

```
...
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. Pointer address arithmetic

The first example:

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

This example works correctly with pointers if the value of "a16 * b16 * c16" expression does not exceed UINT_MAX (4Gb). Such code may always work correctly on the 32-bit platform, as the program has never allocated arrays of large sizes. On the 64-bit architecture the size

of the array exceeded UINT_MAX items. Suppose we would like to shift the pointer value on 6.000.000.000 bytes, and that's why variables a16, b16 and c16 have values 3000, 2000 and 1000 correspondingly. While calculating "a16 * b16 * c16" expression all the variables according to C++ rules will be converted to int type, and only then their multiplication will occur. During the process of multiplication an overflow will occur. The incorrect expression result will be extended to ptrdiff_t type, and the calculation of the pointer will be incorrect.

One should take care to avoid possible overflows in pointer arithmetic. For this purpose it's better to use **memsize** types, or explicit type conversion in expressions which carry pointers. We can rewrite the code in the following way using explicit type conversion:

```
short a16, b16, c16;
char *pointer;
...
pointer += static_cast<ptrdiff_t>(a16) *
           static_cast<ptrdiff_t>(b16) *
           static_cast<ptrdiff_t>(c16);
```

If you think that only inaccurate programs working on larger data sizes face problems, we have to disappoint you. Let's take a look at an interesting piece of code for working with an array containing only 5 items. The second example works in the 32-bit version, but not in the 64-bit version.

```
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
```

Let's follow the calculation flow of the "ptr + (a + b)" expression:

- According to C++ rules variable A of int type is converted to unsigned type.
- Addition of A and B occurs. The result we get is value 0xFFFFFFFF of unsigned type.

Then calculation of "ptr + 0xFFFFFFFFu" takes place, but the result of it depends on the pointer size on the particular architecture. If the addition takes place in a 32-bit program, the given expression will be an equivalent of "ptr - 1" and we'll successfully print number 3.

In a 64-bit program 0xFFFFFFFFu value will be added fairly to the pointer and the result will be that the pointer will be outbound of the array. And we'll face problems while getting access to the item of this pointer.

To avoid the situation shown, as well as in the first case, we advise you to use only **memsize** types in pointer arithmetic. Here are two variants of the code correction:

```
ptr = ptr + (ptrdiff_t(A) + ptrdiff_t(B));
ptrdiff_t A = -2;
size_t B = 1;
...
ptr = ptr + (A + B);
```

You may object and offer the following variant of the correction:

```
int A = -2;
int B = 1;
...
ptr = ptr + (A + B);
```

Yes, this code will work but it is bad for a number of reasons:

1. It will teach you inaccurate work with pointers. After a while you may forget nuances, and make a mistake by making one of the variables of unsigned type.
2. Using of non-**memsize** types along with pointers is potentially dangerous. Suppose variable **Delta** of **int** type participates in an expression with a pointer. This expression is absolutely correct. But the error may hide in the calculation of the variable **Delta** itself, for 32-bit may be not enough to make the necessary calculations while working with large data arrays. The use of **memsize** type for variable **Delta** liquidates the problem automatically.

13. Array indexing

This kind of error is separated from the others for better structuring of the account because indexing in arrays with the usage of square brackets, is just a different record of address arithmetic than that observed before.

Programming in C and then C++ has formed a practice of using variables of int/unsigned types in the constructions of the following kind:

```
unsigned Index = 0;
while (MyBigNumberField[Index] != id)
  Index++;
```

But time passes and everything changes. And now it's high time to say - do not do this anymore! Use **memsize** types for indexing (large) arrays.

The given code won't process an array containing more than UINT_MAX items in a 64-bit program. After the access to the item with UNIT_MAX index, an overflow of the **Index** variable will occur, and we'll get infinite loop.

To fully persuade you of the necessity of using only **memsize** types for indexing, and in the expressions of address arithmetic, I'll give you one last example.

```
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];
}
```

The given code is taken from a real program of mathematics simulation, in which the size of RAM is an important resource, and the possibility to use more than 4 Gb of memory on the 64-bit architecture improves the calculation speed greatly. In programs of this class, one-dimensional arrays are often used to save memory while they participate as three-dimensional arrays. For this purpose there are functions like **GetCell** which provide access to the necessary items. The given code, however, will work correctly only with arrays containing less than INT_MAX items. The reason for this, is the use of 32-bit **int** types for calculation of the items index.

Programmers often make a mistake trying to correct the code in the following way:

```
float Region::GetCell(int x, int y, int z) const {
  return array[static_cast<ptrdiff_t>(x) + y * Width +
               z * Width * Height];
}
```

They know that according to C++ rules, the expression for calculation of the index will have ptrdiff_t type, and hope to avoid the overflow with its help. But the overflow may occur inside the sub-expression "y * Width" or "z * Width * Height" since the int type is still used to calculate them.

If you want to correct the code without changing types of the variables participating in the expression, you may use explicit type conversion of every variable to **memsize** type:

```
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)];
}
```

Another solution is to replace types of variables with **memsize** type:

```
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. Mixed use of simple integer types and memsize types

Mixed use of **memsize** and non-**memsize** types in expressions may cause incorrect results on 64-bit systems, and may be related to the change of the input values rate. Let's study some examples.

```
size_t Count = BigValue;
for (unsigned Index = 0; Index != Count; ++Index)
{ ... }
```

This is an example of an eternal loop if **Count** > UINT_MAX. Suppose this code worked on 32-bit systems with the range less than UINT_MAX iterations. But a 64-bit variant of the program may process more data, and it may need more iterations. As far as the values of the **Index** variable lie in the [0..UINT_MAX] range the "Index != Count" condition will never be executed and this will cause the infinite loop.

Another frequent error is recording expressions in the following form:

```
int x, y, z;
intptr_t SizeValue = x * y * z;
```

Similar examples were discussed earlier, when during the calculation of values with the use of non-**memsize** types an arithmetic overflow occurred. And the last result was incorrect. Identification and correction of the given code is made more difficult because compilers do

not show any warning messages on it as a rule. This construction is absolutely correct for the C++ language. Several variables of int type are multiplied, and after that the result is implicitly converted to intptr_t type and assignment occurs.

Let's provide an example of a small code fragment which shows the danger of inaccurate expressions with mixed types (the results are retrieved in Microsoft Visual C++ 2005, 64-bit compilation mode).

```
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
```

It is necessary that all the operands in such expressions have been converted to the type of larger capacity in time. Remember that the expression of the following kind:

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

does not promise the right result. It promises only that the *intptr_t(x) * y * z* expression will have intptr_t type.

This is why, if the result of the expression should be of **memsize** type, only **memsize** types must participate in the expression. The right variant:

```
intptr_t v2 = intptr_t(x) + intptr_t(y) * intptr_t(z); // OK!
```

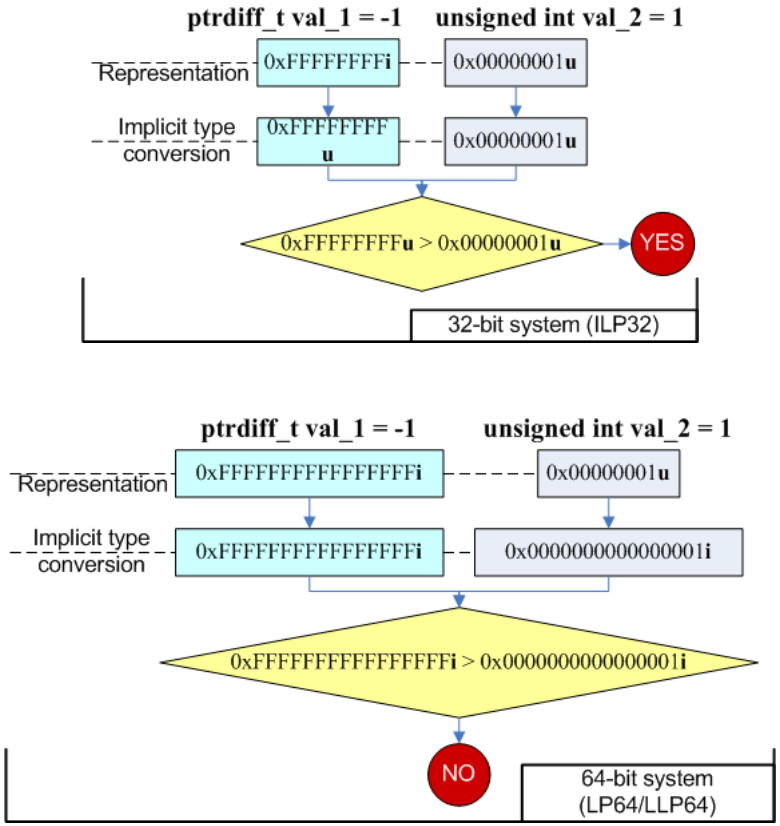
Notice; if you have a lot of integer calculations and control over the overflows is an important task for you, we suggest you to pay attention to the SafeInt class, the description of which can be found in MSDN Library.

Mixed use of types may cause changes in program logic.

```
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"
```

On the 32-bit system the variable **val_1** according to C++ rules was extended to **unsigned int**, and became value 0xFFFFFFFFu. As a result the condition "0xFFFFFFFFu > 1" was executed. On the 64--bit system, it's the other way around - the variable **val_2** is extended to ptrdiff_t type**.** In this case the expression "-1 > 1" is checked. On picture 6 the occurring changes are shown sketchy.



Picture 6. Changes occurring in the expression.

If you need to return the previous behavior, you should change the **val_2** variable type.

```
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. Implicit type conversions while using functions

Observing the previous types of errors, related to mixing of simple integer types and **memsize** types, we have examined only simple expressions. But similar problems may occur while using other C++ constructions too.

```
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;
```

If you work with large arrays (more than INT_MAX items) the given code may behave incorrectly, and we'll address not the items of the MyArray array we wanted. Despite the fact that we return the value of the size_t type, the "x + y * Width + z * Width * Height" expression is calculated with using the int type. We suppose you have already guessed that the corrected code will look as follows:

```
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);
}
```

In the next example we also have **memsize** type (pointer) and simple unsigned type mixed.

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

The result of the "end - begin" expression has ptrdiff_t type. As long as the function returns **unsigned** type, implicit type conversion during which high bits of the results be lost, will occur. Thus, if the **begin** and **end** pointers address the beginning and the end of an array

whose size is larger than UINT_MAX (4Gb), the function will return an incorrect value.

Here is one more example, but now we'll observe not the returned value but the formal function argument.

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

Doesn't this code remind you of the example of the incorrect pointer arithmetic discussed earlier? Yes, we find the same situation here. The incorrect result appears during the implicit type conversion of the actual argument which has the 0xFFFFFFFF value from the unsigned type to the ptrdiff_t type.

16. Overloaded functions

During the port of 32-bit programs to a 64-bit platform, the change of the logic of its work may be found which is related to the use of overloaded functions. If the function is overlapped for 32-bit and 64-bit values, the access to it with the argument of **memsize** type will be compiled into different calls on different systems. This approach may be useful, as for example, in the following code:

```
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);
```

But such a change of logic contains a potential danger. Imagine a program in which a class is used for organizing stack. The peculiarity of this class is that it allows storage of values of different types.

```
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);
```

A careless programmer placed values of different types (ptrdiff_t and int), and then took them from the stack. On the 32-bit system their sizes coincided and everything worked perfectly. When the size of ptrdiff_t type changes in a 64-bit program, the stack object begins to take more bytes than it retrieves later.

We think you understand this kind of error and that you should pay attention to the call of overloaded functions transferring actual arguments of **memsize** type.

17. Data alignment

Processors work more efficiently when they deal with data aligned properly. As a rule the 32-bit data item must be aligned at the border multiple of 4 bytes, and the 64-bit item at the border multiple of 8 bytes. An attempt to work with unaligned data on IA-64 (Itanium) processors will cause an exception as shown in the following example,.

```
#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
}
```

If you have to work with unaligned data on Itanium you should tell this to the compiler. For example, you may use a special macro 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
}
```

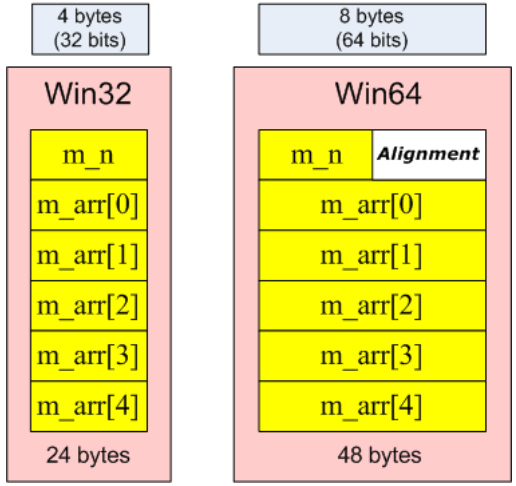
This solution is not efficient, because the access to the unaligned data will be several times slower. A better result may be achieved if you arrange up to 32-bit, 16-bit and 8-bit items in 64-bit data items.

On the x64 architecture during the access to unaligned data, an exception does not occur, but you should avoid them also. Firstly, because of the essential slowdown of the access to this data, and secondly, because of a high probability of porting the program on the IA-64 platform in the future.

Let's take a look at one more example of a code which does not take into account the data alignment.

```
struct MyPointersArray {
  DWORD m_n;
  PVOID m_arr[1];
} object;
...
malloc( sizeof(DWORD) + 5 * sizeof(PVOID) );
...
```

If we want to allocate the memory size necessary for storing an object of the **MyPointersArray** type containing 5 pointers, we should take into account that the beginning of the array **m_arr** will be aligned at the border of 8 bytes. The order of data in memory on different systems (Win32 (/en/t/0054/) / Win64 (/en/t/0055/)) is shown in picture 7.



Picture 7. Alignment of data in memory on Win32 and Win64 systems

The correct calculation of the size should look as follows:

```
struct MyPointersArray {
  DWORD m_n;
  PVOID m_arr[1];
} object;
...
malloc( FIELD_OFFSET(struct MyPointersArray, m_arr) +
        5 * sizeof(PVOID) );
...
```

In this code we get the shift of the last structure member, and add this shift to the member's size. The shift of a member of the structure, or a class, may be recognized when the **offsetof** or **FIELD_OFFSET** macro is used.

Always use these macros to get a shift in the structure without relying on your knowledge of the sizes of types and the alignment. Here is an example of a code with the correct calculation of the structure member address:

```
struct TFoo {
  DWORD_PTR whatever;
  int value;
} object;
int *valuePtr =
  (int *)((size_t)(&object) + offsetof(TFoo, value)); // OK
```

18. Exceptions

Throwing and handling exceptions using integer types is not a good programming practice for the C++ language. You should use more informative types for such purposes, for example, classes derived from the std::exception class. But sometimes one has to work with lower quality code as is shown below.

```
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;
}
```

You should completely avoid throwing or handling exceptions using **memsize** types, since it may cause the change in the program logic. The correction of the given code may consist in the replacement of "catch (int)" with "catch (ptrdiff_t)". A more proper correction is the use of a special class for transferring the information about the error which has occurred.

19. Using outdated functions and predefined constants

While developing a 64-bit application, keep the changes of the environment in which it will be performed in mind. Some functions will become outdated, and it will be necessary to replace them with new versions. GetWindowLong is a good example of such function in the Windows operating system. Pay attention to the constants concerning interaction with the environment in which the program is functioning. In Windows the lines containing "system32" or "Program Files" will be suspicious.

20. Explicit type conversions

Be accurate with explicit type conversions. They may change the logic of the program execution when types change their capacity, or cause loss of significant bits. It is difficult to cite examples of typical errors related to the explicit type conversion, as they are very different and specific for different programs. You have already gotten acquainted with some errors related to the explicit type conversion earlier.

Error diagnosis

The diagnosis of errors occurring while porting 32-bit programs to 64-bit systems is a difficult task. The porting of lower quality code, written without taking into account peculiarities of other architectures, may demand a lot of time and effort. This is why we'll pay particular attention to the description of approaches, and means by which we may simplify this task.

Unit testing

Unit testing earned respect among programmers long ago. Unit tests will help you to check the correctness of the program after the port to a new platform. But there is one nuance which you should keep in mind.

Unit testing may not allow you to check the new ranges of input values which become accessible on 64-bit systems. Unit tests were originally developed in such a way that they can be performed in a short period of time; and the function which usually works with an array with the size of tens of Mb, will probably process tens of Kb in unit tests. It is justified because this function may be called many times with different sets of input values in tests; but suppose you have a 64-bit version of the program, and now the function we study is processing more than 4 Gb of data. Of course, there appears to be a necessity to raise the input size of an array in the tests up to size more than 4 Gb. The problem is that the time spent performing the tests will be greatly increased.

This is why, while modifying the sets of tests, you must keep in mind the compromise between the time spent performing unit tests, and the thoroughness of the checks. Fortunately, there are other approaches which can help you to ensure that your application works correctly.

Code review

Code review is the best method of searching for errors and improving code. Combined and thorough code review may help you to completely rid your code of all errors related to the peculiarities of the development of 64-bit applications. Of course, in the beginning one should learn which errors to search for, otherwise the review won't give good results. For this purpose it is necessary to read this and other articles concerning the porting of programs from 32-bit systems to 64-bit. Some interesting links concerning this topic can be found at the end of the article.

But this approach to the analysis of the original code has an significant disadvantage. It demands a lot of time, and because of this, it is inapplicable on large projects.

The compromise is the use of static analyzers. A static analyzer can be considered to be an automated system for code review, whereby a list of potentially dangerous places is created for a programmer so that he may carry out further analysis.

In any case it is desirable to provide several code reviews in order to teach the team to search for new kinds of errors occurring on 64-bit systems.

Built-in means of compilers

Compilers allow us to solve some of the problems in searching for defective code. They often have built-in mechanisms for diagnosing errors observed. For example, in Microsoft Visual C++ 2005 the following keys may be useful: /Wp64 (/en/t/0057/), /Wall, and in SunStudio C++ key -xport64.

Unfortunately, the possibilities they provide are often not enough, and you should not rely solely on them. In any case, it is highly recommended that you enable the corresponding options of a compiler for diagnosing errors in the 64-bit code.

Static analyzers

Static analyzers are a fine means to improve the quality and safety of program code. The basic difficulty related to the use of static analyzers is in the fact that they generate quite a lot of false warning messages concerning potential errors. Programmers being lazy by nature, use this argument to find some way not to correct the found errors. Microsoft solves this problem by including the found errors in the bug tracking system unconditionally. Thus, a programmer cannot choose between the correction of the code, and an attempt to avoid this.

We think that such strict rules are justified. The profit in the quality code covers the outlay of time for static analysis and corresponding code modification. This profit is achieved by means of simplifying the code support, and reducing the time spent debugging and testing.

Static analyzers may be successfully used for diagnosing many of the errors observed in the article.

The authors know 3 static analyzers which are supposed to have the means to diagnose errors related to the porting of programs to 64-bit systems. We would like to warn you at once that we may be mistaken about the possibilities they have, moreover these are developing products, and new versions may have greater efficiency.

1. **Gimpel Software PC-Lint** (http://www.gimpel.com (http://www.gimpel.com/)). This analyzer has a large list of supported platforms and a general purpose static analyzer. It allows you to catch errors while porting programs on architectures with LP64 data model. The advantage is the possibility to take strict control over the type conversions. The absence of the environment may be thought to be a disadvantage, but it may be corrected by using an additional product, Riverblade Visual Lint.
2. **Parasoft C++test** (http://www.parasoft.com (http://www.parasoft.com/)). Another well-known general purpose static analyzer. This analyzer has support for a lot of devices and program platforms. It has a built-in environment, which greatly simplifies the work process and setting of the analysis rules.

As well as PC-Lint it is intended for LP64 data model.

1. **Viva64** (http://www.viva64.com (/)). Unlike other analyzers, this one is intended to work with Windows (LLP64) data model. It is integrated into the development environment Visual Studio 2005. The analyzer is intended for use only in diagnosing problems related to the porting of programs to 64-bit systems, and that simplifies its setting greatly.

Conclusion

If you read these lines we are glad that you're interested. We hope the article has been useful to you and will help you simplify the development and debugging of 64-bit applications. We will be glad to receive your opinions, remarks, corrections, additions and will surely include them in the next version of the article. The more we can describe typical errors, the more profitable our experience and help will be to you.

References

1. Converting 32-bit Applications Into 64-bit Applications: Things to Consider. http://www.oracle.com/technetwork/server-storage/solaris/ilp32tolp64issues-137107.html (http://www.oracle.com/technetwork/server-storage/solaris/ilp32tolp64issues-137107.html).
2. Andrew Josey. Data Size Neutrality and 64-bit Support. http://www.unix.org/whitepapers/64bit.html (http://www.unix.org/whitepapers/64bit.html).
3. Harsha S. Adiga. Porting Linux applications to 64-bit systems. http://www.ibm.com/developerworks/library/l-port64/index.html (http://www.ibm.com/developerworks/library/l-port64/index.html).
4. Porting an Application to 64-bit Linux on HP Integrity Servers. http://h21007.www2.hp.com/portal/StaticDownload?attachment_ciid=490964c3c39f111064c3c39f1110275d6e10RCRD

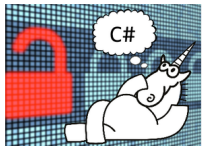
(http://h21007.www2.hp.com/portal/StaticDownload?attachment_ciid=490964c3c39f111064c3c39f1110275d6e10RCRD).

Previous article (/en/a/0003/) Next article (/en/a/0005/)

Evgeniy Ryzhkov (/en/b/a/evgeniy-ryzhkov)
Articles: 109

Andrey Karpov (/en/b/a/andrey-karpov)
Articles: 327

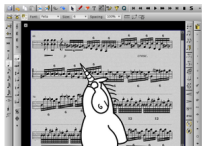
Liked this article? Read these! (/en/b/?utm_source=viva&utm_medium=recent_posts_bottom&utm_campaign=blog)



(/en/b/0537/)

What Is Wrong with Vulnerabilities in C# Projects?

This small article is an intermediate result of a search on a topic of already known vulnerabilities in open source ...



(/en/b/0536/)

Review of Music Software Code Defects. Part 3. Rosegarden

Programs for working with music have small amount of code and, initially, I doubted

(/en/b/0535/)

Appreciate Static Code Analysis!

I am really astonished by the capabilities of static code analysis even though I am one of the developers of ...

(/en/b/0535/)

(/en/b/0537/)

about the ability to find enough ...

(/en/b/0536/)

All articles → (/en/b/?utm_source=viva&utm_medium=all_articles_bottom&utm_campaign=blog)

PVS-Studio

We develop a PVS-Studio static code analyzer that finds errors in the C, C++, and C# programs on Windows and Linux.

goto PVS-Studio; (/en/pvs-studio/)

PVS-Studio

Buy

Achievements

Interesting

Company

Contact Us (/en/about/feedback/)

Site search

Rus (/ru/a/0004/)

(https://twitter.com/Code_Analysis)

(http://feeds.feedburner.com/viva64-blog-en)