Typical errors of porting C++ code on the 64-bit platform

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

This text is an abridged variant of "20 issues of porting C++ code on the 64-bit platform" article. If you are interested in the topic of porting programs to 64-bit systems, you can find a full version of the article here 20 issues of porting C++ code on the 64-bit platform.

We offer you to read the article devoted to the port of the program code of 32-bit applications on 64-bit systems. The article is written for programmers who use C++ but it may be also useful for all who face the port of applications on other platforms.

One should understand properly that the new class of errors which appears while writing 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 be ready for these difficulties and will show the ways to overcome them. Besides advantages, any new technology (in programming and other spheres as well) carries some limitations and even problems of using this technology. The same situation can be found in the sphere of developing 64-bit software. We all know that 64-bit software is the next step of the information technologies development. But in reality, only 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 opens before programmers. 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 those problems which the developer of 64-bit programs can face. In the article you will learn about:

The aim of this article is to observe thoroughly those problems which the developer of 64-bit programs can face. In the article you will learn about:

- typical errors of programming which occur on 64-bit systems;
- the reasons for appearing of these errors and the corresponding examples;
- methods of correcting the listed errors;
- the review of methods and means of searching errors in 64-bit programs.

The given information will allow you:

- to learn the differences between 32-bit and 64-bit systems;
- to avoid errors while writing the code for 64-bit systems;
- to speed up the process of migration of a 32-bit application on the 64-bit architecture with the help of the great reducing of the time of debugging and testing;
- to forecast the time of porting the code on the 64-bit system more accurately and seriously.

There are a lot of examples given in the article which you should try in the programming environment for better understanding. Going into them you will get something more than just an addition of separate elements. You will open the door into the world of 64-bit systems.

We'll use term "memsize" type in the text. By this we'll understand any simple integer type which is capable to keep a pointer and changes its size according to the change of the platform dimension form 32-bit to 64-bit. The examples memsize types: size_t, ptrdiff_t, all pointers, intptr_t, INT_PTR, DWORD_PTR.

We should say some words about the data models which determine the correspondence of the sizes of fundamental types for different systems. Table N1 contains data models which can interest us.

Code:

```
                      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, ptrdiff_t      32     64     64     64
pointer                32     64     64     64
```

On default in this article we'll consider that the program port will be fulfilled from the system with the data model ILP32 on systems with data model LP64 or LLP64.

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

Let's start observing the type errors which occur while porting programs on the 64-bit architecture.

1. Off-warnings.

In all books devoted to the development of the quality code it is recommended to set a warning level of warnings shown by the compiler on as high level as possible. But there are situations in practice when for some project parts there is a lower diagnosis level set or it is even set off. Usually it is very old code which is supported but not modified. Programmers who work over the project are used to that this code works and don't take its quality into consideration. Here it is a danger to miss serious warnings by the compiler while porting programs on the new 64-bit system.

While porting an application you should obligatory set on warnings for the whole project which help to check the compatibility of the code and analyze them thoroughly. It can help to save a lot of time while debugging the project on the new architecture.

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

Code:

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

2. Use of the functions with a variable number of arguments.

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

Code:

```
1) const char *invalidFormat="%u";
   size_t value=SIZE_MAX;
   printf(invalidFormat, value);
```

Code:

```
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. It will cause the printing of an incorrect result in case if value > UINT_MAX.

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

The incorrect use of functions with a variable number of arguments is a typical error on all the architectures, not only on 64-bit ones. 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 recommend you strongly to 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 support the code which uses functions of sscanf type, in the control lines format we can use special macros which open into necessary modifiers for different systems. An example:

Code:

```
// 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

In the low-quality code there are often magic numbers the mere presence of which is dangerous. During the migration of the code on the 64-bit platform these magic numbers may make the code inefficient if they participate in operations of calculation of address, objects size or bit operations.

Table N2 contains basic magic numbers which may influence the workability of an application on a new platform.

Code:

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.

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 look at some errors related to the use of magic numbers. The most frequent is a record of number values of type sizes.

Code:

```
1) size_t ArraySize=N * 4;
   intptr_t *Array=(intptr_t *)malloc(ArraySize);
```

Code:

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

Code:

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

Let's suppose that in all cases the size of the types used is always 4 bytes. The correction of the code is in the use of sizeof() operator.

Code:

```
1) size_t ArraySize=N * sizeof(intptr_t);
   intptr_t *Array=(intptr_t *)malloc(ArraySize);
```

Code:

```
2) size_t values[ARRAY_SIZE];
   memset(values, ARRAY_SIZE * sizeof(size_t), 0);
or
   memset(values, sizeof(values), 0); //preferred alternative
```

Code:

```
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.

Code:

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

This is the incorrect code in the case of the 64-bit system. Such errors are very unpleasant because the record 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 to correct this code using #ifdef or a special macro.

Code:

```
#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 evidently use -1 value. An example of the incorrect code using 0xffffffff value as an error sign.

Code:

```
#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 is the result of "(size_t)(-1)" value 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:

Code:

```
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 the failure of the execution 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 cannot refuse them for some reason or think it unreasonable to correct the code fundamentally just use fair value -1.

Code:

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

4. Bit shifting operations.

Bit shifting operations can cause a lot of troubles during the port from the 32-bit system on the 64-bit one if used inattentively. Let's begin with an example of a function which defines the bit you've chosen as 1 in a variable of memsize type.

Code:

```
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 to define bits with numbers from 0 to 31. After the program port on the 64-bit platform it becomes necessary to define bits from 0 to 63. What do you think which value will the following call of SetBitN(0, 32) function return? If you think that 0x100000000 the authors is glad because he hasn't prepared this article in vain. You'll get 0.

Pay attention that "1" has int type and during the shift on 32 positions an overflow will occur. To correct the code it is necessary to make the constant "1" of the same type as the variable mask.

Code:

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

5. Storing of pointer addresses.

A large number of 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. In the environment with the data ILP32 usual integers and pointers have the same size. Unfortunately the 32-bit code is based on this supposition everywhere. Pointers are often casted to int, unsigned int and other types improper to fulfill address calculations.

You should understand exactly that one should use only memsize types for integer pointers form. Preference should be given to uintptr_t type for it shows intentions more clearly and makes the code more portable saving it from changes in future

Let's look at two small examples.

Code:

```
1) char *p;
   p=(char *) ((int)p & PAGEOFFSET);
```

Code:

```
2) DWORD tmp=(DWORD)malloc(ArraySize);
   ...
   int *ptr=(int *)tmp;
```

The both examples do not take into account that the pointer size may differ from 32-bits. They use the explicit type conversion which truncates high bits in the pointer and this is surely a mistake on the 64-bit system. Here are the corrected variants which use integer memsize type intptr_t and DWORD_PTR to store pointer addresses:

Code:

```
1) char *p;
   p=(char *) ((intptr_t)p & PAGEOFFSET);
```

Code:

```
2) DWORD_PTR tmp=(DWORD_PTR)malloc(ArraySize);
   ...
   int *ptr=(int *)tmp;
```

The danger of the two examples studied is that the fail in the program may be found much time later. The program may work absolutely correctly with a small data size on the 64-bit system while the truncated addresses lie in first 4 Gb of memory. And 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.

Code:

```
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.

Code:

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

There are situations when storing of the pointer address into a 32-bit type is just necessary. Mostly 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.

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

Code:

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

6. 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 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 senseless.

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 number form of the type without using explicit conversions.

Code:

```
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. Changing member m_n on the 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.

Code:

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

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

Code:

```
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 it 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 overwrite the code in the following way.

Code:

```
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]];
```

7. Change of an array type

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

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 result is different in 32-bit and 64-bit variants. On the 32-bit system the access to the array items is fulfilled correctly for sizes of size_t and int coincide and we see the output "2 2".

On the 64-bit system we got "2 17179869187" in the output for it is value 17179869187 which is situated in the first item of sizetPtr array. In some cases we need this very behavior but usually it is an error.

The correction of the described situation consists in the refuse of dangerous type conversions by modernizing the program. Another variant is to create a new array and copy values of the original one into it.

8. Virtual functions with arguments of memsize type

If there are big derived class graphs with virtual functions in your program, there is a risk to use inattentively arguments of different types but 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 type unsigned. So this code will be incorrect on the 64-bit system.

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

Code:

```
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 firstly it was being developed for Microsoft Visual C++ 6.0 when WinHelp function in CWinApp class had the following prototype:

Code:

```
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 in the replacement of DWORD type with DWORD_PTR type. On the 32-bit system the program will work absolutely correctly for here types DWORD and DWORD_PTR coincide. Troubles will appear during the compilation of the given code for the 64-bit platform. We'll gave two functions with the same name but different parameters and as a result the user's code won't be executed.

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

Code:

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

9. Serialization and data exchange.

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

Mostly the errors of this kind consist in the serialization of memsize types and data exchange operations using them.

Code:

```
1) size_t PixelCount;
   fread(&PixelCount, sizeof(PixelCount), 1, inFile);
```

Code:

```
2) __int32 value_1;
   SSIZE_T value_2;
   inputStream >> value_1 >> value_2;
```

Code:

```
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 ignore of the byte order.

The use of types of volatile size.

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

The use of such types provides data portability between programs on different platforms although it needs the use 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.

Code:

```
1) size_t PixelCount;
   __uint32 tmp;
   fread(&tmp, sizeof(tmp), 1, inFile);
   PixelCount=static_cast<size_t>(tmp);
```

Code:

```
2) __int32 value_1;
   __int32 value_2;
   inputStream >> value_1 >> value_2;
```

Code:

```
3) time_t time;
   __uint32 tmp=static_cast<__uint32>(time);
   PackToBuffer(MemoryBuf, &tmp, sizeof(tmp));
```

But the given variant of correction cannot be the best. During the port on the 64-bit system the program may process a large number of data and the use of 32-bit types in the data may become a serious obstacle. 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 a different data presentation. Most frequently it is related to a different byte order.

The byte order is a method of recording of bytes of multibyte numbers. The little-endian order means that the recording begins with the lowest byte and ends with the highest one. This record order was acceptable in the memory of PCs with x86-processors. The big-endian order - the recording begins 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 processors Motorola 68000, SPARC.

While developing the binary interface or data format you should remember about the byte order. 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 byte order and the little-endian one you may use functions htonl(), htons(), bswap_64 etc.

10. Pointer address arithmetic.

The first example.

Code:

```
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 could always work correctly on the 32-bit platform for 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 the explicit type conversion in expressions which carry pointers. Using the explicit type conversion we can rewrite the code in the following way.

Code:

```
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 those inaccurate programs which work on larger data sizes face troubles we have to disappoint you. Let's look at an interesting code for working with an array containing only 5 items. The second example works in the 32-bit variant and does not work in the 64-bit one.

Code:

```
int A=-2;
unsigned B=1;
int array[5]={ 1, 2, 3, 4, 5 };
int *ptr=array + 3;

//Invalid pointer value on 64-bit platform
ptr=ptr + (A + B);

//Access violation on 64-bit platform
printf("%i\n", *ptr);
```

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

- 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 addition will take 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 while getting access to the item of this pointer we'll face troubles.

To avoid the shown situation, 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:

Code:

```
ptr=ptr + (ptrdiff_t(A) + ptrdiff_t(B));
```

Code:

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

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

Code:

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

Yes, this code will work but it is bad due to some reasons:

1. It will teach you inaccurate work with pointers. After a while you may forget nuances and made a mistake by making one of the variables of unsigned type.
2. Using of non-memsize types together with pointers is potentially dangerous. Suppose variable Delta of int type participates in the 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 danger automatically.

11. Arrays indexing.

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

In programming in language C and then C++ a practice formed to use in the constructions of the following kind variables of int/unsigned types:

Code:

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

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

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

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

Code:

```
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 source, and the possibility to use more than 4 Gb of memory on the 64-bit architecture improves the calculation speed greatly. In the 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 alike GetCell which provide access to the necessary items. But the given code will work correctly only with the arrays containing less than INT_MAX items. The reason for that 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:

Code:

```
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" for 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 the explicit type conversion of every variable memsize type:

Code:

```
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:

Code:

```
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];
}
```

12. 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 be related to the change of the input values rate. Let's study some examples.

Code:

```
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 can demand more iterations. As far as the values of the variable Index lie in range [0..UINT_MAX] the condition "Index != Count" will never be executed and this causes the infinite loop.

Another frequent error is a record of the expressions of the following kind:

Code:

```
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. Search and correction of the given code is made more difficult because compilers do not show any warning messages on it as a rule. From the point of view of C++ language this is absolutely correct construction. Several variables of int type are multiplied and after that the result is implicitly converted to intptr_t type and assignment occurs.

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

Code:

```
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 kind

Code:

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

does not promise the right result. It promises only that "intptr_t(x) * y * z" expression will have intptr_t type. The right result shown by this expression in the example is good luck caused by a particular compiler version and occasional process.

The order of the calculation of an expression with operators of the same priority is not defined. To be more exact, the compiler can calculate sub-expressions in such an order which it considers to be more efficient even if sub-expressions cause (side effect). The order of the appearing of side effects is not defined. Expressions including communicative and association operations (*, +, &, |, ^), may be converted in a free way even if there are brackets. To assign the strict order of the calculation of the expression it is necessary to use the explicit temporary variable.

That's why if the result of the expression should be of memsize type, only memsize types must participate in the expression. The right variant:

Code:

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

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

Mixed use of types may occur in the change of the program logic.

Code:

```
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 just the other way round - the variable val_2 is extended to ptrdiff_t type. In this case the expression "-1 > 1" is checked.

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

Code:

```
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");
```

13. Implicit type conversions while using functions.

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

Code:

```
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;
```

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

Code:

```
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.

Code:

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

The result of "end - begin" expression have ptrdiff_t type. As far as the function returns unsigned type the implicit type conversion occurs during which high bits of the results get lost. Thus if pointers begin and end address the beginning and the end of the array according to the larger UINT_MAX (4Gb), the function will return the incorrect value.

And here it is one more example but now we'll observe not the returned value but the formal function argument.

Code:

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

Does not 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 value 0xFFFFFFFF and from unsigned type to ptrdiff_t type.

14. Overload functions.

During the port of 32-bit programs on the 64-bit platform the change of the logic of its work may be found which is related to the use of overload 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 method may be useful as, for example, in the following code:

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 has a potential danger. Imagine a program in which a class for organizing stack is used for some aims. The peculiarity of this class is that it allows to store value of different types.

Code:

```
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 and then chose form the stack of values of different types (ptrdiff_t and int). 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 stack began to include more bytes than it extract out later.

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

15. 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 4 bytes and the 64-bit item at the border 8 bytes. The try to work with unaligned data on processors IA-64 (Itanium) as it is shown in the following example, will cause exception.

Code:

```
#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 indicate this to the compiler. For example, you may use a special macro UNALIGNED:

Code:

```
#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 decision is not efficient for the access to the unaligned data will be several times slower. A better result may be achieved when you arrange up to 32-bit, 16-bit and 8-bit items in 64-bit data items.

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

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

Code:

```
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 of the object of 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 correct calculation of the size should look as follows:

Code:

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

In this code we see the shift of the last structure member and sum up this shift and its size. The shift of a member of the structure or a class may be recognized when macro offsetof or FIELD_OFFSET 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 it is the example of the code with the correct calculation of the structure member address:

Code:

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

16. The use of outdated functions and predefined constants.

While developing a 64-bit application, be sure to bear in mind the changes of the environment in which it will be performed. Some functions will become outdated and it will be necessary to replace them with some variants. GetWindowLong is a good example of such function in the Windows operation system. Pay your attention to the constants referring to the interaction with the environment in which the program is functioning. In Windows the lines containing "system32" or "Program Files" will be suspect.

17. Explicit type conversions.

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

Error diagnosis.

The diagnosis of the errors occurring while porting 32-bit programs on 64-bit systems is a difficult task. The port of a not very quality code written without taking into account peculiarities of other architectures, may demand a lot of time and efforts. That's why we'll pay some attention to the description of methods and means which may simplify this task.

Unit test.

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

Unit test may not allow you to check the new ranges of input values which become accessible on 64-bit systems. Unit tests are originally developed in such a way that they can be passed in a short 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 for this function in tests may be called many times with different sets of input values. But suppose you have a 64-bit variant of the program. And now the function we study is processing more than 4 Gb of data. Surely there appears a necessity to raise the input size of an array in the tests either up to sizes more than 4 Gb. The problem is that the time of passing the tests will increase greatly in such a case.

That's why while modifying the sets of tests keep in mind the compromise between speed of passing unit tests and the fullness of the checks. Fortunately, there are other methods which can help you to make sure of the efficiency of your applications.

Code review.

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

But this approach to the analysis of the original code has on significant disadvantage. It demands a lot of time and because of this it is actually 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 where a fetch of potentially dangerous places is created for a programmer so that he could carry out the further analysis.

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

Built-in means of compilers.

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

Unfortunately, the possibilities they provide are often not enough and you should not rely only on them. But in any case it is highly recommended to 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 quality and safety of the program code. The basic difficulty related to the use of static analyzers consists in the fact that they generate quite a lot false warning messages about potential errors. Programmers being lazy by nature use this argument to find some way not to correct the found errors. In Microsoft this problem is solved by including necessarily the found errors in the bug tracking system. Thus a programmer just cannot choose between the correction of the code and tries to avoid this.

We think that such strict rules are justified. The profit of 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 of debugging and testing.

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

The authors know 3 static analyzers which are supposed to have means of diagnosing errors related to the port of programs on 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 great efficiency.

1. Gimpel Software PC-Lint (http://www.gimpel.com). This analyzer has a large list of supported platforms and is the static analyzer of general purpose. It allows to catch errors while porting program on the architecture with LP64 data model. The advantage is the possibility to organize strict control over the type conversions. What the disadvantages are concerned the only one of them is the absence of the environment but it may be corrected by using the external Riverblade Visual Lint.
2. Parasoft C++test (http://www.parasoft.com). Another well-know static analyzer of general purpose. It has the support of a lot of device and program platforms too. It has built-in environment which simplifies the work and setting of the analysis rules greatly. As well as PC-Lint it is intended for LP64 data model.
3. Viva64 (http://www.viva64.com). Unlike other analyzers it is intended to work with Windows (LLP64) data model. It is integrated into the development environment Visual Studio 2005. It is intended only for diagnosing problems related to the port of programs on 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 for you and will help to 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'll describe typical errors the more profitable will be the use of our experience and getting help.

Resources.

1. Chandra Shekar. Extend your application's reach from 32-bit to 64-bit environments.http://enterprisenetworksandservers....y/art.php?2670
2. Converting 32-bit Applications Into 64-bit Applications: Things to Consider.http://developers.sun.com/sunstudio/...P64Issues.html

3. Andrew Josey. Data Size Neutrality and 64-bit Support.http://www.unix.org/whitepapers/64bit.html
4. Harsha S. Adiga. Porting Linux applications to 64-bit systems.http://www.ibm.com/developerworks/library/l-port64.html
5. Transitioning C and C++ programs to the 64-bit data model.http://devresource.hp.com/drc/STK/do...4datamodel.jsp
6. Porting an Application to 64-bit Linux on HP Integrity Servers.http://h21007.www2.hp.com/dspp/files...whitepaper.pdf
7. Stan Murawski. Beyond Windows XP: Get Ready Now for the Upcoming 64-Bit Version of Windows. http://msdn.microsoft.com/msdnmag/issues/01/11/XP64/
8. Steve Graegert. 64-bit Data Models Explained. http://digitalether.de/index.php?opt...d=31&Itemid=46
9. Updating list of resources devoted to the development of 64-bit applications.http://www.viva64.com/links.php

Source:http://www.go4expert.com/articles/typical-errors-porting-cpp-code-64-bit-t6830/