|  |
| --- |
| D:\chenbilun\我的文档\Work\文档\文档相关\logo-透明.png |
| 溢出专题讨论 |
|  |
| **ADE-001**  **Revision1.10**  **2016-8-30** |

|  |
| --- |
|  |
| Shenzhen SmartChain Technology Co.,LtdPublished |
| [www.dacrs.com](http://www.dacrs.com) |

# 目录

[目录 1](#_Toc460329475)

[1 溢出原理 2](#_Toc460329476)

[2 溢出例子 4](#_Toc460329477)

[2.1 加法溢出 4](#_Toc460329478)

[3 取绝对值时溢出 4](#_Toc460329479)

[3.1 除法溢出 6](#_Toc460329480)

[3.2 减法溢出 8](#_Toc460329481)

[4 溢出判断 11](#_Toc460329482)

[4.1 补码加减法运算 11](#_Toc460329483)

[4.2 除法溢出 15](#_Toc460329484)

[5 溢出防范 16](#_Toc460329485)

[6 附录： 17](#_Toc460329486)

# 

# 溢出原理

1）计算机中整数都有一个宽度，当试图保存一个比它可以表示的最大值还大的数时，就会发生整数溢出。运算结果超出机器所能表示的范围，即为溢出。（通常讲的最高位进位或借位就溢出，这种表述是不正确的）

2）整数溢出将导致“不确定行为”。比如完全忽略该溢出或终止进程。大多数编译器都会忽略这种溢出，这可能会导致不确定或错误的值保存在整数变量中。

3）常见整型的表示范围：



4）整数的存储方式

在实际使用的过程中，人们经常使用的整数表达(编码)方式是10进制，因此，通常用10进制来表示整数。但是计算机不能直接处理10进制，所以在计算机中，整数以2进制进行存储。但是，2进制又太长，不好表达，因此，有时候又用16进制表示，因为2进制和16进制能够很方便地转换。

对于有符号的整数，在32位系统下，正数的存储方式就是其二进制，如2，在内存中的存储方式为：

00000000 00000000 00000000 00000010

16进制表示为：0x00000002

负数的存储一般是其补码(绝对值取反+1)。

如-2的存储方法是： 首先取绝对值，2，

存储方式为： 00000000 00000000 00000000 00000010

取反，成为： 11111111 11111111 11111111 11111101

加1，成为： 11111111 11111111 11111111 11111110

16进制表示为：0xFFFFFFFE。

一般说来，如果最高位置1，这个变量就被解释为负数；如果置0，这个变量就解释为正数。 还有一种是无符号整数，不管最高位是1还是0，都理解为正数。

如： 11111111 11111111 11111111 11111110

如果理解为无符号类型，将是一个很大的数。

什么情况下会出现整数溢出呢? 由于整数在内存里面保存在一个固定长度 (在本章中使用32位)的空间内，它能存储的最大值就是固定的，当尝试去存储一个数，而这个数又大于这个固定的最大值时，将会导致整数溢出。

举个例子，有两个无符号的整数，num1和num2，两个数都是32位长，首先赋值给num1 一个32位整数的最大值，num2被赋值为1。

然后让num1和num2相加，然后存储结果到第3个无符号32位的整数num3，

代码如下：

num1 = 0xFFFFFFFF;

num2 = 0x00000001;

num3 = num1 + num2;

很显然，

num1的值是：11111111 11111111 11111111 11111111；

num2的值是：00000000 00000000 00000000 00000001；

两者相加，得到结果为：00000000 00000000 00000000 00000000。

因此，num3中的值是0，发生了整数溢出。

此时，如果一个整数用来计算一些敏感数值，如缓冲区大小或数组索引，就会产生潜在的危险。

不过，并不是所有的整数溢出都可以被利用，毕竟，整数溢出并没有改写额外的内存；但是，在有些情况下，整数溢出将会导致“不能确定的行为”，由于整数溢出出现之后，很难被被立即察觉，比较难用一个有效的办法去判断是否出现或者可能出现整数溢出。 就发现的难度而言，和缓冲区溢出相比，整数溢出更加难被发现。因此，即使是审核过的代码，有时候也难以幸免。

综上所述，一言以蔽之，整数溢出是尝试存储一个很大的数到一个变量中，由于这个变量能够存储的数值范围太小，不足以存储那个很大的数，造成溢出。

# 溢出例子

## 加法溢出

unsigned \_\_int8 i1 = 250;

unsigned \_\_int8 i2 = 150;

* unsigned \_\_int8 ret = i1 + i2; //overflow! 144

char i = (char)64 + (char)64; //overflow!//-128

0x8000

# 取绝对值时溢出

template <typename T>

unsigned \_\_int64 AbsVal(T t)

{

if(t < 0)

return (unsigned \_\_int64)-t;

return t;

}

unsigned \_\_int64 result = AbsVal(-128);

cout << result << endl;//128

int a = 0x80000000;

floatl a = 0x4588e

result = AbsVal(a);

cout << result << endl; //0xffffffff80000000

3）double和int64\_t比较时，由于浮点数精度问题导致两数相等

double d = pow(2, 63);

int64\_t i = INT64\_MAX;

printf("%f > %lld is %s\n", d, i, d > i ? "true" : "false");

double d2 = 9223372036854775808.000000;

int64\_t i2 = 9223372036854775807;

printf("%s\n", d2 > i2 ? "true" : "false");

## 除法溢出

char c\_a = -128;

char c\_b = -1;

char c\_result = c\_a / c\_b;

printf("c\_a = %d, c\_b = %d, c\_result = %d\n", c\_a, c\_b, c\_result); //-128

5）运算时，类型提升导致两数相等

signed char a = 0xff;

unsigned int b = 0xffffffff;

printf("result = %d\n", ((unsigned int) a) < b );

6）默认类型为int，

void foo()

{

int i;

for(i = 0; i < 0xffff; i++)

....

}

7）运算时，类型提升导致两数相等

unsigned int l = 0xffffffff;

char c = -1;

if(c == l)

printf("Why is -1 equal to 4 billion???\n");

if((\_\_int64)c == (\_\_int64)l)

printf("Why is -1 equal to 4 billion???\n");

else

printf("Why doesn't the compiler upcast to 64-bits when needed?\n");

符号位扩展：

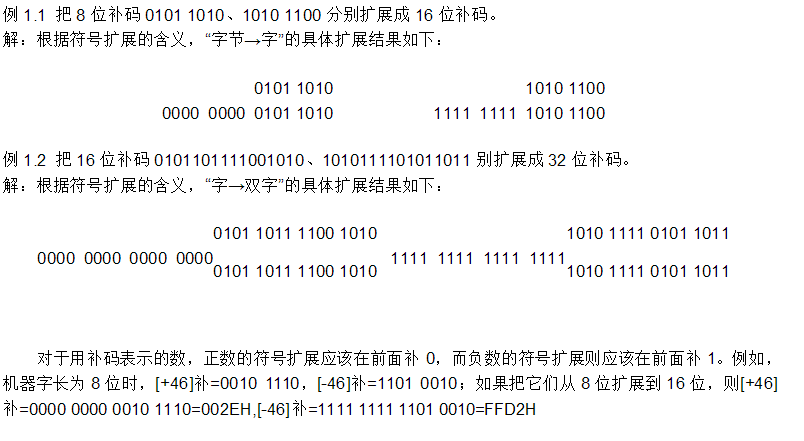
所谓符号扩展问题是指一个数从位数较少扩展到位数较多(如从8位扩展到16位，或从16位扩展到32位)时应该注意的问题。

有符号数是用最高位是0或1来标记正负的，如果最高位是0（如8位数中的第7位，从0位开始算的）表示正数，而是1表示负数。16位数中的第15位控制符号。符号数扩展实称为带符号扩展。只是位数的扩展，不能改变原值的！

如0000 1101这个数是带符号数为13，扩展为16位时，一个16位数也要是13的！而这个数是0000 0000 0000 1101就可以了！所以正数的带符号扩展前边是加0，这只是一个规律而不是本质，本质就是数大小不改变！

而1000 1101带符号数不是-13的！而是将其取补加1就是负数结果，即-0111 0010 + 1，结果就是-115，如果将这个带符号数扩展时，只有16位1111 1111 1000 1101才是-115，扩展只是表示范围大了，而不是改变数值的。如果是正数前8位是0，如果是负数，前8位是1，这样才是带符号扩展的。这不是本质，只是一个规律而已！

在汇编语言中，我们经常要对字/字节的数据进行操作。当把“字节”转换成“字”，或“字”转换成“双字”时，就需要进行符号扩展。符号扩展的具体操作就是把已知信息的最高位扩展到所有更高位。



## 减法溢出

char a1 = 126;

unsigned char b = 255;

char ret = 0;

ret = a1 - b;

printf("%d\n", ret); //127

0111 1110

1111 111 1

0111 1111

三.实例分析

C++中常用的数据类型有整形，字符型，浮点型（单精度和双精度）等等。其中基本整形（按长度递增的顺序排列）分别是 char、short、int和long，其中每种类型都有符号版本和无符号版本，因此总共有8种类型可供选择。但是char类型常用来表示字符，而不是数字。

在C++中short、int、long它们的长度：

  short至少16位（8位=1个字节）；

  int至少与short一样长；

  long至少32位，且至少与int一样长。

介绍了一些基本知识，现在来说说溢出吧：

看下面这段代码

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **short** a = SHRT\_MAX; //此处 SHRT\_MAX表示有符号short数据类型的最大值
6. unsigned **short** b = a;
7. cout << "a = " << a << " b = " << b << endl;
8. a++;
9. b++;
10. cout << "a = " << a << " b = " << b << endl;
11. **return** 0;
12. }

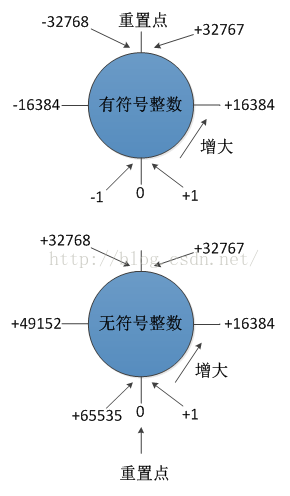
输出结果为：

a = 32767 b = 32767  
a = -32768 b = 32768

a和b都只进行了一次自加运算为什么得到的结果不同呢？

因为a是有符号的整形变量而b是无符号的整形变量而

一个short变量和一个unsigned short变量它们的长度都为16位，short变量的取值范范围是-32768~+32767（-2^15 ~ 2^15-1），而unsigned的取值范围是0~65535（0~2^16-1）



上面这幅图表示出了有符号和无符号short类型的的溢出情况：

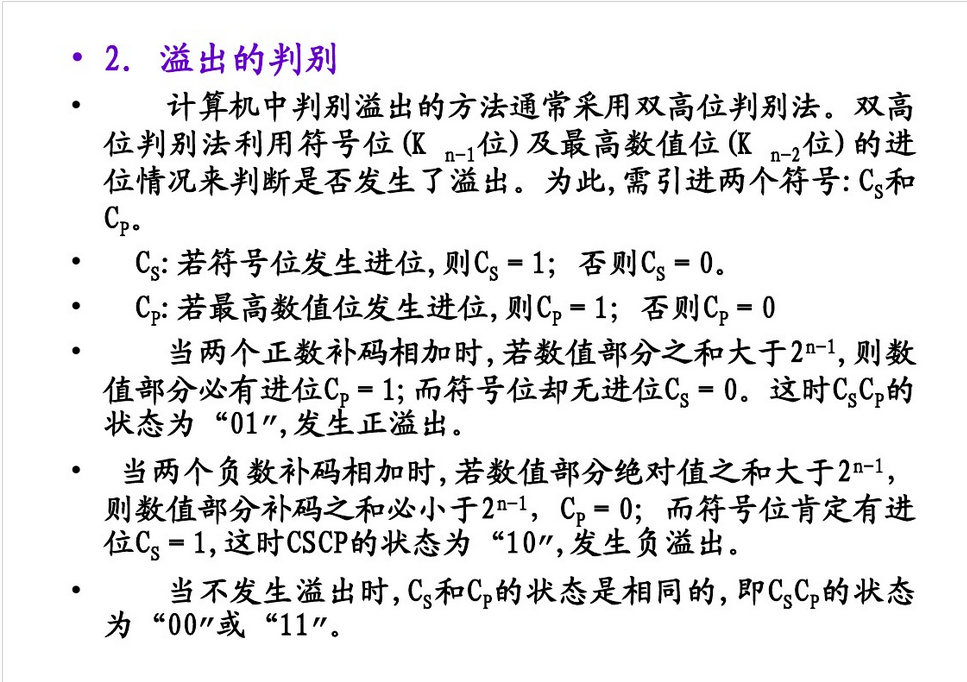
当是有符号整数时，其最大值为+32767，再+1之后就会溢出为-32768；当为无符号整数时就无影响，继续+1为32768。

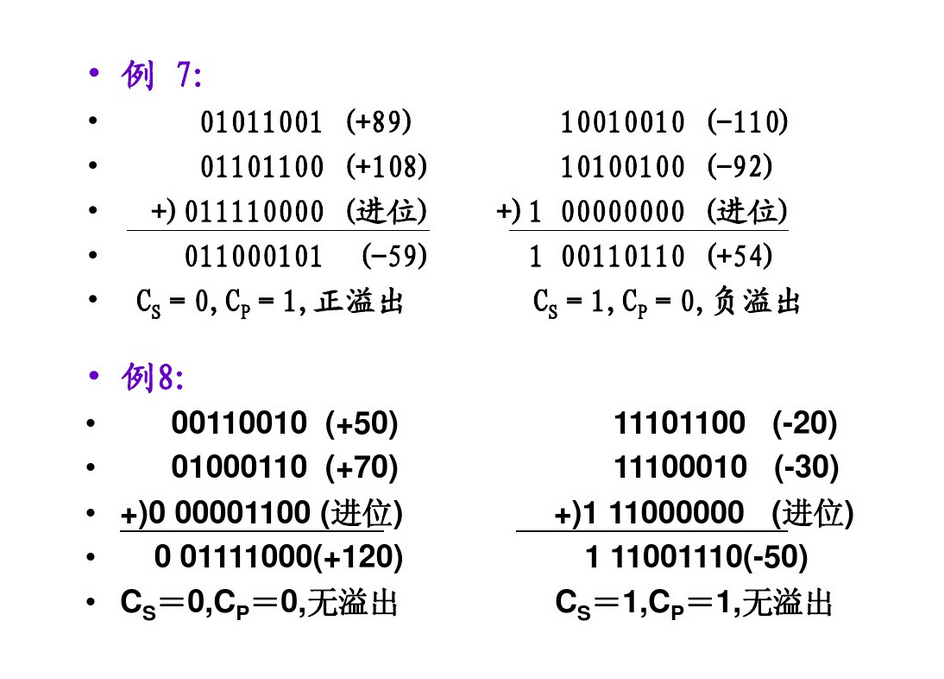
# 溢出判断

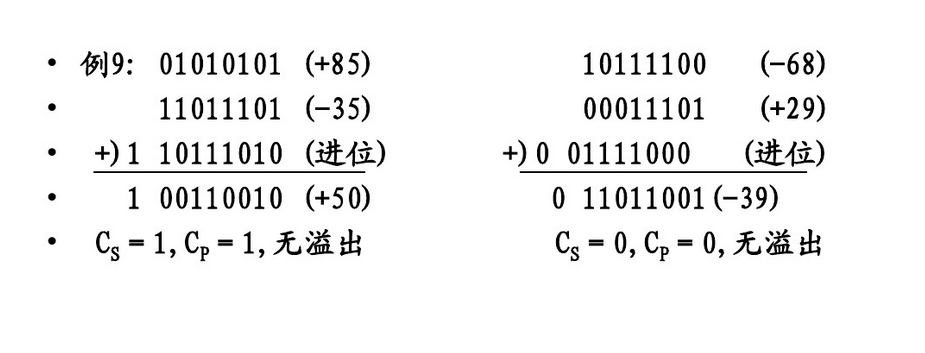
## 补码加减法运算

1）两个异号数相加或两个同号数相减，不会溢出。

2）两个同号数相加或两个异号数相减，有可能溢出。







提升类型去判断溢出：

加法：

ushort a, b, c;

c = a + b;

将转换为：

int tmp = a + b;

if( tmp <= (int)USHORT\_MAX)

c = (ushort)tmp;

else

溢出

减法：

short a;

unsigned short b;

short c = a - b;

将转换为：

int tmp = a - b;

if( tmp >= (int)SHORT\_MIN)

c = (short)tmp;

else

溢出

上述判断只是特例，根据a, b的类型，有无符号，溢出判断比较复杂，但基本都是提升类型后，再根据具体情况判断。

## 除法溢出

如：

int a, b, c;

c = a / b;

只有a = INT\_MIN并且b = -1时，才会溢出。

3.乘法溢出

一般通过类型提升去判断溢出如：

int a, b, c;

c = a \* b;

将转换为：

int64 tmp = (int64)a \* (int64)b;

if(tmp > INT32\_MAX || tmp < INT32\_MIN)

溢出!!!

else

c = (int)tmp;

上述判断只是特例，根据a, b的类型，有无符号，溢出判断比较复杂，但基本都是提升类型后，再根据具体情况判断。

# 溢出防范

整数溢出是非常危险的，部分原因是因为它在发生后不可能被发现，也就是说，一个整数溢出发生了，应用程序并不知道它的计算是错误的。因此应用程序在假定它是正确的情况下，会继续运行下去。在安全的系统中，这种结果具有巨大的危害，有时甚至能够造成系统崩溃。

整数溢出在很大程度上，是因为不好的编程习惯和疏忽。解决整数溢出的方案，主要是编程之前必须进行详细的预测，多考虑一些问题， 在编程时将各种问题考虑到并且进行相应的处理。

1）形成关于特殊数据输入的意识，比如之前先确定最大和最小输入，充分考虑各种数据的取值范围，使用合适的数据类型。

2）尽量避免对两个整数相加之后，再取结果比较，上例应该改成：

if(MAX\_INFO > len1 + len2) 改为：

if(MAX\_INFO - len1 > len2)

......

3）在使用变量申请内存，或者作为数组下标时，注意对越界的监测。

4）尽量不要在不同范围的数据类型之间进行赋值。

# 附录：

（摘自INT32-C. Ensure that operations on signed integers do not result in overflow）

1）Addition

Addition is between two operands of arithmetic type or between a pointer to an object type and an integer type. This rule applies only to addition between two operands of arithmetic type. (See ARR37-C. Do not add or subtract an integer to a pointer to a non-array object and ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.)

Incrementing is equivalent to adding 1.

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the addition of the signed operands si\_a and si\_b:

void func(signed int si\_a, signed int si\_b) {

signed int sum = si\_a + si\_b;

/\* ... \*/

}

Compliant Solution

This compliant solution ensures that the addition operation cannot overflow, regardless of representation:

#include <limits.h>

void f(signed int si\_a, signed int si\_b) {

signed int sum;

if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||

((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {

/\* Handle error \*/

} else {

sum = si\_a + si\_b;

}

/\* ... \*/

}

2）Subtraction

Subtraction is between two operands of arithmetic type, two pointers to qualified or unqualified versions of compatible object types, or a pointer to an object type and an integer type. This rule applies only to subtraction between two operands of arithmetic type. (See ARR36-C. Do not subtract or compare two pointers that do not refer to the same array, ARR37-C. Do not add or subtract an integer to a pointer to a non-array object, and ARR30-C. Do not form or use out-of-bounds pointers or array subscripts for information about pointer subtraction.)

Decrementing is equivalent to subtracting 1.

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the subtraction of the signed operands si\_a and si\_b:

void func(signed int si\_a, signed int si\_b) {

signed int diff = si\_a - si\_b;

/\* ... \*/

}

Compliant Solution

This compliant solution tests the operands of the subtraction to guarantee there is no possibility of signed overflow, regardless of representation:

#include <limits.h>

void func(signed int si\_a, signed int si\_b) {

signed int diff;

if ((si\_b > 0 && si\_a < INT\_MIN + si\_b) ||

(si\_b < 0 && si\_a > INT\_MAX + si\_b)) {

/\* Handle error \*/

} else {

diff = si\_a - si\_b;

}

/\* ... \*/

}

3）Multiplication

Multiplication is between two operands of arithmetic type.

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the multiplication of the signed operands si\_a and si\_b:

void func(signed int si\_a, signed int si\_b) {

signed int result = si\_a \* si\_b;

/\* ... \*/

}

Compliant Solution

The product of two operands can always be represented using twice the number of bits than exist in the precision of the larger of the two operands. This compliant solution eliminates signed overflow on systems where long long is at least twice the precision of int:

#include <stddef.h>

#include <assert.h>

#include <limits.h>

#include <inttypes.h>

extern size\_t popcount(uintmax\_t);

#define PRECISION(umax\_value) popcount(umax\_value)

void func(signed int si\_a, signed int si\_b) {

signed int result;

signed long long tmp;

assert(PRECISION(ULLONG\_MAX) >= 2 \* PRECISION(UINT\_MAX));

tmp = (signed long long)si\_a \* (signed long long)si\_b;

/\*

\* If the product cannot be represented as a 32-bit integer,

\* handle as an error condition.

\*/

if ((tmp > INT\_MAX) || (tmp < INT\_MIN)) {

/\* Handle error \*/

} else {

result = (int)tmp;

}

/\* ... \*/

}

The assertion fails if long long has less than twice the precision of int. The PRECISION() macro and popcount() function provide the correct precision for any integer type. (See INT35-C. Use correct integer precisions.)

Compliant Solution

The following portable compliant solution can be used with any conforming implementation, including those that do not have an integer type that is at least twice the precision of int:

#include <limits.h>

void func(signed int si\_a, signed int si\_b) {

signed int result;

if (si\_a > 0) { /\* si\_a is positive \*/

if (si\_b > 0) { /\* si\_a and si\_b are positive \*/

if (si\_a > (INT\_MAX / si\_b)) {

/\* Handle error \*/

}

} else { /\* si\_a positive, si\_b nonpositive \*/

if (si\_b < (INT\_MIN / si\_a)) {

/\* Handle error \*/

}

} /\* si\_a positive, si\_b nonpositive \*/

} else { /\* si\_a is nonpositive \*/

if (si\_b > 0) { /\* si\_a is nonpositive, si\_b is positive \*/

if (si\_a < (INT\_MIN / si\_b)) {

/\* Handle error \*/

}

} else { /\* si\_a and si\_b are nonpositive \*/

if ( (si\_a != 0) && (si\_b < (INT\_MAX / si\_a))) {

/\* Handle error \*/

}

} /\* End if si\_a and si\_b are nonpositive \*/

} /\* End if si\_a is nonpositive \*/

result = si\_a \* si\_b;

}

4）Division

Division is between two operands of arithmetic type. Overflow can occur during two's complement signed integer division when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to ?1. Division operations are also susceptible to divide-by-zero errors. (See INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.)

Noncompliant Code Example

This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent a signed integer overflow error in two's-complement.

void func(signed long s\_a, signed long s\_b) {

signed long result;

if (s\_b == 0) {

/\* Handle error \*/

} else {

result = s\_a / s\_b;

}

/\* ... \*/

}

Implementation Details

On the x86-32 architecture, overflow results in a fault, which can be exploited as a denial-of-service attack.

Compliant Solution

This compliant solution eliminates the possibility of divide-by-zero errors or signed overflow:

#include <limits.h>

void func(signed long s\_a, signed long s\_b) {

signed long result;

if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {

/\* Handle error \*/

} else {

result = s\_a / s\_b;

}

/\* ... \*/

}

Remainder

The remainder operator provides the remainder when two operands of integer type are divided. Because many platforms implement remainder and division in the same instruction, the remainder operator is also susceptible to arithmetic overflow and division by zero. (See INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.)

Noncompliant Code Example

Many hardware architectures implement remainder as part of the division operator, which can overflow. Overflow can occur during a remainder operation when the dividend is equal to the minimum (negative) value for the signed integer type and the divisor is equal to ?1. It occurs even though the result of such a remainder operation is mathematically 0. This noncompliant code example prevents divide-by-zero errors in compliance with INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors but does not prevent integer overflow:

void func(signed long s\_a, signed long s\_b) {

signed long result;

if (s\_b == 0) {

/\* Handle error \*/

} else {

result = s\_a % s\_b;

}

/\* ... \*/

}

Implementation Details

On x86-32 platforms, the remainder operator for signed integers is implemented by the idiv instruction code, along with the divide operator. Because LONG\_MIN / ?1 overflows, it results in a software exception with LONG\_MIN % ?1 as well.

Compliant Solution

This compliant solution also tests the remainder operands to guarantee there is no possibility of an overflow:

#include <limits.h>

void func(signed long s\_a, signed long s\_b) {

signed long result;

if ((s\_b == 0 ) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {

/\* Handle error \*/

} else {

result = s\_a % s\_b;

}

/\* ... \*/

}

5）Unary Negation

The unary negation operator takes an operand of arithmetic type. Overflow can occur during two's complement unary negation when the operand is equal to the minimum (negative) value for the signed integer type.

Noncompliant Code Example

This noncompliant code example can result in a signed integer overflow during the unary negation of the signed operand s\_a:

void func(signed long s\_a) {

signed long result = -s\_a;

/\* ... \*/

}

Compliant Solution

This compliant solution tests the negation operation to guarantee there is no possibility of signed overflow:

#include <limits.h>

void func(signed long s\_a) {

signed long result;

if (s\_a == LONG\_MIN) {

/\* Handle error \*/

} else {

result = -s\_a;

}

/\* ... \*/

}

附录2：参看微软SafeInt3.hpp，该实现对各种情况都有讨论。此类已经集成在VS2010中。