INT36-C. Converting a pointer to integer or integer to pointer

Although programmers often use integers and pointers interchangeably in C, pointer-to-integer and integer-to-pointer conversions are implementation-defined.

Conversions between integers and pointers can have undesired consequences depending on the implementation. According to the C Standard, subclause 6.3.2.3 [ISO/IEC 9899:2011],

> *An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.*

> *Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.*

Do not convert an integer type to a pointer type if the resulting pointer is incorrectly aligned, does not point to an entity of the referenced type, or is a trap representation.

Do not convert a pointer type to an integer type if the result cannot be represented in the integer type. (See undefined behavior 24.)

The mapping between pointers and integers must be consistent with the addressing structure of the execution environment. Issues may arise, for example, on architectures that have a segmented memory model.

Noncompliant Code Example

The size of a pointer can be greater than the size of an integer, such as in an implementation where pointers are 64 bits and unsigned integers are 32 bits. This code example is noncompliant on such implementations because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type:

```
void f(void) {
  char *ptr;
  /* ... */
  unsigned int number = (unsigned int)ptr;
  /* ... */
}
```

Compliant Solution

Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` and back with no change in value. (See **INT36-EX2**.) The C Standard guarantees that a pointer to `void` may be converted to or from a pointer to any object type and back again and that the result must compare equal to the original pointer. Consequently, converting directly from a `char *` pointer to a `uintptr_t`, as in this compliant solution, is allowed on implementations that support the `uintptr_t` type.

```
#include <stdint.h>

void f(void) {
  char *ptr;
  /* ... */
  uintptr_t number = (uintptr_t)ptr;
  /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the pointer `ptr` is converted to an integer value. The high-order 9 bits of the number are used to hold a flag value, and the result is converted back into a pointer. This example is noncompliant on an implementation where pointers are 64 bits and unsigned integers are 32 bits because the result of converting the 64-bit `ptr` cannot be represented in the 32-bit integer type.

```
void func(unsigned int flag) {
  char *ptr;
  /* ... */
  unsigned int number = (unsigned int)ptr;
  number = (number & 0x7fffff) | (flag << 23);
  ptr = (char *)number;
}
```

A similar scheme was used in early versions of Emacs, limiting its portability and preventing the ability to edit files larger than 8MB.

Compliant Solution

This compliant solution uses a `struct` to provide storage for both the pointer and the flag value. This solution is portable to machines of different word sizes, both smaller and larger than 32 bits, working even when pointers cannot be represented in any integer type.

```
struct ptrflag {
  char *pointer;
  unsigned int flag : 9;
} ptrflag;

void func(unsigned int flag) {
  char *ptr;
  /* ... */
  ptrflag.pointer = ptr;
  ptrflag.flag = flag;
}
```

Noncompliant Code Example

It is sometimes necessary to access memory at a specific location, requiring a literal integer to pointer conversion. In this noncompliant code, a pointer is set directly to an integer constant, where it is unknown whether the result will be as intended:

```
unsigned int *g(void) {
  unsigned int *ptr = 0xdeadbeef;
  /* ... */
  return ptr;
}
```

The result of this assignment is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

Compliant Solution

Adding an explicit cast may help the compiler convert the integer value into a valid pointer. A common technique is to assign the integer to a volatile-qualified object of type `intptr_t` or `uintptr_t` and then assign the integer value to the pointer:

```
unsigned int *g(void) {
  volatile uintptr_t iptr = 0xdeadbeef;
  unsigned int *ptr = (unsigned int *)iptr;
  /* ... */
  return ptr;
}
```

Exceptions

INT36-C-EX1: A null pointer can be converted to an integer; it takes on the value 0. Likewise, the integer value 0 can be converted to a pointer; it becomes the null pointer.

INT36-C-EX2: Any valid pointer to `void` can be converted to `intptr_t` or `uintptr_t` or their underlying types and back again with no change in value. Use of underlying types instead of `intptr_t` or `uintptr_t` is discouraged, however, because it limits portability.

```
#include <assert.h>
#include <stdint.h>

void h(void) {
  intptr_t i = (intptr_t)(void *)&i;
  uintptr_t j = (uintptr_t)(void *)&j;

  void *ip = (void *)i;
  void *jp = (void *)j;

  assert(ip == &i);
  assert(jp == &j);
}
```

Risk Assessment

Converting from pointer to integer or vice versa results in code that is not portable and may create unexpected pointers to invalid memory locations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
INT36-C	Low	Probable	High	P2	L3

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**pointer-integral-cast** **pointer-integral-cast-implicit** **function-pointer-integer-cast** **function-pointer-integer-cast-implicit**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-INT36**	Fully implemented
Clang	3.9	`-Wint-to-pointer-cast`, `-Wint-conversion`	Can detect some instances of this rule, but does not detect all
CodeSonar	5.2p0	**LANG.CAST.PC.CONST2PTR** **LANG.CAST.PC.INT**	Conversion: integer constant to pointer Conversion: pointer/integer
Compass/ROSE			
Coverity	2017.07	**PW.POINTER_CONVERSION_LOSES_BITS**	Fully implemented
Klocwork	2018	**MISRA.CAST.OBJ_PTR_TO_INT.2012**	
LDRA tool suite	9.7.1	**439 S, 440 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-INT36-a** **CERT_C-INT36-b**	An object with integer type or pointer to void type shall not be converted to an object with pointer type A conversion should not be performed between a pointer to object type and an integer type other than 'uintptr_t' or 'intptr_t'
Polyspace Bug Finder	R2019b	CERT C: Rule INT36-C	Checks for unsafe conversion between pointer and integer (rule fully covered)
PRQA QA-C	9.7	**0303, 0305, 0306, 0309, 0324,** **0326, 0360, 0361, 0362**	Partially implemented
PRQA QA-C++	4.4	**3040, 3041, 3042, 3043, 3044,** **3045, 3046, 3047, 3048**	
PVS-Studio	6.23	**V542**, **V566**, **V647**	

RuleChecker	19.04	**pointer-integral-cast** **pointer-integral-cast-implicit** **function-pointer-integer-cast** **function-pointer-integer-cast-implicit**	Fully checked
SonarQube C/C++ Plugin	3.11	**S1767**	Partially implemented

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C	INT11-CPP. Take care when converting from pointer to integer or integer to pointer	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772: 2013	Pointer Casting and Pointer Type Changes [HFC]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961: 2013	Converting a pointer to integer or integer to pointer [intptrconv]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-587, Assignment of a Fixed Address to a Pointer	2017-07-07: CERT: Partial overlap
CWE 2.11	CWE-704	2017-06-14: CERT: Rule subset of CWE
CWE 2.11	CWE-758	2017-07-07: CERT: Rule subset of CWE
CWE 3.1	CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer	2018-10-19:CERT:None
CWE 3.1	CWE-466, Return of Pointer Value Outside of Expected Range	2018-10-19:CERT:None

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-758 and INT36-C

Independent(INT34-C, INT36-C, MEM30-C, MSC37-C, FLP32-C, EXP33-C, EXP30-C, ERR34-C, ARR32-C)

CWE-758 = Union(INT36-C, list) where list =

- Undefined behavior that results from anything other than integer <-> pointer conversion

CWE-704 and INT36-C

CWE-704 = Union(INT36-C, list) where list =

- Incorrect (?) typecast that is not between integers and pointers

CWE-587 and INT36-C

Intersection(CWE-587, INT36-C) =

- Setting a pointer to an integer value that is ill-defined (trap representation, improperly aligned, mis-typed, etc)

CWE-587 – INT36-C =

- Setting a pointer to a valid integer value (eg points to an object of the correct t ype)

INT36-C – CWE-587 =

- Illegal pointer-to-integer conversion

Intersection(INT36-C,CWE-466) =

Intersection(INT36-C,CWE-466) =

An example explaining the above two equations follows:

```
static char x[3];
char* foo() {
  int x_int = (int) x; // x_int = 999 eg
  return x_int + 5; // returns 1004 , violates CWE 466
}
...
int y_int = foo(); // violates CWE-466
char* y = (char*) y_int; //  // well-defined but y may be invalid, violates INT36-C
char c = *y; // indeterminate value, out-of-bounds read, violates CWE-119
```

Bibliography

[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"