The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

6.3.2.3 Pointers

pointer to void converted to/from

A pointer to **void** may be converted to or from a pointer to any incomplete or object type. 743

Commentary

generic pointer

Pointer to **void** is intended to implement the concept of a generic pointer (other terms include abstract pointer, anonymous pointer), which may point at an object having any type. Constraints on pointer conversions are discussed elsewhere.

pointer conversion constraints

C++

5.2.9p10 *An rvalue of type "pointer to* cv **void***" can be explicitly converted to a pointer to object type.*

object types

In C++ incomplete types, other than *cv* **void**, are included in the set of object types.

In C++ the conversion has to be explicit, while in C it can be implicit. C source code that relies on an implicit conversion being performed by a translator will not be accepted by a C++ translator.

The suggested resolution to SC22/WG21 DR #137 proposes changing the above sentence, from 5.2.9p10, to:

Proposed change to C++ *An rvalue of type "pointer to* cv1 **void***" can be converted to an rvalue of type "pointer to* cv2 *T", where T is an object type and* cv2 *is the same cv-qualification as, or greater cv-qualification than,* cv1*.*

If this proposal is adopted, a pointer-to qualified type will no longer, in C++, be implicitly converted unless the destination type is at least as well qualified.

Common Implementations

pointer to void same representation and alignment as

A pointer to **void** is required to have the same representation as a pointer to **char**. On existing processors where there is a difference between pointer representations, it is between pointer-to character types and pointers-to other types. So the above requirement is probably made possible by the previous representation requirement.

Coding Guidelines

The pointed-to type, of a pointer type, is useful information. It can be used by a translator to perform type checking and it tells readers of the source something about the pointed-to object. Converting a pointer type to pointer to **void** throws away this information, while converting a pointer to **void** to another pointer type adds information. Should this conversion process be signaled by an explicit cast, or is an implicit conversion acceptable? This issue is discussed elsewhere. The main difference between conversions involving pointer to **void** and other conversions is that they are primarily about a change of information content, not a change of value.

operand convert automatically

Example

```
1  extern int ei;
2  extern unsigned char euc;
3
4  void f(void *p)
5  {
6  (*(char *)p)++;
7  }
8
9  void q(void)
10 {
11 unsigned char *p_uc = &euc;
```

```
12  int *p_i = &ei;
13
14  f(p_uc);
15  f((void *)p_uc);
16
17  f(p_i);
18  f((void *)p_i);
19  }
```

744 A pointer to any incomplete or object type may be converted to a pointer to **void** and back again;

pointer converted to pointer to void

Commentary

This is actually a requirement on the implementation that is worded as a permission for the language user. It implies that any pointer can be converted to pointer to **void** without loss of information about which storage location is being referred to. There is no requirement that the two pointer representations be the same (except for character types), only that the converted original can be converted back to a value that compares equal to the original (see next C sentence). This sentence is a special case of wording given elsewhere, except that here there are no alignment restrictions.

pointer to void same representation and alignment as 758 pointer converted to pointer to different object or type

The purpose of this requirement is to support the passing of different types of pointers as function arguments, allowing a single function to handle multiple types rather than requiring multiple functions each handling a single type. This single function is passed sufficient information to enable it to interpret the pointed-to object in a meaningful way, or at least pass the pointer on to another function that does. Using a union type would be less flexible, since a new member would have to be added to the union type every time a new pointer type was passed to that function (which would be impossible to do in the interface to a third-party library).

C++

> *Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.* 5.2.10p7

The C++ wording is more general than that for C. A pointer can be converted to any pointer type and back again, delivering the original value, provided the relative alignments are no stricter.

Source developed using a C++ translator may make use of pointer conversion sequences that are not required to be supported by a C translator.

Coding Guidelines

Developers who want to convert a pointer-to X into a pointer-to Y usually do so directly, using a single cast. A conversion path that goes via pointer to **void** is not seen as providing a more reliable result (it doesn't) or a more *type correct* way of doing things (it isn't).

A pointer converted to pointer to **void** and subsequently converted to a different pointer type is either unintended behavior by the developer or a case of deliberate type punning by the developer. Guideline recommendations are not intended to deal with constructs that are obviously faults, and the issue of type punning is discussed elsewhere.

guidelines not faults
type punning union

745 the result shall compare equal to the original pointer.

converted via pointer to void compare equal

Commentary

This is a requirement on the implementation. It is not the same as one specifying that the representation shall be bit-for-bit identical. The standard says nothing about the internal representation of pointers. It is possible for two pointers to compare equal and have different bit representations.

pointer segmented architecture

C++

5.2.9p10 *A value of type pointer to object converted to "pointer to* `cv` ***void****" and back to the original pointer type will have its original value.*

In C++ incomplete types, other than *cv* **void**, are included in the set of object types.

Common Implementations

pointer segmented architecture

Implementations don't usually change the representation of a pointer value when it is cast to another pointer type. However, when the target processor has a segmented architecture, there may be a library call that checks that the value is in canonical form. On most implementations, it is very likely that when converted back the result will be bit-for-bit identical to the original value.

---

pointer converting qualified/unqualified

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; 746

Commentary

pointer 758 converted to pointer to different object or type

This specification is asymmetric. The specification for when a pointer to a q-qualified type is converted to a pointer to a non-q-qualified version of the type is handled by the general wording on converting object and incomplete types. Converting a pointer to a non-q-qualified type to a pointer to a q-qualified version of the type is going to possibly:

- Limit the operations that will be performed on it (for the **const** qualifier).
- Prevent the translator from optimizing away accesses (for the **volatile** qualifier) and reusing values already held in a register.
- Improve the quality of optimizations performed by a translator (for the **restrict** qualifier) or cause undefined behavior.

restrict undefined behavior

When converting from a pointer-to non-q-qualified to a pointer-to q-qualified type, under what circumstances is it possible to guarantee that the following requirements (sometimes known as *type safe* requirements) are met:

1. A program will not attempt to change the value of an object defined using the **const** qualifier.
2. A program will not access an object defined with the **volatile** in a manner that is not a volatile-qualified access.

For instance, allowing the implicit conversion `char **` ⇒ `char const **` would violate one of the above requirements. It could lead to a const-qualified object being modified:

```
1  char const    c  = 'X';
2  char        *pc       ;
3  char const **pcc = &pc; /* Assume this was permitted. */
4  *pcc = &c;
5  *pc = 'Y';              /* Modifies c, a const-qualified object. */
```

This example shows that simply adding the **const** qualifier within a pointer type does not automatically ensure type safety.

Smith[7] formally proved the necessary conditions that need to hold for the two requirements listed above to be met (provided the type system was not overridden using an explicit cast). Let `T1` and `T2` be the following types, respectively:

$$T cv_{1,n} * \ldots cv_{1,1} * cv_{1,0} \quad (746.1)$$

$$T cv_{2,n} * \ldots cv_{2,1} * cv_{2,0} \quad (746.2)$$

where cv is one of nothing, **const**, **volatile**, or **const volatile**. If $T1$ and $T2$ are different, but similar pointer types, both requirements listed above are met if the following conditions are also met:

1. For every $j > 0$, if **const** is in $cv_{1,j}$, then **const** is in $cv_{2,j}$; and if **volatile** is in $cv_{1,j}$, then **volatile** is in $cv_{2,j}$.
2. If $cv_{1,j}$ and $cv_{2,j}$ are different, then **const** is in every $cv_{2,k}$ for $0 < k < j$.

The first condition requires the converted-to type to contain at least the same qualifiers as the type being converted from for all but the outermost pointer type. For the second case, $cv_{1,j}$ and $cv_{2,j}$ are different when the converted-to type contains any qualifier not contained in $cv_{2,j}$. The requirement on $cv_{2,j}$ including the **const** qualifier ensures that, once converted, a volatile-qualified value cannot be accessed.

```
1  char **    ⇒   char const * const *                  /* Is type safe.     */
2  char **    ⇒   char volatile * volatile *            /* Is not type safe. */
3  char **    ⇒   char const volatile * const volatile *  /* Is type safe.     */
```

There is an important difference of intent between the two qualifiers **const** and **volatile**, and the qualifier **restrict**. The **restrict** qualifier is not subject to some of the constraints on type qualifiers in conversions and assignments of pointers that apply to the other two qualifiers. This is because performing the semantically appropriate checks on uses of the **restrict** qualifier is likely to cause a significant overhead for implementations. The burden of enforcement is placed on developers, to use **restrict** only when they know that the conditions it asserts hold true.

restrict requires all accesses

Applying the ideas of type safety, when assigning pointers, to restrict-qualified types leads to the conclusion that it is *safe* to assign to a less **restrict** qualified type (in the formalism used above $j = 0$ rather than $j > 0$) and *unsafe* to assign to a more restrict-qualified type (the opposite usage to that applying to uses of const- or volatile-qualified types). However, it is the *unsafe* conversions that are likely to be of most use in practice (e.g., when passing arguments to functions containing computationally intensive loops).

C++

An rvalue of type "pointer to `cv1 T`*" can be converted to an rvalue of type "pointer to* `cv2 T`*" if "*`cv2 T`*" is more cv-qualified than "*`cv1 T`*."* 4.4p1

An rvalue of type "pointer to member of X of type `cv1 T`*" can be converted to an rvalue of type "pointer to member of X of type* `cv2 T`*" if "*`cv2 T`*" is more cv-qualified than "*`cv1 T`*."* 4.4p2

Other Languages

The qualifiers available in C do not endow additional access permissions to objects having that type, they only reduce them. This situation is not always true in other languages where specific cases sometimes need to be called out.

Coding Guidelines

This C sentence describes the behavior when a conversion adds qualifiers to a pointer type. The standard does not contain a sentence that specifically discusses the behavior when conversions remove qualifiers from a pointer type. This issue is discussed elsewhere.

758 pointer converted to pointer to different object or type

quali-fied/unqualified pointer compare equal

pointer to quali-fied/unqualified types

the values stored in the original and converted pointers shall compare equal. 747

Commentary

This is a requirement on the implementation. Qualifiers provide additional information on the properties of objects, they do not affect the representation of the pointer used.

C++

3.9.2p3 *Pointers to cv-qualified and cv-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.9).*

By specifying layout-compatible types, not the same type, the C++ Standard restricts the freedom of implementations more than C99 does.

Common Implementations

Qualifiers provide information that enables translators to optimize, or not, programs. It is very unlikely that the conversion operation itself will result in any machine code being generated. The likely impact (quality of machine code generated by a translator) will be on subsequent accesses using the pointer value, or the pointed-to object.

null pointer constant

An integer constant expression with the value 0, or such an expression cast to type **`void *`**, is called a *null pointer constant*.[55] 748

Commentary

This defines the term *null pointer constant*, which is rarely used in its full form by developers. The shortened term *null pointer* is often used to cover this special case (which technically it does do).

null pointer constant 749

null pointer constant

Note that a constant expression is required— a value known at translation time. A value of zero computed during program execution is not a null pointer constant. The reason for this distinction is that the token `0` (and the sequence of tokens that includes a cast operation) is purely an external representation, in the source code, of something called a null pointer constant. The internal, execution-time value can be any sequence of bits. The integer constant expression `(1-1)`, for example, has the value `0` and could be used to denote the null pointer constant (the response to DR #261 confirmed this interpretation). However, while the value of the expression `(x-x)`, where `x` is an initialized integer object, might be known at translation time it does not denote a null pointer constant.

Developers (and implementations) often treat any cast of the value `0` to a pointer type as a null pointer constant. For instance, the expression `(char *)0` is often used to represent a null pointer constant in source code. While it is a null pointer, it is not the null pointer constant. Such usage is often a hangover from the prestandard days, before the type **`void`** was introduced.

C++

4.10p1 *A null pointer constant is an integral constant expression (5.19) rvalue of integer type that evaluates to zero.*

The C++ Standard only supports the use of an integer constant expression with value `0`, as a null pointer constant. A program that explicitly uses the pointer cast form need not be conforming C++; it depends on the context in which it occurs. Use of the implementation-provided `NULL` macro avoids this compatibility problem by leaving it up to the implementation to use the appropriate value.

The C++ Standard specifies the restriction that a null pointer constant can only be created at translation time.

Footnote 64

64) Converting an integral constant expression (5.19) with value zero always yields a null pointer (4.10), but converting other expressions that happen to have value zero need not yield a null pointer.

Other Languages

Many languages use a reserved word, or keyword (**nil** is commonly seen), to represent the concept and value of the null pointer.

Common Implementations

Many implementations use an execution-time representation of all bits zero as the value of the null pointer constant. Implementations for processors that use a segmented memory architecture have a number of choices for the representation of the null pointer. The Inmos Transputer[5] has a signed address space with zero in the middle and uses the value 0x80000000 as its execution-time representation; the IBM/390 running CICS also used this value. Non-zero values were also used on Prime and Honeywell/Multics.[1]

pointer
segmented
architecture

Coding Guidelines

Technically there is a distinction between a null pointer constant and a null pointer. In practice implementations rarely make a distinction. It is debatable whether there is a worthwhile benefit in trying to educate developers to distinguish between the two terms.

749 null pointer

As the following two points show attempting to chose between using `0` and `(void *)0` involves trading off many costs and benefits:

cognitive
switch

- The literal `0` is usually thought about in arithmetic, rather than pointer, terms. A cognitive switch is needed to think of it as a null pointer constant. Casting the integer constant to pointer to **void** may remove the need for a cognitive switch, but without a lot of practice (to automate the process) recognizing the token sequence `(void *)0` will need some amount of conscious effort.
- When searching source using automated tools (e.g., `grep`), matches against the literal `0` will not always denote the null pointer constant. Matches against `(void *)0` will always denote the null pointer constant; however, there are alternative character sequences denoting the same quantity (e.g., `(void*)0`).

However, there is a third alternative. The `NULL` macro is defined in a number of standard library headers. Use of this macro has the advantages that the name is suggestive of its purpose; it simplifies searches (although the same sequence of characters can occur in other names) and source remains C++ compatible. It is likely that one of the standard library headers, defining it, has already been **#include**d by a translation unit. If not, the cost of adding a **#include** preprocessor directive is minimal.

Cg 748.1 The null pointer constant shall only be represented in the visible form of source code by the `NULL` macro.

Casting the value of an object having an integer type to a pointer type makes use of undefined behavior. While the null pointer constant is often represented during program execution, by all bits zero, there is no requirement that any particular representation be used.

Example

```
1  #include <stdio.h>
2
3  extern int glob = 3;
4
5  char *p_c_1 = 0;
6  char *p_c_2 = (void *)0;
```

```
7  char *p_c_3 = 9 - 8 - 1;
8  char *p_c_4 = (1 == 2) && (3 == 4);
9  char *p_c_5 = NULL;
10
11 void f(void)
12 {
13 if (NULL != (void *)(glob - glob))
14    printf("Surprising, but possible\n");
15 }
```

null pointer

If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function. 749

Commentary

This is a requirement on the implementation. The null pointer constant is often the value used to indicate that a pointer is not pointing at an object; for instance, on a linked list or other data structure, that there are no more objects on the list. Similarly, the null pointer constant is used to indicate that no function is referred to by a pointer-to function.

The `(void *)0` form of representing the null pointer constant already has a pointer type.

C++

4.10p1 *A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of pointer to object or pointer to function type.*

4.11p1 *A null pointer constant (4.10) can be converted to a pointer to member type; the result is the null member pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant.*

Presumably *distinguishable* means that the pointers will compare unequal.

5.10p1 *Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array.*

From which we can deduce that a null pointer constant cannot point one past the end of an object either.

Other Languages

Languages that have a construct similar to the null pointer constant usually allow it to be converted to different pointer types (in Pascal it is a reserved word, **`nil`**).

Common Implementations

All bits zero is a convenient execution-time representation of the null pointer constant for many implementations because it is invariably the lowest address in storage. (The INMOS Transputer[5] had a signed address space, which placed zero in the middle.) Although there may be program bootstrap information at this location, it is unlikely that any objects or functions will be placed here. Many operating systems leave this storage location unused because experience has shown that program faults sometimes cause values to be written into the location specified by the null pointer constant (the more developer-oriented environments try to raise an exception when that location is accessed).

Another implementation technique, when the host environment does not include address zero as part of a processes address space, is to create an object (sometimes called `__null`) as part of the standard library. All references to the null pointer constant refer to this object, whose address will compare unequal to any other object or function.

750 Conversion of a null pointer to another pointer type yields a null pointer of that type.

null pointer conversion yields null pointer

Commentary

This is a requirement on the implementation. It is the only situation where the standard guarantees that a pointer to any type may be converted to a pointer to a completely different type, and will deliver a defined result. (Conversion to pointer to **void** requires the pointer to be converted back to the original type before the value is defined.)

743 pointer to void converted to/from
744 pointer converted to pointer to void

C90

The C90 Standard was reworded to clarify the intent by the response to DR #158.

Common Implementations

In most implementations this conversion leaves the representation of the null pointer unmodified and is essentially a no-op (at execution time). However, on a segmented architecture, the situation is not always so simple. For instance, in an implementation that supports both near and far pointers, a number of possible representation decisions may be made, including:

pointer segmented architecture

- The *near* null pointer constant might have offset zero, while the *far* null pointer constant might use an offset of zero and a segment value equal to the segment value used for all near pointers (which is likely to be nonzero because host environments tend to put their own code and data at low storage locations). When the integer constant zero is converted to a *far* null pointer constant representation, translators can generate code to create the appropriate segment offset. When objects having an integer type are converted to pointers, implementations have to decide whether they are going to treat the value as a meaningless bit pattern, or whether they are going to check for and special-case the value zero. A similar decision on treating pointer values as bit patterns or checking for special values has to be made when generating code to convert pointers to integers.
- The *near* null pointer constant might have offset zero, while the *far* null pointer constant might have both segment and offset values of zero. In this case the representation is always all bits zero; however, there are two addresses in storage that are treated as being the null pointer constant. When two pointers that refer to the same object are subtracted, the result is zero; developers invariably extend this behavior to include the subtraction of two null pointer constants (although this behavior is not guaranteed by the standard because the null pointer constant does not refer to an object). Because the null pointer constant may be represented using two different addresses in storage, implementation either has to generate code to detect when two null pointer constants are being subtracted or be willing to fail to meet developer expectations in some cases.

Debugging implementations that make use of a processor's automatic hardware checking (if available) of address accesses may have a separate representation (e.g., offset zero and a unique segment value) for every null pointer constant in the source. (This enables runtime checks to trace back dereferences of the null pointer constant to the point in the source that created the constant.)

751 Any two null pointers shall compare equal.

null pointer compare equal

Commentary

This is a requirement on the implementation. They compare equal using the equality operator (they are not required to have the same, bit-for-bit, representations) irrespective of their original pointer type. (There may be equality operator constraints that effectively require one or the other of the operands to be cast before the equality operation is performed.)

equality operators constraints

C++

Two null pointer values of the same type shall compare equal.

4.11p1 *Two null member pointer values of the same type shall compare equal.*

The C wording does not restrict the null pointers from being the same type.

4.10p3 *The null pointer value is converted to the null pointer value of the destination type.*

This handles pointers to class type. The other cases are handled in 5.2.7p4, 5.2.9p8, 5.2.10p8, and 5.2.11p6.

Other Languages

Null pointers are usually considered a special value and compare equal, independent of the pointer type.

---

integer permission to convert to pointer

An integer may be converted to any pointer type. 752

Commentary

This is the one situation where the standard allows a basic type to be converted to a derived type. Prior to the C90 Standard, a pointer value had to be input as an integer value that was then cast to a pointer type. The %p qualifier was added to the `scanf` library function to overcome this problem.

Other Languages

Many other languages permit some form of integer-to-pointer type conversion. Although supporting this kind of conversion is frowned on by supporters of strongly typed languages, practical issues (it is essential in some applications) mean that only the more academic languages provide no such conversion path. Java does not permit an integer to be converted to a reference type. Such a conversion would allow the security in the Java execution-time system to be circumvented.

Common Implementations

Most implementations do not generate any machine code for this conversion. The bits in the integer value are simply interpreted as having a pointer value.

Coding Guidelines

The standard may allow this conversion to be made, but why would a developer ever want to do it? Accessing specific (at known addresses) storage locations might be one reason. Some translators include an extension that enables developers to specify an object's storage location in its definition. But this extension is only available in a few translators. Casting an integer value to a pointer delivers predictable results on a wider range of translators. The dangers of converting integer values to pointers tend to be talked about more often than the conversion is performed in practice. When such a conversion is used, it is often essential.

object specifying address

In some development environments (e.g., safety-critical) this conversion is sometimes considered sufficiently dangerous that it is often banned outright. However, while use of the construct might be banned on paper, experience suggests that such usage still occurs. The justifications given to support these usages, where it is necessary to access specific storage locations, can involve lots of impressive-sounding technical terms. Such obfuscation serves no purpose. Clear rationale is in everybody's best interests. Such conversions make use of representation information and, as such, are covered by a guideline recommendation. The following wording is given as a possible deviation to this representation usage guideline recommendation:

representation information using ??

Dev ??

Values whose integer type has a rank that is less than or equal to that of the type `intptr_t` may be converted to a pointer type.

Dev ??

Integer values may be converted to a pointer type provided all such conversions are commented in the source code and accompanied by a rationale document.

Example

```
1   char *p_c = (char *)43;
```

753 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.[56)]

integer-to-pointer implementation-defined

Commentary

This is a complete list of all the issues that need to be considered when relying on such a conversion. The integer constant zero is a special case. C does not provide any mechanisms for checking that a pointer is well-formed in the sense of being correctly aligned or pointing at an object whose lifetime has not terminated.

748 null pointer constant

There is a long-standing developer assumption that a pointer type occupies the same number of value bits as the type **long**. The growing availability of processors using 64-bit addressing support for type **long long** (which often means a 32-bit representation is used for type **long**) means that uncertainty over the minimum rank of the integer type needed to represent a pointer value is likely to grow. The introduction of the typedef name `intptr_t` is intended to solve this problem.

C++

The C++ Standard specifies the following behavior for the **reinterpret_cast**, which is equivalent to the C cast operator in many contexts.

> *A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.*

5.2.10p5

The C++ Standard provides a guarantee— a round path conversion via an integer type of sufficient size (provided one exists) delivers the original value. Source developed using a C++ translator may contain constructs whose behavior is implementation-defined in C.

The C++ Standard does not discuss trap representations for anything other than floating-point types.

Other Languages

The above issues are not specific to C, but generic to languages designed to run on a wide range of processors. Anything that could go wrong in one language could go equally wrong in another.

Common Implementations

Most implementations take the pattern of bits in the integer value and interpret them as a pointer value. They do not perform any additional processing on the bit pattern (address representations used by implementations are discussed elsewhere).

byte address unique

Just because the equality `sizeof(char *) == sizeof(int *)` holds does not mean that the same representations are used. For instance, one Data General MV/Eclipse C compiler used different representations, although the sizes of the pointer types were the same. The IAR PICmicro compiler[2] provides access to more than 10 different kinds of banked storage. Pointers to this storage can be 1, 2, or 3 bytes in size. On such a host, the developer needs to be aware of the storage bank referred to by a particular pointer. Converting an integer type to a pointer-to type using the ILE C translator for the IBM AS/400[4] causes the right four bytes of the 16-byte pointer to hold the integer value. This pointer value cannot be dereferenced.

Coding Guidelines

?? representation information using

If the guideline recommendation dealing with use of representation information is followed, then this conversion will not occur. If there are deviations to this guideline recommendation, it is the developer's responsibility to ensure that the expected result is obtained. Possible deviations and situations to watch out for are not sufficiently common to enable deviations to be specified here.

pointer permission to convert to integer

Any pointer type may be converted to an integer type. 754

integer 752 permission to convert to pointer

Commentary

The inverse operation to the one described in a previous C sentence. Prior to the introduction of the C Standard, C90, pointer values had to be converted to integers before they could be output. The `%p` conversion specifier was added to the `printf` family of functions to overcome this problem. Pointer types are sometimes converted to integers because developers want to perform bit manipulation on the value, often followed by a conversion back to the original pointer type. Application areas that perform this kind of manipulation include garbage collectors, tagged pointers (where unused address bits are used to hold information on the pointed-to object), and other storage management functions.

C++

5.2.10p4 *A pointer can be explicitly converted to any integral type large enough to hold it.*

The C++ wording is more restrictive than C, which has no requirement that the integer type be large enough to hold the pointer.

integer-to-pointer 753 implementation-defined

While the specification of the conversion behaviors differ between C++ and C (undefined vs. implementation-defined, respectively), differences in the processor architecture is likely to play a larger role in the value of the converted result.

Other Languages

There are fewer practical reasons for wanting to convert a pointer-to integer type than the reverse conversion, and languages are less likely to support this kind of conversion. Java does not permit a reference type to be converted to an integer. Such a conversion would allow the security in the Java execution-time system to be circumvented.

Common Implementations

Most implementations do not generate any machine code for this conversion. The bits in the integer value are simply interpreted as having a pointer value.

Coding Guidelines

Pointer-to-integer conversions differ from integer-to-pointer conversions in that there are likely to be fewer cases where a developer would need to perform them. The C language supports pointer arithmetic, so it is not necessary to convert the value to an integer type before incrementing or decrementing it. One use (perhaps the only real one) of such a conversion is the need to perform some nonarithmetic operation, for instance a bitwise operation, on the underlying representation. If bit manipulation is to be carried out, then type punning (e.g., a union type) is an alternative implementation strategy.

type punning union

representation information using 752 integer permission to convert to pointer

Such conversions make use of representation information and as such are covered by a guideline recommendation. The following wording is given as a possible deviation to this representation usage guideline recommendation (it mirrors those suggested for integer-to-pointer conversions):

Dev ?? A value having a pointer type may only be converted to an integer type whose rank is greater than or equal to that of the type `intptr_t`.

Dev ?? A value having a pointer type may only be converted to an integer type provided all such conversions are commented in the source code and accompanied by a rationale document.

Example

```
1  #include <stdint.h>
2
3  extern char *c_p;
4
5  void f(void)
6  {
7  unsigned char l_uc  = (unsigned char)c_p;
8  int           l_i   = (int)          c_p;
9  intptr_t      l_ip  = (intptr_t)     c_p;
10 uintptr_t     l_uip = (uintptr_t)    c_p;
11 long long     l_ll  = (long long)    c_p;
12 }
```

Usage

Usage information on pointer conversions is given elsewhere (see Table 758.1 and Figure **??**).

---

755 Except as previously specified, the result is implementation-defined.

Commentary

There is both implementation-defined and undefined behavior involved here. By specifying implementation-defined behavior, if the result can be represented, the Committee is requiring that the developer have access to information on the converted representation (in the accompanying documentation).

756 pointer conversion undefined behavior
implementation-defined behavior

Other Languages

Like integer-to-pointer conversions the issues tend to be generic to all languages. In many cases language specification tends to be closer to the C term *undefined behavior* rather than implementation-defined behavior.

753 integer-to-pointer implementation-defined

Common Implementations

Like integer-to-pointer conversions, most implementations simply take the value representation of a pointer and interpret the bits (usually the least significant ones if the integer type contains fewer value bits) as the appropriate integer type. Choosing a mapping such that the round trip of pointer-to-integer conversion, followed by integer-to-pointer conversion returns a pointer that refers to the same storage location is an objective for some implementations.

753 integer-to-pointer implementation-defined

Some pointer representations contain status information, such as supervisor bits, as well as storage location information. The extent to which this information is included in the integer value can depend on the number of bits available in the value representation. Converting a pointer-to function type to an integer type using the ILE C translator for the IBM AS/400[4] always produces the result 0.

---

756 If the result cannot be represented in the integer type, the behavior is undefined.

pointer conversion undefined behavior

Commentary

The first requirement is that the integer type have enough bits to enable it to represent all of the information present in the pointer value. There then needs to be a mapping from pointer type values to the integer type.

C90

If the space provided is not long enough, the behavior is undefined.

The C99 specification has moved away from basing the specification on storage to a more general one based on representation.

C++

The C++ Standard does not explicitly specify any behavior when the result cannot be represented in the integer type. (The wording in 5.2.10p4 applies to "any integral type large enough to hold it.")

Common Implementations

Most implementations exhibit no special behavior if the result cannot be represented. In most cases a selection of bits (usually the least significant) from the pointer value is returned as the result.

The result need not be in the range of values of any integer type. 757

Commentary

This statement means that there is no requirement on the implementation to provide either an integer type capable of representing all the information in a pointer value, or a mapping if such an integer type is available

C90

The C90 requirement was based on sufficient bits being available, not representable ranges.

C++

There is no equivalent permission given in the C++ Standard.

Common Implementations

Implementation vendors invariably do their best to ensure that such a mapping is supported by their implementations. The IBM AS/400[3] uses 16 bytes to represent a pointer value (the option `datamodel` can be used to change this to 8 bytes), much larger than can be represented in any of the integer types available on that host.

Coding Guidelines

While the typedef names `intptr_t` and `uintptr_t` (specified in the library subsection) provide a mechanism for portably representing pointer values in an integer type, support for them is optional. An implementation is not required to provide definitions for them in the **<stdint.h>** header.

---

pointer converted to pointer to different object or type

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. 758

Commentary

This is a more general case of the permission given for pointer to **void** conversions.

pointer 743 to void converted to/from

C++

The C++ Standard states this in 5.2.9p5, 5.2.9p8, 5.2.10p7 (where the wording is very similar to the C wording), and 5.2.11p10.

Other Languages

Many languages that support pointer types and a cast operator also support some form of conversion between different pointers.

Coding Guidelines

Experience suggests that there are two main reasons for developers wanting to convert a pointer-to object type to point at different object types:

1. Pointer-to object types are often converted to pointer-to character type. A pointer to **unsigned char** is commonly used as a method of accessing all of the bytes in an object's representation (e.g., so they can be read or written to a different object, file, or communications link). The standard specifies requirements on the representation of the type **unsigned char** in order to support this common usage. The definition of effective type also recognizes this usage,

unsigned char pure binary effective type

2. Pointers to different structure types are sometimes cast to each other's types. This usage is discussed in detail elsewhere.

union special guarantee

Pointer conversions themselves are rarely a cause of faults; it is the subsequent accesses to the referenced storage that is the source of the problems. Pointer conversion can be looked on as a method of accessing

objects using different representations. As such, it is implicitly making use of representation information, something which is already covered by a guideline recommendation. Recognizing that developers do sometimes need to make use of representation information, the following wording is given as a possible deviation to this representation usage guideline recommendation:

?? representation information using

Dev ??

A pointer-to character type may be converted to a pointer-to another type having all of the qualifiers of the original pointed-to type.

Dev ??

A pointer may be converted to a pointer-to character type having all of the qualifiers of the original pointed-to type.

Dev ??

A pointer-to structure type may be converted to a different pointer-to structure type provided they share a common initial sequence.

Example

```
1  extern int *p_i;
2  extern const float *p_f;
3  extern unsigned char *p_uc;
4
5  void f(void)
6  {
7  p_uc = (unsigned char *)p_i;
8  /* ... */
9  p_uc = (unsigned char *)p_uc;
10 p_uc = (unsigned char const *)p_uc;
11 }
```

Table 758.1: Occurrence of implicit conversions that involve pointer types (as a percentage of all implicit conversions that involve pointer types). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
(struct *)	**int**	44.0	**(void *)**	**int**	4.2
(function *)	**int**	18.4	**(unsigned char *)**	**int**	3.4
(char *)	**int**	7.9	(ptr-to *)	**int**	2.0
(const char *)	**int**	6.9	**(int *)**	**int**	1.9
(union *)	**int**	5.5	**(long *)**	**int**	1.1
(other-types *)	other-types *	4.7			

759 If the resulting pointer is not correctly aligned[57] for the pointed-to type, the behavior is undefined.

Commentary

It is the pointed-to object whose alignment is being discussed here. Object types may have different alignment requirements, although the standard guarantees certain alignments. It is guaranteed that converting a pointer value pointer to **void** and back to the original type will produce a result that compares equal to the original value.

alignment

744 pointer converted to pointer to void

What does the term *correctly* mean here? Both C implementations and processors may specify alignment requirements, but the C Standard deals with implementations (which usually take processor requirements into account). It needs to be the implementation's alignment requirements that are considered correct, if they are met. Different processors handle misaligned accesses to storage in different ways. The overhead of checking each access, via a pointer value, before it occurred would be excessive. The Committee traded off performance for undefined behavior.

alignment

C++

5.2.10p7 *Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.*

The unspecified behavior occurs if the pointer is not cast back to its original type, or the relative alignments are stricter.

Source developed using a C++ translator may contain a conversion of a pointer value that makes use of unspecified behavior, but causes undefined behavior when processed by a C translator.

Other Languages

storage layout

Alignment is an issue resulting from processor storage-layout requirements. As such the resulting undefined behavior could almost be said to be language-independent.

Common Implementations

Most implementations treat a pointer-to-pointer conversion as a no-op in the generated machine code; that is, no conversion code is actually generated. The original bit pattern is returned. The effects of alignment usually become apparent later when the object referenced by the converted pointer value is accessed. An incorrectly aligned pointer value can cause a variety of different execution-time behaviors, including raising a signal.

The Unisys A Series[8] uses byte addresses to represent pointer-to character types and word addresses (six bytes per word) to represent pointers to other integer types. Converting a pointer to **char** to a pointer to **int** requires that the pointer value be divided by six. Unless the **char** object pointed to is on a 6-byte address boundary, the converted pointer will no longer point at it.

Coding Guidelines

alignment representation information using ??

Converting pointer type requires developers to consider a representation issue other than that of the pointed-to types alignment. The guideline recommendation dealing with use of representation information is applicable here.

What if a deviation from the guideline recommendation dealing with use of representation information is made? In source code that converts values having pointer types, alignment-related issues are likely to be encountered quickly during program testing. However, pointer conversions that are correctly aligned on one processor may not be correctly aligned for another processor. (This problem is often encountered when porting a program from an Intel x86-based host, few alignment restrictions, to a RISC-based host, which usually has different alignment requirements for the different integer types.) Alignment of types, as well as representation, thus needs to be considered when creating a deviation for conversion of pointer types.

---

pointer converted back to pointer converted via pointer to void 745 compare equal

Otherwise, when converted back again, the result shall compare equal to the original pointer. 760

Commentary

This is a requirement on the implementation. It mirrors the requirement given for pointer to **void**, except that in this case there is an issue associated with the relative alignments of the two pointer types.

C++

5.2.10p7 *Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.*

The C++ Standard does not specify what *original pointer value* means (e.g., it could be interpreted as bit-for-bit equality, or simply that the two values compare equal).

Common Implementations

For most implementations the converted and original values are bit for bit identical, even when the target processor uses a segmented architecture. (On such processors, it is usually only the result of arithmetic operations that need to be checked for being in canonical form.)

pointer segmented architecture

761 When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object.

pointer converted to pointer to character object lowest addressed byte

object contiguous sequence of bytes

Commentary

This is a requirement on the implementation. By specifying an address, the standard makes it possible to predictably walk a pointer over the entire storage space allocated to the object (using **sizeof** to determine how many bytes it contains). The standard could equally have chosen the highest address. But common developer thinking is for addresses to refer to the beginning of an object, which is the lowest address when storage starts at zero and increases. The vast majority of processors have an address space that starts at zero (the INMOS Transputer[5] has a signed address space). Instructions for loading scalars from a given address invariably assume that successive bytes are at greater addresses.

The lowest address of an object is not required to correspond to the least significant byte of that object, if it represents a scalar type.

C90

The C90 Standard does not explicitly specify this requirement.

C++

The result of converting a "pointer to `cv` *T" to a "pointer to* `cv void`*" points to the start of the storage location where the object of type T resides, . . .* 4.10p2

However, the C wording is for pointer-to character type, not pointer to **void**.

A cv-qualified or cv-unqualified (3.9.3) `void*` *shall have the same representation and alignment requirements as a cv-qualified or cv-unqualified* `char*`*.* 3.9.2p4

A pointer to an object can be explicitly converted to a pointer to an object of different type[65]. Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. 5.2.10p7

The C++ Standard does not require the result of the conversion to be a pointer to the lowest addressed byte of the object. However, it is very likely that C++ implementations will meet the C requirement.

Other Languages

Most languages do not get involved in specifying this level of detail.

Coding Guidelines

Needing to know that an address refers to the lowest addressed byte of an object suggests that operations are performed at this level of implementation detail (which violates the guideline recommendation dealing with use of representation information). However, there are situations where the specification of different functionality, in the standard, is interconnected. For instance, the specification of the `memcpy` library function needs to know whether the addresses passed to it points at the lowest or highest byte.

?? representation information using

Dev ??

A program may depend on the lowest address of a byte being returned by the address-of operator (after conversion to pointer-to character type) when the result is either passed as an argument to a standard library function defined to accept such a value, or assigned to another object that is used for that purpose.

---

footnote 55

55) The macro **NULL** is defined in **<stddef.h>** (and other headers) as a null pointer constant; see 7.17. 762

Commentary

It is also defined in the headers **<locale.h>**, **<stdio.h>**, **<stdlib.h>**, **<string.h>**, **<time.h>**, and **<wchar.h>**.

Other Languages

Many languages defined a reserved word to represent the null pointer constant. The word **nil** is often used.

Coding Guidelines

Some existing source code provides its own definition of this macro rather than **#include**ing the appropriate header. Given that the value of the null pointer constant is defined by the standard this usage would appear to be harmless.

null pointer 748 constant

---

footnote 56 pointer/integer consistent mapping

56) The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment. 763

Commentary

Allowing conversions between pointers and integers is of no practical use unless implementations are willing to go along with developers' expectations of being able to convert pointers in some meaningful way. One, formally validated, C90 implementation[6] represents pointers as an index into an array (the array holding the base address of the actual object and the offset within it). In this implementation a pointer-to integer conversion yielded the value of the array index, not the actual address of the object. While presence of this footnote does not prevent such an implementation from conforming to C99, its behavior might limit the number of product sales made.

C++

5.2.10p4 *[Note: it is intended to be unsurprising to those who know the addressing structure of the underlying machine.]*

Is an unsurprising mapping the same as a consistent one? Perhaps an unsurprising mapping is what C does. :-)

Other Languages

While languages do not get involved in this level of detail, their implementations often pander to developer expectations.

Common Implementations

The intended execution environments addressing structure is the one visible to an executing program. The extent to which the underlying hardware addressing is visible to programs varies between target processors. A processor may have a linear address space at the physical level, but at the logical level the addressing structure could be segmented. It is likely to be in vendors' commercial interest for their implementations to support some form of meaningful mapping.

pointer segmented architecture

Some of the issues involved in representing the null point constant in a consistent manner are discussed elsewhere.

null pointer 748 constant

Coding Guidelines

This footnote expresses the assumption that C developers might want to convert between integer and pointer types in some meaningful way. Such usage is making use of representation information and is covered by a guideline recommendation.

?? representation information using

---

764 57) In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

footnote 57

Commentary

Your author knows of no implementations where the concept *correctly aligned* is not transitive.

C90

This observation was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not point out that alignment is a transitive property.

---

765 Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

object point at each bytes of

Commentary

Here the standard is specifying the second of the two requirements needed to allow strictly conforming C code to handle objects as a sequence of bytes, via a pointer-to character type (the other is that the address of individual bytes is unique). The increment value here being `sizeof(unsigned char)`, which is defined to be 1.

The standard specifies elsewhere that the bytes making up an object are contiguously allocated. There is also a guarantee to be able to increment the result one past the end of the object.

byte address unique
sizeof char defined to be 1
object contiguous sequence of bytes
array contiguously allocated set of objects
value copied using unsigned char
pointer one past end of object

C90

The C90 Standard does not explicitly specify this requirement.

C++

The equivalent C++ requirement is only guaranteed to apply to pointer to **`void`** and it is not possible to perform arithmetic on this pointer type. However, in practice C++ implementations are likely to meet this C requirement.

Other Languages

Most languages do not support the concept of converting a pointer-to character type for the purpose of accessing the bytes of the pointed-to object. However, this concept is sometimes used by developers in other languages (they have to make use of particular implementation details that are not given any language standard's blessing).

Coding Guidelines

Pointing at the individual bytes making up an object is one of the steps involved in making use of representation information. The applicable coding guideline discussion is the one dealing with the overall general issue of using representation information complete process, not the individual steps involved.

types representation

---

766 A pointer to a function of one type may be converted to a pointer to a function of another type and back again;

pointer to function converted

Commentary

This is a requirement on the implementation. It mimics the one specified for pointer-to object conversions. The standard is silent on the issue of converting a pointer-to function type to a pointer-to object type. As such the behavior is implicitly undefined.

758 pointer converted to pointer to different object or type

Other Languages

Few languages support pointers to functions. Some languages allow objects to have a function type, they can be assigned references to functions but are not treated as pointers. Those languages that do support some form of indirect function call via objects do not always support casting of function types.

Common Implementations

Most implementations use the same representation for pointers to objects and pointers to functions (the pointer value being the address of the function in the program's address space). The C compiler for the Unisys e-@ction Application Development Solutions[9] returns the value 32 when **sizeof** is applied to a pointer-to function and 8 when it is applied to a pointer-to object.

call function via converted pointer 768

Although pointers to different function types usually have the same representation and alignment requirements, the same cannot be said for their argument-passing conventions. Some implementations represent a pointer-to function type as an index into a table. This approach is sometimes used to provide a form of memory management, in software, when this is not provided by the target host hardware. Accessing functions via an index into a table, rather than jumping directly to their address in memory, provides the hook for a memory manager to check if the machine code for a function needs to be swapped into memory because it has been called (and potentially another function machine code swapped out).

The IAR PICmicro compiler[2] offers four different storage banks that pointers to functions can refer to. Part of the information needed to deduce the address of a function is the storage bank that contains it (the translator obtains this information from the declaration of each pointer type). The value of a pointer-to function may be assigned to another object having a pointer-to function type, but information on the bank it refers to is not part of the value representation. If two pointers to function refer to different storage banks, it is not possible to call a function referred to by one via the other. This compiler also provides a mechanism for specifying which storage bank is to be used to pass the arguments in a call to a particular function.

Coding Guidelines

pointer converted to pointer to void 744

Pointers to functions are usually converted to another pointer type for the same reasons as pointer-to object types are converted to pointer to **void**; the desire for a single function accepting arguments that have different pointer-to function types. Manipulation of functions as pointer values is not commonly seen in code and developers are likely to have had little practice in reading or writing the types involved. The abstract declarator that is the pointer-to function type used in the conversion tends to be more complex than most other types, comprising both the return type and the parameter information. This type also needs to be read from the inside out, not left-to-right.

abstract declarator syntax

Is there a pointer-to function type equivalent to the pointer to **void** type for pointer-to function conversions? The keyword **void** is an explicit indicator that no information is available. In the case of pointer-to function conversions either the return type, or the parameter type, or both could be unknown.

Example

```
1  extern int f_1(int (*)(void));
2  extern int f_2(void (*)(int, char));
3
4  extern int g_1a(int);
5  extern int g_1b(long);
6  extern long g_2a(int, char);
7  extern float g_2b(int, char);
8
9  void h(void)
10 {
11 f_1((int (*)(void))g_1a);
12 f_1((int (*)(void))g_1b);
13
14 f_2((void (*)(int, char))g_2a);
15 f_2((void (*)(int, char))g_2b);
16 }
```

767 the result shall compare equal to the original pointer.

Commentary

This is a requirement on the implementation. This is the only guarantee on the properties of pointer-to function conversion. It mimics that given for pointer-to object conversions.

760 pointer converted back to pointer

Other Languages

Those languages that do support some form of indirect function call, via values held in objects, often support equality comparisons on the values of such objects, even if they do not support conversions on these types.

Example

```
1  extern int f(void);
2
3  void g(void)
4  {
5  if ((int (*)(void))(int (*)(float, long long))f != f)
6     ; /* Complain. */
7  }
```

768 If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined.

call function via converted pointer

Commentary

In practice the representation of pointers to functions is rarely dependent on the type of the function (although it may be different from the representation used for pointers to objects). The practical issue is one of argument-passing. It is common practice for translators to use different argument passing conventions for functions declared with different parameters. (It may depend on the number and types of the parameters, or whether a prototype is visible or not.) For instance, an implementation may pass the first two arguments in registers if they have integer type; but on the stack if they have any other type. The code generated by a translator for the function definition and calls to it are aware of this convention. If a pointer to a function whose first parameter has a structure type (whose first member is an integer) is cast to a pointer to a function whose first parameter is the same integer type (followed by an ellipsis), a call via the converted pointer will not pass the parameters in the form expected by the definition (the first member of the structure will not be passed in a register, but on the stack with all the other members).

C++

The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined. 5.2.10p6

C++ requires the parameter and return types to be the same, while C only requires that these types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

compatible type if

Common Implementations

In the majority of implementations a function call causes a jump to a given address. The execution of the machine instructions following that address will cause accesses to the storage locations or registers, where the argument values are expected to have been assigned to. Thus the common undefined behavior will be to access these locations irrespective of whether anything has been assigned to them (potentially leading to other undefined behaviors).

Coding Guidelines

Conversions involving pointer-to function types occur for a variety of reasons. For instance, use of callback functions, executing machine code that is known to exist at a given storage location (by converting an integer value to pointer-to function), an array of pointer-to function (an indirect call via an array index being deemed more efficient, in time and/or space, than a **switch** statement). While indirect function calls are generally rare, they do occur relatively often in certain kinds of applications (e.g., callbacks in GUI interface code, dispatchers in realtime controllers).

Example

```
1  extern void f(void);
2
3  extern int (*p_f_1)(void);
4  extern int (*p_f_2)(int);
5  extern int (*p_f_3)(char, float);
6  extern int (*p_f_4)();
7
8  int (*p_f_10)(void) = (int (*)(void))f;
9  long (*p_f_11)(int) = (long (*)(int))f;
10 int (*p_f_12)(char, float) = (int (*)(char, float))f;
11 int (*p_f_13)() = (int (*)())f;
12
13 void f(void)
14 {
15 ((int (*)())p_f_1)();
16 ((int (*)(int))p_f_1)(2);
17 ((void (*)(void))p_f_1)();
18 ((int (*)())p_f_2)();
19 ((int (*)(float, char))p_f_3)(1.2, 'q');
20 ((int (*)(void))p_f_4)();
21 }
```

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.18.1.4), simple assignment (6.5.16.1). 769

References

1. H. Bull. *Multics C User's Guide*. Honeywell Bull, Inc, 1987.
2. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
3. IBM. *ILE C/C++ Compiler Reference*. IBM Canada Ltd, Ontario, Canada, sc09-48 16-00 edition, May 2001.
4. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.
5. INMOS Limited. *Transputer Reference Manual*. Prentice–Hall, 1988.
6. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.
7. P. Smith. Implicit pointer conversion involving const and volatile. Technical Report WG21/N0283, JTC1/SC22/WG21 C++ Standards Committee, 1993.
8. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.
9. Unisys Corporation. *C Compiler Programming Reference Manual Volume 1: C Language and Library*. Unisys Corporation, 7831 0422■006, release level 8R1A edition, 2001.