The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.4 Cast operators

cast-expression syntax

1133

cast-expression:
unary-expression
(*type-name* **)** *cast-expression*

Commentary

A *cast-expression* is also a unary operator. Given that the evaluation of a sequence of unary operators always occurs in a right-to-left order, the lower precedence of the cast operator is not significant.

C++

The C++ Standard uses the terminal name *type-id*, not *type-name*.

Other Languages

Some languages parenthesize the *cast-expression* and leave the *type-name* unparenthesized.

Usage

Measurements by Stiff, Chandra, Ball, Kunchithapadam, and Reps[1] of 1.36 MLOC (SPEC95 version of gcc, binutils, production code from a Lucent Technologies product and a few other programs) showed a total of 23,947 casts involving 2,020 unique types. Of these 15,704 involved scalar types (not involving a structure, union, or function pointer) and 447 function pointer types. Of the remaining casts 7,796 (1,276 unique types) involved conversions between pointers to **void**/**char** and pointers to structure (in either direction) and 1,053 (209 unique types) conversions between pointers to structs.

Constraints

cast scalar or void type

Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type. 1134

Commentary

Casting to the **void** type is a method of explicitly showing that the value of the operand is discarded. Casting a value having a structure or union type has no obvious meaning. (Would corresponding member names be assigned to each other? What would happen to those members that did not correspond to a member in the other type?)

C++

There is no such restriction in C++ (which permits the type name to be a class type). However, the C++ Standard contains a requirement that does not exist in C.

5.4p3 *Types shall not be defined in casts.*

A C source file that defines a type within a cast is likely to cause a C++ translator to issue a diagnostic (this usage is rare).

```
1  extern int glob;
2
3  void f(void)
4  {
5  switch ((enum {E1, E2, E3})glob) /* does not affect the conformance status of the program */
6                                   // ill-formed
7     {
8     case E1: glob+=3;
9              break;
10    /* ... */
11    }
12 }
```

Other Languages

Some languages require the cast to have an arithmetic type. Algol 68 permits a cast to any type for which an assignment would be permitted, and nothing else (e.g., if `T var; var := value` is permitted, then `value` can be cast to type T).

Common Implementations

`gcc` supports the casting of scalar types to union types. The scalar type must have the same type as one of the members of the union type. The cast is treated as being equivalent to assigning to the member having that type. This extension removes the need to know the name of the union member.

```
union T {
          int mem_1;
          double mem_2;
         } u;
int x;
double y;

void f(void)
{
u = (union T)x;       /* gcc: equivalent to u.mem_1 = x */
u = (__typeof__(u))y; /* gcc: equivalent to u.mem_2 = y */
}
```

Coding Guidelines

In this discussion a suffixed literal will be treated as an explicit cast of a literal value, while an unsuffixed literal is not treated as such. An explicit cast is usually interpreted as showing that the developer intended the conversion to take place. It is taken as a statement of intent. It is often assumed, by readers of the source, that an explicit cast specifies the final type of the operand. An explicit cast followed by an implicit one is suspicious; it suggests that either the original developer did not fully understand what was occurring or that subsequent changes have modified the intended behavior.

Cg 1134.1 The result of a cast operation shall not be implicitly converted to another type.

Dev 1134.1 If the result of a macro substitution has the form of an expression, that expression may be implicitly converted to another type.

Example

```
#include "stuff.h"

#define MAX_THINGS 333333u
#define T_VAL ((int)x)

extern SOME_INTEGER_TYPE x;

void f(void)
{
long num_1_things = MAX_THINGS;
long num_2_things = (long)MAX_THINGS;
short num_3_things = (short)MAX_THINGS;

long count_1_things = (long)44444u;
short count_2_things = (short)44444u;

long things_1_val = x;
```

```
18  long things_2_val = (long)x;
19  long things_3_val = (long)(int)x;
20  long things_4_val = (long)T_VAL;
21  }
```

Usage

Usage information on implicit conversions is given elsewhere (see Table **??**).

Table 1134.1: Occurrence of the cast operator having particular operand types (as a percentage of all occurrences of this operator). Based on the translated form of this book's benchmark programs.

To Type	From Type	%	To Type	From Type	%
`(` other-types `)`	other-types	40.1	`(char *)`	`const char *`	1.6
`(void *)`	`_int`	18.9	`(union *)`	`void *`	1.5
`(struct *)`	`struct *`	11.2	`(void)`	`long`	1.3
`(struct *)`	`_int`	4.2	`(unsigned long)`	`unsigned long`	1.3
`(char *)`	`char *`	4.0	`(int)`	`int`	1.3
`(char *)`	`struct *`	3.9	`(unsigned int)`	`int`	1.2
`(struct *)`	`void *`	2.8	`(enum)`	int:8 24	1.2
`(unsigned char)`	`int`	1.7	`(char)`	`_int`	1.2
`(struct *)`	`char *`	1.7	`(unsigned long)`	ptr-to *	1.0

pointer conversion constraints

Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast. 1135

Commentary

simple assignment constraints
conditional operator second and third operands

The special cases listed in 6.5.16.1 allow some values that are assigned to have their types implicitly converted to that of the object being assigned to. There are also contexts (e.g., the conditional operator) where implicit conversions occur.

C90

This wording appeared in the Semantics clause in the C90 Standard; it was moved to constraints in C99. This is not a difference if it is redundant.

C++

The C++ Standard words its specification in terms of assignment:

4p3 *An expression* `e` *can be implicitly converted to a type* `T` *if and only if the declaration "*`T t=e;`*" is well-formed, for some invented temporary variable* `t` *(8.5).*

Semantics

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. 1136

Commentary

simple assignment constraints
default argument promotions
pointer conversion 1135 constraints
_Bool converted to

Most of the implicit conversions that occur in C cause types to be widened. In the case of assignment and passing arguments narrowing may occur. Explicit casts are required to support conversions involving pointers. The use of the phrase *named type* is misleading in that the type converted to may be anonymous. For most pairs of types, but not all, the cast operator is commutative (assuming the operand is within the defined range of the types).

Although C has a relatively large number of integer types (compared to the single one supported in most languages), the language has been designed so that explicit casts are rarely required for these constructs.

For instance, an explicit cast is required to convert an array index having a floating type or to convert an argument having type **long** to **int** when using old style function declarations. Typedef names are synonyms and do not require the use of explicit casts seen in more strongly typed languages.

typedef
is synonym

Common Implementations

The most common implementation of pointer and widening integer conversions is to treat the existing value bits as having the new type (in many cases values having integer type will already be represented in a processor's register using the full number of bits).

In the case of conversions to narrower integer types, the generated machine code may depend on what operation performed on the result of the cast. In the case of assignment the appropriate least-significant bits of the unconverted operand are usually stored. For other operators implementations often zero out the nonsignificant bits (performing a bitwise-AND operation is often faster than a remainder operation) and for signed types sign extend the result.

The representation difference between integer and floating-point types is so great that many processors contain instructions that perform the conversion. In other cases internal calls to library functions have to be generated by the translator.

Coding Guidelines

Casts are sometimes used as a means of reducing a value so that it falls within a particular range of values (i.e., effectively performing a modulo operation). This usage may be driven by the desire not to use two other possible operators: (1) bitwise operators, because of the self-evident use of representation information; and (2) the remainder operator, because it is viewed as having poor performance. A cast operator used for this purpose is making use of representation information; the range of values that happen to be supported by the cast type on the implementation used. This usage is a violation of the guideline recommendation dealing with the use of representation information.

% operator
result

?? representation information
using
?? object
int type only

Following the guideline recommendation specifying the use of a single integer type removes the need to consider many of the conversion issues that can otherwise arise with these types.

In some contexts an explicit conversion is required, while in other contexts a translator will perform an implicit conversion. Are there any benefits to using an explicit cast operator in these other contexts? An explicit cast can be thought about in several ways:

operand
convert automatically

- An indicator of awareness on the part of the codes author. The assumption is often made by subsequent readers of the source that an explicit cast shows that a conversion was expected/intended. Static analysis tools tend to treat implicit conversions with some suspicion.
- A change of interpretation of a numeric value (e.g., signed/unsigned or integer/floating-point conversions).
- A method of removing excess accuracy in floating-point operations.

FLT_EVAL_METH

- A method of changing the conversions that would have occurred as a result of the usual arithmetic conversions.

usual arithmetic conversions

- A method of explicitly indicating that a change in role, or symbolic name status, of an object is intended.

object
role
symbolic name

The cost/benefit issues and possible guideline recommendations are discussed elsewhere.

operand
convert automatically

1137 This construction is called a *cast*.[86)]

cast

Commentary

This defines the term *cast*. Developers often call this construction an *explicit cast*, while an implicit conversion performed by a translator is often called an *implicit cast*.

C++

The C++ Standard uses the phrase *cast notation*. There are other ways of expressing a type conversion in C++ (functional notation, or a type conversion operator). The word *cast* could be said to apply in common usage to any of these forms (when technically it refers to none of them).

Other Languages

The terms *cast* or *coercion* are often used in programming language definitions.

Coding Guidelines

The term *cast* is often used by developers to refer to the implicit conversions performed by an implementation. This is incorrect use of terminology, but there is very little to be gained in attempting to change existing, common usage developer terminology.

A cast that specifies no conversion has no effect on the type or value of an expression.[87)] 1138

Commentary

The standard does not define what is meant by *no conversion*; only one kind of cast can have no effect on the semantic type (i.e., a cast that has the same type as the expression it operates on). If an implementation uses the same representation for two types, any cast between those two types will have no effect on the value. A cast to **void** specifies no conversion in the sense that the value is discarded. As footnote 87 points out, it is possible to cast an operand to the type it already has and change its value.

footnote 87 1142

The footnote was moved to normative text by the response to DR #318.

C++

The C++ Standard explicitly permits an expression to be cast to its own type (5.2.11p1), but does not list any exceptions for such an operation.

Common Implementations

The special case of a cast that specifies no conversion is likely to be subsumed into an implementation's general handling of machine code generation for cast operations.

Coding Guidelines

redundant code

Technically, a cast that specifies no conversion is a redundant operation. A cast that has no effect can occur for a number of reasons:

- When typedef names are used; for instance, converting a value having type `TYPE_A` to `TYPE_B`, when both typedef names are defined to have the type **int**.
- When an object is defined to have different types in different arms of a **#if** preprocessing directive. Casts of such an object may be redundant when one arm is selected, but not the other.
- When an argument to a macro invocation is cast to its declared type in the body of the macro.
- When an invocation of a macro is cast to the type of the expanded macro body.
- Developer incompetence, or changes to existing code, such that uses no longer fit in the above categories.

Some redundant casts may unnecessarily increase the cost of comprehending source code. (Their existence in source code will require effort to process; a redundant cast may generate additional effort because it is surprising and the reader may invest additional effort investigating it.) However, in practice many redundant casts exist for a good reason. But, providing a definition for an *unnecessary* redundant cast is likely to be a complex task. A guideline recommending against some form of redundant casts does not appear to be worthwhile.

While a cast operation may not specify any conversion based on the requirements contained in the C Standard, it may specify a conceptual conversion based on the representation of the application domain (i.e.,

the types of the cast and its operand are specified using different typedef names which happen to use the same underlying type). Checking that the rules behind these conceptual conversions are followed requires support from a static analysis tool (at the translator level these conversions appear to be redundant code). redundant code

Example

```
1   typedef int TYPE_A;
2   typedef int TYPE_B;
3
4   extern signed int i_1;
5   extern TYPE_A t_1;
6
7   #include "stuff.h"
8
9   void f(void)
10  {
11  int loc_1 = i_1;
12  int loc_2 = (int)i_1;
13  int loc_3 = (TYPE_B)i_1;
14  int loc_4 = t_1;
15  int loc_5 = COMPLEX_EXPR;
16  int loc_6 = (int)COMPLEX_EXPR;
17  int loc_7 = (int)loc_1;
18
19  TYPE_B toc_1 = t_1;
20  TYPE_B toc_2 = (int)t_1;
21  TYPE_B toc_3 = (TYPE_B)t_1;
22  TYPE_B toc_4 = i_1;
23  TYPE_B toc_5 = COMPLEX_EXPR;
24  TYPE_B toc_6 = (TYPE_B)COMPLEX_EXPR;
25  TYPE_B toc_7 = (TYPE_B)toc_1;
26  }
```

1139 **Forward references:** equality operators (6.5.9), function declarators (including prototypes) (6.7.5.3), simple assignment (6.5.16.1), type names (6.7.6).

1140 86) A cast does not yield an lvalue. footnote 86

Commentary

The idea behind the cast operator is to convert values, not the types of objects. A cast of a value having a point type may not be an lvalue, but the result can be dereferenced to yield an lvalue.

```
1   extern int *p_i;
2
3   void f(void)
4   {
5   (int)*p_i = 3;   /* Constraint violation, left operand not an lvalue.      */
6   *(int *)p_i = 3; /* Does not affect the conformance status of the program. */
7   }
```

C++

The result is an lvalue if T is a reference type, otherwise the result is an rvalue. 5.4p1

Reference types are not available in C, so this specification is not a difference in behavior for a conforming C program.

Common Implementations

Some implementations (e.g., `gcc`) allow casts to yield an lvalue.

```
1  int loc;
2
3  (char)loc = 0x02; /* Set one of the bytes of an object to 0x02. */
```

Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type. 1141

Commentary

qualifier meaningful for lvalues
pointer to qualified/unqualified types

Type qualifiers affect how translators treat objects, not values. The standard specifies some requirements on pointers to unqualified/qualified versions of types.

C++

Casts that involve qualified types can be a lot more complex in C++ (5.2.11). There is a specific C++ cast notation for dealing with this form of type conversion, `const_cast<T>` (where `T` is some type).

5.2.11p12 *[Note: some conversions which involve only changes in cv-qualification cannot be done using* ***`const_cast`****. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior.*

The other forms of conversions involve types not available in C.

Coding Guidelines

Casting to a qualified type can occur through the use of typedef names. Explicitly specifying a type qualifier in the visible source is sufficiently rare that it does not warrant a guideline (while it may not affect the behavior of a translator, the developer's beliefs about such a cast are uncertain).

---

footnote 87

~~87)~~ If the value of the expression is represented with greater precision or range than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type. 1142

Commentary

float promoted to double or long double
FLT_EVAL_METHOD

This footnote clarifies that the permission to perform operations with greater precision (values having floating-point types may be represented to greater precision than is implied by the type of the expression) does not apply to the cast operator. The issues surrounding this usage are discussed elsewhere.

The footnote was moved to normative text by the response to DR #318.

C++

The C++ Standard is silent on this subject.

Common Implementations

The representation used during the evaluation of operands having floating-point type is usually dictated by the architecture of the processor. Processors that operate on a single representation often have instructions for converting to other representations (which can be used to implement the cast operation). An alternative implementation technique, in those cases where no conversion instruction is available, is for the implementation to specify all floating-point types as having the same representation.

References

1. M. Stiff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. Technical Report Technical Report BL0113590-990202-03, Bell Laboratories, 1999.