PRQA Programming Research

QA·C 8.1 SOURCE CODE ANALYZER ESSENTIAL TYPE

USER GUIDE

July 2012

This document describes the rationale of Essential Type and the specification of the messages that QA·C produces to enforce it.

IMPORTANT NOTICE

DISCLAIMER OF WARRANTY

The staff of Programming Research Ltd have taken due care in preparing this document which is believed to be accurate at the time of printing. However, no liability can be accepted for errors or omissions nor should this document be considered as an expressed or implied warranty that the products described perform as specified within.

COPYRIGHT NOTICE

TRADEMARKS

PRQA, the PRQA logo, and QA·C are registered trademarks of Programming Research Ltd. Windows is a registered trademark of Microsoft Corporation.

CONTACTING PROGRAMMING RESEARCH LTD

For technical support, contact your nearest Programming Research Ltd authorized distributor or you can contact Programming Research's head office:

by telephone on +44 (0) 1 932 888 080
by fax on +44 (0) 1 932 888 081
or by e-mail on support@programmingresearch.com
www.programmingresearch.com

Table of Contents

1. INTRODUCTION

Version 8.1 of QA·C introduces an extensive range of new messages associated with type usage and type conversion. These messages are based on the concept of essential type as described in this user guide.

1.1 MISRA-C

In the MISRA-C:2004 Guidelines, several terms were introduced in an attempt to describe the essential nature of an arithmetic expression in a way that the ISO-C language standard does not provide. These terms were:

- *Underlying type*
- *Effectively Boolean*
- *Complex expression*

Although these concepts have been found useful, a number of weaknesses have been identified:

a) The term *underlying type* is used already in a different sense in the C++ standard to describe the implementation of enum types.

b) The term *complex* is a source of confusion, being more obviously associated with the algebra of *complex numbers* – and hence complex floating types within the C language.

c) The definition of *underlying type* was not specified with sufficient rigour.

d) The concept of *underlying type* did not address the whole range of integer data types – such as character data, enum data, Boolean data and bit-fields.

1.2 Essential Type

The *essential type* concept has evolved from the concept of *underlying type* with the intention of addressing some of the problems referred to above. It provides an alternative way of describing the essential characteristics of objects, constants and expressions of *arithmetic type*.

The term *essentially Boolean* now supersedes the term *effectively Boolean*.

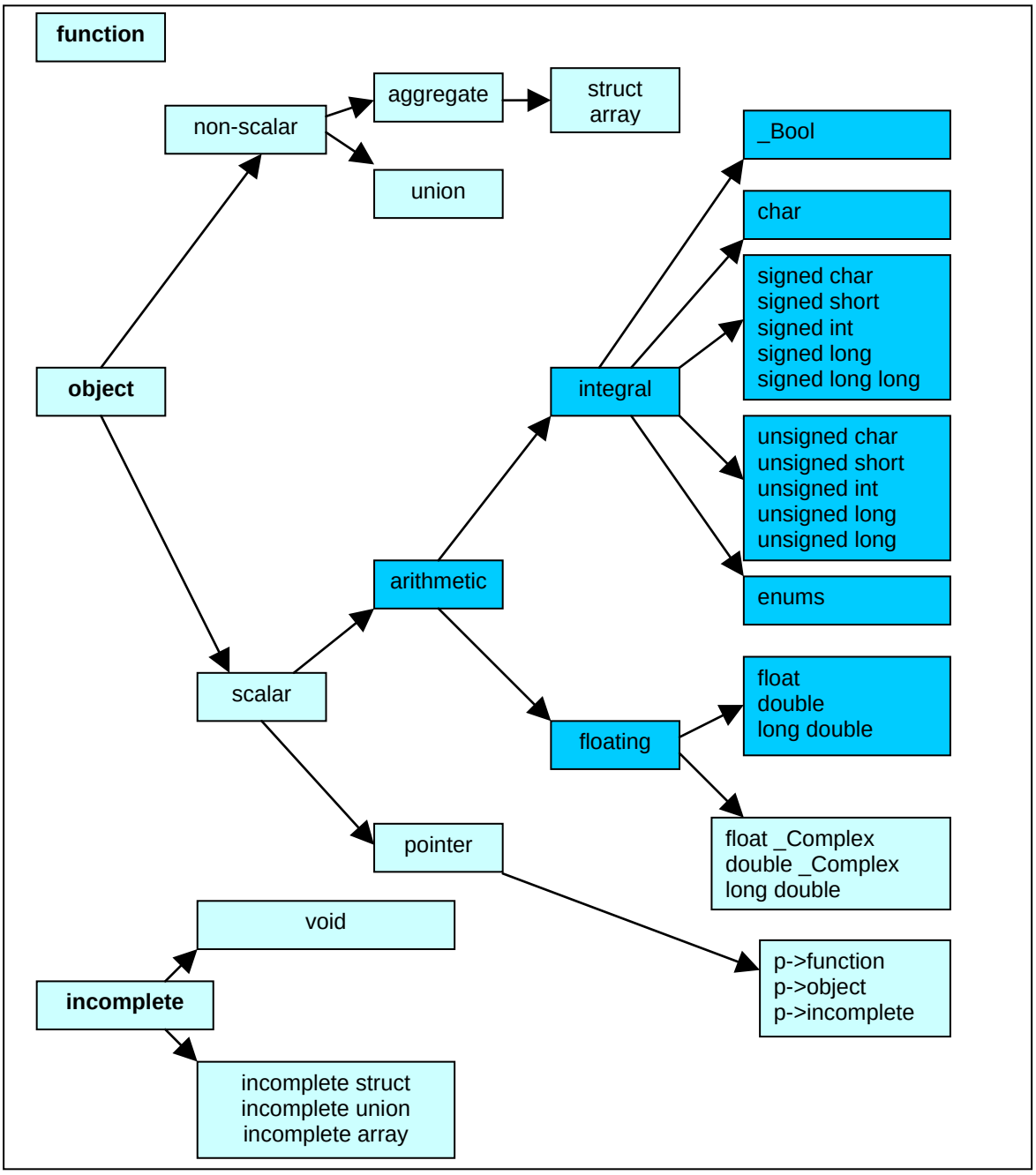
1.3 Composite Expression

The term *composite expression* now supersedes the term *complex expression*.

2. THE STANDARD TYPE SYSTEM

2.1 ISO:C99 Types

The ISO:C99 language standard defines a range of types as depicted in the diagram shown below. The discussion of essential type addressed in this document provides an alternative way of describing any expression of arithmetic type.



Arithmetic types and pointer types are two sub-classes of the set of "scalar" types. Every object, literal constant and expression of arithmetic type has a type defined by the semantics of the C language. We will refer to this type as the **Standard Type (ST)** to distinguish it from **Essential Type (ET)** – which will be described later.

2.1.1 Terminology

The ISO-C99 document defines the following terms in section 6.2.5

- arithmetic types: the integer and floating types.
- integer types: char, the signed/unsigned integer types, and enum types
- real floating types: float, double and long double
- real types: the integer and real floating types.
- standard signed integer types: signed char/short/int/long/long long
- standard unsigned integer types: unsigned char/short/int/long/long long and type _Bool
- basic types: char, the signed/unsigned integer types, and float types

2.1.2 Rank

The family of standard signed and unsigned types and type *char* form a hierarchy which is described by the term “rank”.

Signed and unsigned *long long* share the highest rank. *_Bool* has the lowest rank. The three character types *signed char, unsigned char* and “plain” *char* are all of the same rank.

In the ISO-C standard, "rank" is a term applied only to integral types. In this discussion we shall extend the concept to floating types thus:

- No two floating types shall have the same rank, even if they have the same representation.
- The rank of *long double* shall be greater than the rank of *double*, which shall be greater than the rank of *float*.

2.2 Anomalies

The C language as defined in the ISO-C standard contains a number of weaknesses and inconsistencies in its definition of arithmetic types. The effect of this is that the standard type of an expression is not always fully descriptive of the essential nature of the data being represented. For example:

2.2.1 Integral Promotion

The set of integer types do not behave in a consistent way. As a result of integral promotion, an expression which is intrinsically unsigned (e.g. "*unsigned char* + *unsigned char*") typically has a standard type which is *signed int*.

2.2.2 Integer Constants

Integer constants only exist for the larger numeric types, i.e. [*signed*|*unsigned*] *int*, *long* and *long long* types.

There are no constants of type [*signed*|*unsigned*] *short* or [*signed*|*unsigned*] *char* (although it is possible to define a *constant expression* of these types by using a cast).

2.2.3 Character Constants

The type of a character constant (e.g. 'x') is defined to be *int* (not *char*).

2.2.4 Logical Expressions

A Boolean type does not exist in the C90 language. Type *_Bool* was introduced into C99, but unfortunately, for reasons of backwards compatibility, the result of an equality (==, !=), relational (<, <=, >=, >) or logical (&&, ||, !) operator is still of type *signed int* rather than type *_Bool*.

2.2.5 Bit-fields

There is no such thing as a *bit-field* type. Bit-fields are permitted by the language standard to be defined with type *int, signed int, unsigned int* (and also *_Bool* in C99). This means, for example, that it is not possible to cast to a bit-field type.

2.2.6 enum Variables

An enum variable has a type which is “implementation-defined”. The implementation is free to use any integer type which is capable of representing the enumeration. If the enumeration includes negative values, the enum type will be a signed integer type. If the enumeration consists only of non-negative values, the enum type may be either a signed or unsigned integer type.

2.2.7 enum Constants

Regardless of the type used to implement an enum variable, an enum constant is always of type *signed int*.

2.3 Diagnostic Requirements

The C language imposes a very minimal set of constraints on the type of an expression which may be used in any particular context. It is therefore quite possible to write perfectly legal code which, because of the *type* of an expression, is either of questionable meaning or simply unsafe.

A few examples:

2.3.1 Operands of an Unlikely Type

- Using an expression of Boolean type as an operand of a relational operator.
- Using an expression of plain char type as an operand of a shift operator.

2.3.2 Unexpected Operand Combinations

- Subtracting an expression of plain char type from an expression of signed or unsigned type.
- Comparing an expression of plain char type with an expression of floating type.

2.3.3 Explicit Type Conversions

- Casting an expression of Boolean type to a floating type.

2.3.4 Implicit Type Conversions which may Result in Loss of Precision or Value

- Assigning an expression of floating type to an integer object.
- Operations which result in the conversion of a signed expression (which could be negative) to an unsigned type.

2.3.5 Implicit Type Conversions which are 'questionable'

- Assigning a non-enum value to an enum variable.
- Arithmetic expressions which mix signed/unsigned/character/Boolean/enum types in a way which violates strong typing principles.

A fundamental problem encountered in identifying coding problems such as these is that, quite frequently, the type of an expression as defined in the C language is inadequate or misleading in describing the *essential* characteristics of the data which it represents. For example, it frequently happens that, because of *integral promotion,* an expression which is intuitively and intrinsically *unsigned* is technically of type *signed int* according to the ISO C language standard. This is just one of the anomalies in the C language described in the previous section which can give rise to much confusion.

Consider an example:

```
extern void foo(int si, unsigned char uca, unsigned char ucb)
{
  unsigned int ui;

  ui = si;            /* A */
  ui = uca + ucb;     /* B */
  ...
}
```

Statement A: If the variable ***si*** contains a negative value, the result of the assignment in statement A will be 'implementation defined' - and probably meaningless.

A static analysis tool will usually generate a diagnostic located on the assignment operation to point out the

dangerous type conversion from signed int to unsigned int. Implicit conversions from a signed type to an unsigned type are, generally, dangerous.

Statement B: The implicit type conversion which takes place in the assignment operation is no different from that in statement A because the type of the expression which results from adding two objects of type *unsigned char* is also *signed int*.

However, most developers would consider statement B to be fundamentally different: the expression 'uca + ucb' is *'essentially unsigned'* - it cannot have a negative value. However, in practice, the great majority of developers are quite unaware of integral promotion and a diagnostic message which reported a "signed to unsigned" conversion would come as an unwelcome surprise.

For this reason, a coding rule which simply bans "signed to unsigned" conversions is far too simplistic and a static analysis tool which does not distinguish between "Statement A" and "Statement B" will inevitably generate messages which are unhelpful and distracting.

2.3.6 Essential Type

In order to address subtle distinctions like those described above, the concept of essential type has been developed as a way of describing more faithfully the essential nature of the data which is being represented in an arithmetic expression. The essential type system provides a way of defining coding rules which are both intuitively sensible and unambiguous.

3. ESSENTIAL TYPE OVERVIEW

3.1 Essential Type Category

The primary function of essential type (ET) is to provide an alternative type system which provides a more rational way of describing the nature of any arithmetic expression.

To this end it is helpful to be able to refer to an arithmetic expression as having an **essential type category**. The expression may then be described under one of the following headings:

- essentially Boolean
- essentially character
- essentially enum
- essentially signed
- essentially unsigned
- essentially floating

These 6 terms are used to distinguish the essential nature of the data represented by any expression of arithmetic type.

3.2 Essential Type

The **essential type** of an expression is denoted by one of the usual types:

Essential Type Category	Essential Types
Boolean	*_Bool*
character	*char*
enum<i>	*enum*
signed	*signed* [*char*|*short*|*int*|*long*|*long long*]
unsigned	*unsigned* [*char*|*short*|*int*|*long*|*long long*]
floating	*float, double or long double*

Notice the following:

- *char* is only used to describe data which is *essentially character* in nature.
- *signed char* is only used to represent data which is *essentially signed* in nature.
- *unsigned char* is only used to represent data which is *essentially unsigned* in nature.
- Each enumeration is considered a unique type with a unique *essential type category* – referred to in this document using the notation *enum<i>* where i simply represents a unique reference by which the enumeration is distinguished.

3.3 Essential Type Principles

ET is a property of any expression having arithmetic type

The ET of an expression is often but not always the same as the standard type (ST).

The ET and ST are only ever different where ST is either *signed int* or *unsigned int*.

Where the ST of an expression is *signed int*, the ET may be either *unsigned char, unsigned short, signed char, signed short, signed int, char, _Bool* or *enum<i>*.

Where the ST of a non-constant expression is *unsigned int*, the ET is either *unsigned int* or another *unsigned integer type* which is implemented in the same size as an int.

Where the ST of a constant expression is *unsigned int*, the ET may be either *unsigned int, unsigned short* or *unsigned char*.

4. ESSENTIAL TYPE DEFINITIONS

As noted above, the ET of many expressions is exactly the same as the ST. In the following paragraphs we refer only to the specific situations where ET and ST are different.

4.1 STLR and UTLR

The ET of a constant expression which has an ST of *signed int* and is *essentially signed* is defined by determining the *signed type of lowest rank* which is capable of representing the actual value. This is referred to as the STLR rule.

The UTLR rule (*unsigned type of lowest rank*) is similar but applies to constant expressions which have an ST of *unsigned int* and are *essentially unsigned*.

4.2 The Essential Type of a Bit-field

- BF1: The ET of a bit-field which is implemented with an unsigned type is the unsigned type of lowest rank which is able to represent the bit-field.
- BF2: The ET of a bit-field which is implemented with a signed type is the signed type of lowest rank which is able to represent the bit-field.

4.3 The Essential Type of a Literal Constant

The ISO-C99 standard defines the following constants of "integral type":

- Integer constants
- Enumeration constants
- Character constants

Note that enumeration constants and character constants are of "integral type" but are not "integer constants".

4.3.1 Integer Constants

- I1: If the ST of an integer constant is “signed int”, the ET is the STLR.
- I2: If the ST of an integer constant is “unsigned int”, the ET is the UTLR.

4.3.2 Character Constants

The ST of a character constant (e.g. ‘q’) is *int* – not type *char*. (Note that in C++ character constants are of type *char*).

- C1: The ET of a character constant consisting of a single character is "char".

(However the ET of a multiple character constant ('xy') is the same as its ST i.e. "int".)

4.3.3 Boolean Constants

The C99 standard provides no syntax to define a constant of explicitly _Bool type. The library header “stdbool.h” defines “false” and “true” only in terms of constants 0 and 1 of type int. The only way to define a constant expression of type _Bool explicitly is to use a cast, e.g. (_Bool)0 or (_Bool)1.

4.3.4 enum Constants

The ST of an enum constant in C is always int, regardless of the implemented type of the enumeration and regardless of the type of any initializer expression used to define its value. For example, if an enum constant is initialised with an unsigned value, say “500U”, the constant will still be considered to have a ST of signed int.

Two distinct types of enumeration need to be considered:

i) An enumeration which defines an *enumeration type*. An enumeration defined in this way will be identified either by a tag or a typedef or by being used directly in the definition of an object or function. For example:

```
enum ETAG {A, B, C};
typedef enum {A, B, C} ETYPE;
typedef enum ETAG {A, B, C} ETYPE;
enum {A, B, C} x;
```

ii) An enumeration which is used to define a set of constants (which may or may not be related). This will be referred to as an *anonymous enumeration.* An enumeration defined in this way does not define a type because it is not associated with any tag, typedef, object or function. For example:

```
enum {A = 10, B = 20, C = 30};
```

- E1: If an enumeration defines an enumeration type, the ET of its enumeration constants is enum<i>.
- E2: If an enumeration is anonymous, the ET of its enumeration constants is the STLR.

4.4 The Essential Type of an Arithmetic Expression

4.4.1 Relational (< <= >= >), Equality (== !=) and Logical (&& || !)

- LOGIC: The ET is _Bool

4.4.2 Shift (<< >>)

- SHIFT1: If the LH operand is essentially signed or unsigned, the ET of the result is the ET of the LH operand.

4.4.3 Bitwise Complement (~)

- COMP1: If the operand is essentially unsigned, the ET of the result is the ET of the operand.

4.4.4 Unary Plus (+)

- PLUS: If the operand is essentially signed or unsigned, the ET of the result is the same.

4.4.5 Unary Minus (-)

- MINUS1: If the operand is essentially signed and is a constant expression, the ET of the result is the STLR.
- MINUS2: If the operand is essentially signed and the expression is non-constant, the ET of the result is the ET of the operand.

4.4.6 Conditional (? :)

- COND1: If the ET of the 2nd and 3rd operands is the same, the ET of the result is the same.
- COND2: If the 2nd and 3rd operands are both essentially unsigned, the ET of the result is the ET of highest rank.
- COND3: If the 2nd and 3rd operands are both essentially signed, the ET of the result is the ET of highest rank.
- COND4: If one operand is an expression of essentially unsigned type and the other is a non-negative constant expression of essentially signed type with an essential rank which is no greater than the essential rank of the unsigned operand, the ET of the result is the ET of the unsigned operand.

4.4.7 Type Balancing Operations (* / % + - & | ^)

- BAL1: If both operands are essentially unsigned and both are constant expressions, the ET of the result is the UTLR.
- BAL2: If both operands are essentially unsigned and either is non-constant, the ET of the result is the ET of highest rank.
- BAL3: If both operands are essentially signed and both are constant expressions, the ET of the result is the STLR.
- BAL4: If both operands are essentially signed and either is non-constant, the ET of the result is the ET of highest rank.
- BAL5: If one operand is essentially unsigned and the other is a non-negative constant expression of essentially signed type with an essential type rank which is no greater than the essential type rank of the unsigned operand, the ET of the result is the ET of the unsigned operand.

4.4.8 Multiply

- MUL1 If one operand is essentially signed or essentially unsigned and the other is essentially Boolean, the ET of the result is ET of the non-Boolean operand.

4.4.9 Addition

- ADD1 If one operand is essentially character and the other is essentially signed or unsigned with an essential type which is no greater than int, the ET of the result is char.

4.4.10 Subtraction

- SUB1 If the first operand is essentially character and the second is essentially signed or unsigned with an essential type which is no greater than int, the ET of the result is char.
- SUB2 If both operands are essentially character constant expressions, the ET of the result is the STLR.

4.4.11 Comma Operator

- COMMA: The ET is the ET of the right hand operand.

5. COMPOSITE EXPRESSIONS

5.1 Terminology

The term *composite expression* supersedes the term *complex expression* which was introduced in the MISRA-C:2004 Guidelines document. Although the concepts remain similar, the definitions of the two terms are not identical.

The concept of a *composite expression* is used in order to define coding rules which consider it inadvisable for consecutive arithmetic operations to be conducted in different types. For example:

```
ula = uca + ucb;           /* Example 1 */
ula = uca + ucb + ulc;     /* Example 2 */
```

In **Example 1**, an addition operation is followed by an assignment operation. The addition operation occurs (notionally) in the essential type *unsigned char*; the result of the addition then undergoes an implicit widening conversion to *unsigned long* for the purpose of assignment.

In **Example 2**, two operands of type *unsigned char* (uca and ucb) are added and then the result is implicitly widened to type *unsigned long* for addition to ulc.

In both cases, the subexpression “uca +ucb” is considered a *composite expression*. In Example 1, the expression is widened when assigned to “ula”. In Example 2, the expression is widened when added to to “ulc”.

5.2 Composite Operators

The following operators are described as *composite operators*:

- Multiplicative * / %
- Additive + -
- Bitwise & | ^
- Shift << >>
- Conditional ? :

However, the conditional operator is only considered *composite* if either the 2nd or the 3rd operand is a *composite expression*.

5.3 Composite Expressions

An expression is deemed to be “*composite*” if it is *non-constant* and its value is the result of a *composite operator*.

6. ESSENTIAL TYPE MESSAGES

6.1 Overview

QA·C generates a large number of messages to diagnose a wide range of conditions in source code. The specification of a particular message is dictated in many instances by the need to support flexibility in meeting the requirements of differing coding standards.

Within the complete library of QA·C messages, there are many messages associated with 'type' usage which may be used to identify:

- The type of an individual operand being used with a particular operator
- The combination of particular types of operand in binary and ternary operations
- **Explicit** type conversions which occur in a cast operation
- **Implicit** type conversions which occur in a variety of contexts

The following sections of this document provide an overview of these messages. A key aspect throughout is that the messages usually refer to the "**essential type**" and "**essential type category**" of an expression rather than the "**standard type**" defined in the ISO C language standard. As discussed previously, in many situations the *essential type* and *standard type* of an expression are identical; but in some contexts, the concept of *essential type* is useful in capturing the essential nature of an expression in a way that is obscured by the *standard* C type system.

6.2 The -strictrank option

When comparing types, some of the essential type messages use the terms "**wider**" and "**narrower**" when describing the relationship between types. By default, the terms *wider* and *narrower* reflect the relationship between the implemented sizes of the two types; therefore type *long* will only be *wider* than type *int* if the types are implemented in a different number of bits.

If the **strictrank** option is applied, the behaviour of such messages is altered and the terms *wider* and *narrower* no longer reflect the relative implemented size but the relative rank of the two types. Therefore, if the strictrank option is applied, type *long* is always considered to be *wider* than type *int*, even if both are implemented in the same number of bits.

The facility to apply this option provides a means of exposing situations where issues of portability may be significant.

6.3 Expression Type Usage Messages

The C language supports a wide range of operators but imposes very few restrictions on the way in which they may be used. There are many situations where an expression of a particular type might be considered an inappropriate operand for a specific operator. For example:

- Performing a bitwise operation (&, |, ^ , ~, <<, >>) on *essentially signed* data
- Performing a logical operation on *essentially floating* data
- Performing a relational operation (< <= >= >) on *essentially Boolean* data

The table below presents 9 columns which distinguish various kinds of expression, i.e. its essential type and whether or not it is a constant expression. Each row in the table corresponds to an operator and one of its operands (i.e. operand 1, 2 or 3). The cells in the table show the message number which may be used to highlight situations when the essential type of the operand may be considered inappropriate to the context.

		ESSENTIAL TYPE CATEGORY OF OPERAND								
Operator	Operand	Boolean	character	enum non-const expression	enum const expression	signed non-const expression	negative signed constant expression	non-neg signed constant expression	unsigned	floating
[ ]		4500	4510							constraint
Unary +	1	4501	4511	4521	4521					
Unary -	1	4501	4511	4521	4521					
+	1	4501		4521	4521					
+	2	4501		4521	4521					
-	1	4501		4521	4521					
-	2	4501		4521	4521					
*	1	4501	4511	4521	4521					
*	2	4501	4511	4521	4521					
/	1	4501	4511	4521	4521					
/	2	4501	4511	4521	4521					
%	1	4501	4511	4521	4521					constraint
%	2	4501	4511	4521	4521					constraint
~	1	4502	4512	4522	4522	4532	4532	4542		constraint
& | ^	1	4502	4512	4522	4522	4532	4532	4542		constraint
& | ^	2	4502	4512	4522	4522	4532	4532	4542		constraint
<< >>	1	4503	4513	4523	4523	4533	4533	4543		constraint
<< >>	2	4504	4514	4524	4524	4534	4534	4544		constraint
< <= >= >	1	4505								
< <= >= >	2	4505								
== !=	either									
++ --	1	4507	4517	4527	constraint		constraint	constraint		
!	1		4518	4528	4528	4538	4538	4548	4558	4568
&& ||	1		4518	4528	4528	4538	4538	4548	4558	4568
&& ||	2		4518	4528	4528	4538	4538	4548	4558	4568
? :	1		4519	4529	4529	4539	4539	4549	4559	4569
? :	2 and 3									

ETC of operand	Message No.	Message Text
Boolean	4500	An expression of 'essentially Boolean' type (%1s) is being used as an array subscript.
	4501	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
	4502	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
	4503	An expression of 'essentially Boolean' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
	4504	An expression of 'essentially Boolean' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
	4505	An expression of 'essentially Boolean' type (%1s) is being used as the %2s operand of this relational operator (%3s).
	4507	An expression of 'essentially Boolean' type (%1s) is being used as the operand of this increment/decrement operator (%2s).
character	4510	An expression of 'essentially character' type (%1s) is being used as an array subscript.
	4511	An expression of 'essentially character' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
	4512	An expression of 'essentially character' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
	4513	An expression of 'essentially character' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
	4514	An expression of 'essentially character' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
	4517	An expression of 'essentially character' type (%1s) is being used as the operand of this increment/decrement operator (%2s).
	4518	An expression of 'essentially character' type (%1s) is being used as the %2s operand of this logical operator (%3s).
	4519	An expression of 'essentially character' type (%1s) is being used as the first operand of this conditional operator (%2s).
enum	4521	An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this arithmetic operator (%3s).
	4522	An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
	4523	An expression of 'essentially enum' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
	4524	An expression of 'essentially enum' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
	4527	An expression of 'essentially enum' type is being used as the operand of this increment/decrement operator.
	4528	An expression of 'essentially enum' type (%1s) is being used as the %2s operand of this logical operator (%3s).
	4529	An expression of 'essentially enum' type (%1s) is being used as the first operand of this conditional operator (%2s).
signed	4532	An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
	4533	An expression of 'essentially signed' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
	4534	An expression of 'essentially signed' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
	4538	An expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).
	4539	An expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
	4542	A non-negative constant expression of 'essentially signed' type (%1s) is being used as the %2s operand of this bitwise operator (%3s).
	4543	A non-negative constant expression of 'essentially signed' type (%1s) is being used as the left-hand operand of this shift operator (%2s).
	4544	A non-negative constant expression of 'essentially signed' type (%1s) is being used as the right-hand operand of this shift operator (%2s).
	4548	A non-negative constant expression of 'essentially signed' type (%1s) is being used as the %2s operand of this logical operator (%3s).

ETC of operand	Message No.	Message Text
	4549	A non-negative constant expression of 'essentially signed' type (%1s) is being used as the first operand of this conditional operator (%2s).
unsigned	4558	An expression of 'essentially unsigned' type (%1s) is being used as the %2s operand of this logical operator (%3s).
	4559	An expression of 'essentially unsigned' type (%1s) is being used as the first operand of this conditional operator (%2s).
floating	4568	An expression of 'essentially floating' type (%1s) is being used as the %2s operand of this logical operator (%3s).
	4569	An expression of 'essentially floating' type (%1s) is being used as the first operand of this conditional operator (%2s).

6.4 Assigning Type Conversion Messages

The term "*assigning type conversion*" is used to describe the *implicit type conversion* which occurs in a number of situations in the C language where the value of an expression is converted to a different type.

For example:

- A simple assignment operation (=). The value of the expression is implicitly converted to the type of the object to which the expression is being assigned.
- A compound assignment operation (*= /= += -= <<= >>= &= ^= |=). These operations are equivalent to 2 operations of which the second is a simple assignment.
- The initialisation of an arithmetic object at its point of definition.
- The assignment of a function argument to a function parameter in a function call.
- The assignment of the value of a *return* expression to the type of the notional object represented by the function return type.

All these operations share the common characteristic that the value of an expression may have to undergo an implicit conversion to another type 'as if by assignment' (a term used in the C language standard).

The precise message will depend on the following features of the expression being assigned:

1. The essential type of the expression
2. The standard type of the object to which the expression is being assigned.
3. Whether the expression is
    a. a constant – i.e. a floating constant or an integer constant of some form
    b. a constant expression
    c. a *non-constant expression* - any other type of expression
4. Whether the expression is a *composite* expression
5. The context in which the type conversion occurs - initialisation, assignment, function argument, function return.

6.4.1 Assigning a Constant

	Essential Type Category of Constant						
Standard Type of Object	Boolean	character	enum<1>	enum<2>	signed	unsigned	floating
_Bool		4410	4420	4420	1294 1257	1295 1257	1266
char	4401		4421	4421	1292 1257	1293 1257	1266
enum<1>	4402	4412		4422	1296 1257	1297 1257	1266
enum<2>	4402	4412	4422		1296 1257	1297 1257	1266
signed	4403	4413	4423	4423	1256 1257	1291 1256 1257	1266
unsigned	4404	4414	4424	4424	1290 1256 1257	1256 1257	1266
floating	4405	4415	4425	4425	1298	1299	1264 1265

6.4.2 Assigning a Constant Expression

	Essential Type Category of Constant Expression						
Standard Type of Object	Boolean	character	enum<1>	enum<2>	signed	unsigned	floating
_Bool		4410	4420	4420	4430	4440	4450
char	4401		4421	4421	4431	4441	4451
enum<1>	4402	4412		4422	4432	4442	4452
enum<2>	4402	4412	4422		4432	4442	4452
signed	4403	4413	4423	4423	4463	4447	4453
unsigned	4404	4414	4424	4424	4436	4464	4454
floating	4405	4415	4425	4425	4437	4445	4465

6.4.3 Assigning a Non-constant Expression

	Essential Type Category of Non-Constant Expression						
Standard Type of Object	Boolean	character	enum<1>	enum<2>	signed	unsigned	floating
_Bool		4410	4420	4420	4430	4440	4450
char	4401		4421	4421	4431	4441	4451
enum<1>	4402	4412		4422	4432	4442	4452
enum<2>	4402	4412	4422		4432	4442	4452
signed	4403	4413	4423	4423	4460 4470 4480 4490	4443 4446	4453
unsigned	4404	4414	4424	4424	4434	4461 4471	4454

						4481 4491	
floating	4405	4415	4425	4425	4435	4445	4462 4472 4482 4492

6.4.4 Assigning Messages

ETC of assigned expression	Message No.	Message Text
Boolean	4400	N/A
	4401	An expression of 'essentially Boolean' type (%1s) is being converted to character type
	4402	An expression of 'essentially Boolean' type (%1s) is being converted to enum type
	4403	An expression of 'essentially Boolean' type (%1s) is being converted to signed type
	4404	An expression of 'essentially Boolean' type (%1s) is being converted to unsigned type
	4405	An expression of 'essentially Boolean' type (%1s) is being converted to floating type
character	4410	An expression of 'essentially character' type (%1s) is being converted to Boolean type, '%2s' on assignment.
	4411	N/A
	4412	An expression of 'essentially character' type (%1s) is being converted to enum type, '%2s' on assignment.
	4413	An expression of 'essentially character' type (%1s) is being converted to signed type, '%2s' on assignment.
	4414	An expression of 'essentially character' type (%1s) is being converted to unsigned type, '%2s' on assignment.
	4415	An expression of 'essentially character' type (%1s) is being converted to floating type, '%2s' on assignment.
enum	4420	An expression of 'essentially enum' type (%1s) is being converted to Boolean type
	4421	An expression of 'essentially enum' type (%1s) is being converted to character type
	4422	An expression of 'essentially enum' type (%1s) is being converted to a different enum type
	4423	An expression of 'essentially enum' type (%1s) is being converted to signed type
	4424	An expression of 'essentially enum' type (%1s) is being converted to unsigned type
	4425	An expression of 'essentially enum' type (%1s) is being converted to floating type
signed	4430	An expression of 'essentially signed' type (%1s) is being converted to Boolean type, '%2s' on assignment.
	4431	An expression of 'essentially signed' type (%1s) is being converted to character type, '%2s' on assignment.
	4432	An expression of 'essentially signed' type (%1s) is being converted to enum type, '%2s' on assignment.
	4433	N/A
	4434	A non-constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
	4435	A non-constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
	4436	A constant expression of 'essentially signed' type (%1s) is being converted to unsigned type, '%2s' on assignment.
	4437	A constant expression of 'essentially signed' type (%1s) is being converted to floating type, '%2s' on assignment.
	4460	A non-constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
	4463	A constant expression of 'essentially signed' type (%1s) is being converted to narrower signed type, '%2s' on assignment.
	4470	A non-constant expression of 'essentially signed' type (%1s) is being passed to a function parameter of wider signed type, '%2s'.
	4480	A non-constant expression of 'essentially signed' type (%1s) is being returned from a function defined with a wider signed return type, '%2s'.
	1290	An integer constant of 'essentially signed' type is being converted to unsigned type on assignment.
	1292	An integer constant of 'essentially signed' type is being converted to type char on assignment.
	1294	An integer constant of 'essentially signed' type is being converted to type _Bool on assignment.
	1296	An integer constant of 'essentially signed' type is being converted to enum type on assignment.
	1298	An integer constant of 'essentially signed' type is being converted to floating type on assignment.
	1256	An integer constant suffixed with L is being converted to type signed or unsigned long long on assignment.
	1257	An integer constant suffixed with L or LL is being converted to a type of lower rank on assignment.
unsigned	4440	An expression of 'essentially unsigned' type (%1s) is being converted to Boolean type, '%2s' on assignment.
	4441	An expression of 'essentially unsigned' type (%1s) is being converted to character type, '%2s' on assignment.
	4442	An expression of 'essentially unsigned' type (%1s) is being converted to enum type, '%2s' on assignment.
	4443	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to a wider signed type, '%2s' on

ETC of assigned expression	Message No.	Message Text
		assignment.
	4444	N/a
	4445	An expression of 'essentially unsigned' type (%1s) is being converted to floating type, '%2s' on assignment.
	4446	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
	4447	A constant expression of 'essentially unsigned' type (%1s) is being converted to signed type, '%2s' on assignment.
	4461	A non-constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.
	4464	A constant expression of 'essentially unsigned' type (%1s) is being converted to narrower unsigned type, '%2s' on assignment.
	4471	A non-constant expression of 'essentially unsigned' type (%1s) is being passed to a function parameter of wider unsigned type, '%2s'.
	4481	A non-constant expression of 'essentially unsigned' type (%1s) is being returned from a function defined with a wider unsigned return type, '%2s'.
	1291	An integer constant of 'essentially unsigned' type is being converted to signed type on assignment.
	1293	An integer constant of 'essentially unsigned' type is being converted to type char on assignment.
	1295	An integer constant of 'essentially unsigned' type is being converted to type _Bool on assignment.
	1297	An integer constant of 'essentially unsigned' type is being converted to enum type on assignment.
	1299	An integer constant of 'essentially unsigned' type is being converted to floating type on assignment.
	1256	An integer constant suffixed with L is being converted to type signed or unsigned long long on assignment.
	1257	An integer constant suffixed with L or LL is being converted to a type of lower rank on assignment.
floating	4450	An expression of 'essentially floating' type (%1s) is being converted to Boolean type '%2s' on assignment.
	4451	An expression of 'essentially floating' type (%1s) is being converted to character type '%2s' on assignment.
	4452	An expression of 'essentially floating' type (%1s) is being converted to enum type '%2s' on assignment.
	4453	An expression of 'essentially floating' type (%1s) is being converted to signed type '%2s' on assignment.
	4454	An expression of 'essentially floating' type (%1s) is being converted to unsigned type '%2s' on assignment.
	4455	N/A
	4462	A non-constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on assignment.
	4465	A constant expression of 'essentially floating' type (%1s) is being converted to narrower floating type, '%2s' on assignment.
	4472	A non-constant expression of 'essentially floating' type (%1s) is being passed to a function parameter of wider floating type, '%2s'.
	4482	A non-constant expression of 'essentially floating' type (%1s) is being returned from a function defined with a wider floating return type, '%2s'.
	1264	A suffixed floating constant is being converted to a different floating type on assignment.
	1265	An unsuffixed floating constant is being converted to a different floating type on assignment.
	1266	A floating constant is being converted to integral type on assignment.

6.5 Balancing Type Conversion Messages

The term “type balancing” describes what happens when 2 operands of different arithmetic type undergo implicit conversion to a common type in the context of certain operators. This process is referred to in the ISO C language standard as the “usual arithmetic conversions”. The operators where type balancing occurs are:

- Arithmetic: + - * / %
- Bitwise: & | ^
- Relational: < <= >= >
- Equality: == !=
- Conditional: ? : (2nd and 3rd operands)

These messages may be used to identify situations where dangerous or inappropriate type balancing occurs between operands of different essential type. Notice that other contexts where the type of an individual

operand is inappropriate (regardless of the type of other operands), are identified in a previous section.

6.5.1 Integer / Float Conversions:

	Boolean	character	enum	signed	unsigned	floating
Boolean						
character						
enum						X
signed						X
unsigned						X
floating			*x*	*x*	*x*	

The following messages are generated when an expression of *essentially enum*, *essentially signed* or *essentially unsigned* type is balanced with an operand of *essentially floating* type.

Message No.	Message Text
1800	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this arithmetic operation.
N/A	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this bitwise operation.
1802	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this relational operation.
1803	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this equality operation.
1804	The %1s operand (essential type: '%2s') will be implicitly converted to a floating type, '%3s', in this conditional operation.

6.5.2 Addition and Subtraction of Essentially Character Expressions

	Boolean	character	enum	signed	unsigned	floating
Boolean						
character		x		x	x	
enum						
signed		x				
unsigned		x				
floating						

The following messages are generated when an expression of *essentially character* type is balanced in an inappropriate way in an addition or subtraction operation.

Message No.	Message Text
1810	An operand of 'essentially character' type is being added to another operand of 'essentially character' type
1811	An operand of 'essentially character' type is being subtracted from an operand of 'essentially signed' type
1812	An operand of 'essentially character' type is being subtracted from an operand of 'essentially unsigned' type

6.5.3 Signed / Unsigned Conversions

	Boolean	character	enum	signed	unsigned	floating
Boolean						
character						
enum						
signed				x	x	
unsigned				x	x	
floating						

An extensive set of messages is associated with the situation when expressions of *essentially signed* and *essentially unsigned* type are balanced. The messages distinguish 5 subtly different variations in the way that these type conversions may be described:

A. The *essentially signed* operand is non-constant and is converted to an *unsigned* type.
B. The *essentially signed* operand is a constant expression of negative value and is converted to an *unsigned* type.
C. The *essentially signed* operand is a constant expression of non-negative value and is converted to an *unsigned* type.
D. The *essentially unsigned* operand is implicitly converted to the *essential type* of the *signed* operand.
E. The *essential types* of the two operands are both of lower rank than *int*. The result is of type *signed int* because of integral promotion.

The signed/unsigned messages are further extended by generating a different message for each of the 5 classes of operator, i.e.

1. Arithmetic
2. Bitwise
3. Relational
4. Equality
5. Conditional

		Message No.	Message Text
A	1	1820	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
A	2	1821	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
A	3	1822	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this relational operation.
A	4	1823	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this equality operation.
A	5	1824	The %1s operand is non-constant and 'essentially signed' (%2s) but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
B	1	1830	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
B	2	1831	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
B	3	1832	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
B	4	1833	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
B	5	1834	The %1s operand is constant, 'essentially signed' (%2s) and negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
C	1	1840	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this arithmetic operation.
C	2	1841	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this bitwise operation.
C	3	1842	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this relational operation.
C	4	1843	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this equality operation.
C	5	1844	The %1s operand is constant, 'essentially signed' (%2s) and non-negative but will be implicitly converted to an unsigned type (%3s) in this conditional operation.
D	1	1850	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this arithmetic operation.
D	2	1851	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this bitwise operation.
D	3	1852	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this relational operation.
D	4	1853	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this equality operation.
D	5	1854	The %1s operand is 'essentially unsigned' (%2s) but will be implicitly converted to a signed type (%3s) in this conditional operation.
E	1	1860	The operands of this arithmetic operator are of different 'essential signedness' but will generate a result of type 'signed int'.
E	2	1861	The operands of this bitwise operator are of different 'essential signedness' but will generate a result of type 'signed int'.
E	3	1862	The operands of this relational operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
E	4	1863	The operands of this equality operator are of different 'essential signedness' but will both be promoted to 'signed int' for comparison.
E	5	1864	The 2nd and 3rd operands of this conditional operator are of different 'essential signedness'. The result will be in the promoted type 'signed int'.

6.5.4 Operands of Different ET Category in a Relational Operation

	Boolean	character	enum	signed	unsigned	floating
Boolean						
character			x			x
enum		x	x	x	x	
signed			x			
unsigned			x			
floating		x				

Message No.	Message Text
1880	The operands of this relational operator are expressions of different 'essential type' categories (%1s and %2s).

6.5.5 Operands of Different ET Category in an Equality or Conditional Operation

	Boolean	character	enum	signed	unsigned	floating
Boolean		x	x	x	x	x
character	x		x			x
enum	x	x	x	x	x	
signed	x		x			
unsigned	x		x			
floating	x	x				

Message No.	Message Text
1881	The operands of this equality operator are expressions of different 'essential type' categories (%1s and %2s).
1882	The 2nd and 3rd operands of this conditional operator are expressions of different 'essential type' categories (%1s and %2s).

6.6 Switch Case Type Conversion Messages

According to the ISO C99 language standard, section 6.8.4.2, when a switch statement is executed:

*"The integer promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed."*

Messages are generated on switch case labels when the essential type of the label is not consistent with the essential type of the switch controlling expression.

	ETC OF SWITCH CONTROLLING EXPRESSION					
ETC OF CASE LABEL	**Boolean**	**character**	**enum<1>**	**enum<2>**	**signed**	**unsigned**
Boolean		570	570	570	570	570
character	570		570	570	570	570

enum<1>	570	570		570	570	570
enum<2>	570	570	570		570	570
signed	570	570	570	570	571 572	570
unsigned	570	570	570	570	570	571 572

Message No.	Message Text
570	This switch case label of 'essential type' '%1s' is not consistent with a controlling expression of essential type '%2s'.
571	This switch case label of 'essential type' '%1s' is not consistent with a controlling expression which has an essential type of higher rank (%2s).
572	This switch case label of 'essential type' '%1s' is not consistent with a controlling expression which has an essential type of lower rank (%2s).

- Message 570 is generated when
  - the essential type category of the switch case label differs from the essential type category of the switch controlling expression, or
  - the case label and controlling expression are of different enum types.
- Message 571 is generated when
  - the case label and controlling expression are both essentially signed or both essentially unsigned, and
  - the controlling expression is of type signed long long or unsigned long long, and
  - the case label is a constant suffixed with L or UL.
- Message 572 is generated when
  - the case label and controlling expression are both essentially signed or both essentially unsigned, and
  - the essential type of the controlling expression is smaller than long, and
  - the case label is a constant suffixed with L, UL, LL or ULL.

6.7 Cast Operation Messages

According to the ISO C standard, casts may be used without restriction to convert the value of any expression of arithmetic type to some specified arithmetic type. The freedom that the C language allows may sometimes conceal cast operations which are of very doubtful meaning. For example:

- Casting an expression of Boolean type to type char
- Casting an expression of type char to type float

The following table displays a range of messages which may be used to identify suspicious cast operations.

	Essential Type Category of Operand						
Standard Type of Result	Boolean	character	enum<1>	enum<2>	signed	unsigned	floating
_Bool		4310	4320	4320	4330	4340	4350
char	4301				4393	4394	4351
enum<1>	4302	4312		4322	4332	4342	4352
enum<2>	4303	4312	4322		4332	4342	4352

Signed	4304				4390	4394	4395
unsigned	4305				4393	4391	4395
floating	4306	4315	4325	4325	4393	4394	4392

ETC of cast expression	Message No.	Message Text
Boolean	4301	An expression of 'essentially Boolean' type (%1s) is being cast to character type '%2s'.
	4302	An expression of 'essentially Boolean' type (%1s) is being cast to enum type, '%2s'.
	4303	An expression of 'essentially Boolean' type (%1s) is being cast to signed type, '%2s'.
	4304	An expression of 'essentially Boolean' type (%1s) is being cast to unsigned type, '%2s'.
	4305	An expression of 'essentially Boolean' type (%1s) is being cast to floating type, '%2s'.
	4301	An expression of 'essentially Boolean' type (%1s) is being cast to character type '%2s'.
character	4310	An expression of 'essentially character' type (%1s) is being cast to Boolean type, '%2s'.
	4312	An expression of 'essentially character' type (%1s) is being cast to enum type, '%2s'.
	4315	An expression of 'essentially character' type (%1s) is being cast to floating type, '%2s'.
enum	4320	An expression of 'essentially enum' type (%1s) is being cast to Boolean type, '%2s'.
	4322	An expression of 'essentially enum' type (%1s) is being cast to type a different enum type, '%2s'.
	4325	An expression of 'essentially enum' type (%1s) is being cast to floating type, '%2s'.
signed	4330	An expression of 'essentially signed' type (%1s) is being cast to Boolean type '%2s'.
	4332	An expression of 'essentially signed' type (%1s) is being cast to enum type, '%2s'.
unsigned	4340	An expression of 'essentially unsigned' type (%1s) is being cast to Boolean type '%2s'.
	4342	An expression of 'essentially unsigned' type (%1s) is being cast to enum type, '%2s'.
floating	4350	An expression of 'essentially floating' type (%1s) is being cast to Boolean type '%2s'.
	4351	An expression of 'essentially floating' type (%1s) is being cast to character type '%2s'.
	4352	An expression of 'essentially floating' type (%1s) is being cast to enum type, '%2s'.

6.8 Bitwise ~ and << Messages

MISRA-C imposes some particular constraints on the way in which bitwise ~ and << may be used. Bitwise operations such as ~ and << can generate high order bits in the promoted type of expressions. Problems arise when these operators are used on an operand whose essential type is smaller than int.

```
MISRA-C2 Rule 10.5:
If the bitwise operators ~ and << are applied to an operand of underlying
type unsigned char or unsigned short, the result shall be immediately
cast to the underlying type of the operand.
```

Message No.	Message Text
4397	An expression which is the result of a ~ or << operation has not been cast to its essential type.
4398	An expression which is the result of a ~ or << operation has been cast to a different essential type category.
4399	An expression which is the result of a ~ or << operation has been cast to a wider type.
4498	An expression which is the result of a ~ or << operation has been converted to a different essential type category on assignment.
4499	An expression which is the result of a ~ or << operation has been converted to a wider essential type on assignment.
4570	The operand of this ~ operator has an 'essential type' which is narrower than type 'int'.
4571	The left-hand operand of this << operator has an 'essential type' which is narrower than type 'int'.

These messages are generated by examining the context when a ~ or << operation is encountered

according to the following algorithm:

```
if (essential type of operand is "narrower" than "int")
{
   if (result is the operand of a cast operator)
   {
      if the cast is to a different essential type category
      {
         Message 4398
      }
      else
      {
         if the cast is to a "wider" essential type
         {
            Message 4399
         }
      }
   }
   else if (result is used in an assigning operation)
   {
      if the assignment is to a different essential type category
      {
          Message 4498
      }
      else
      {
         if the assignment is to a "wider" essential type
         {
            Message 4499
         }
      }
   }
   else
   {
      Message 4397
   }
}
```

6.9 Composite Expression Messages

QA·C generates messages which identify when:

a) The value of a composite expression is implicitly converted to a wider *essential type* in an assigning operation

b) The value of a composite expression is used as one operand of an operator in which the *usual arithmetic conversions* (“type balancing”) are performed and the other operand is of wider *essential type*. Type balancing is conducted in the following operations:

- Multiplicative * / %
- Additive + -
- Bitwise & | ^
- Relational < <= >= >
- Equality == !=
- Conditional ? :

c) The value of a *composite expression* is cast to a different *essential type category.*

d) The value of a *composite expression* is shall not be cast to a wider *essential type*.

Message No.	Message Text
4490	A composite expression of 'essentially signed' type (%1s) is being converted to wider signed type, '%2s' on assignment.
4491	A composite expression of 'essentially unsigned' type (%1s) is being converted to wider unsigned type, '%2s' on assignment.
4492	A composite expression of 'essentially floating' type (%1s) is being converted to wider floating type, '%2s' on assignment.
1890	A composite expression of 'essentially signed' type (%1s) is being implicitly converted to a wider signed type, '%2s'.
1891	A composite expression of 'essentially unsigned' type (%1s) is being implicitly converted to a wider unsigned type, '%2s'.
1892	A composite expression of 'essentially floating' type (%1s) is being implicitly converted to a wider floating type, '%2s'.
4393	A composite expression of 'essentially signed' type (%1s) is being cast to a different type category, '%2s'.
4394	A composite expression of 'essentially unsigned' type (%1s) is being cast to a different type category, '%2s'.
4395	A composite expression of 'essentially floating' type (%1s) is being cast to a different type category, '%2s'.
4390	A composite expression of 'essentially signed' type (%1s) is being cast to a wider signed type, '%2s'.
4391	A composite expression of 'essentially unsigned' type (%1s) is being cast to a wider unsigned type, '%2s'.
4392	A composite expression of 'essentially floating' type (%1s) is being cast to a wider floating type, '%2s'.