

# **Escher C/C++ Verifier 7.0**

## **Reference Manual**

© 2017 Escher Technologies Ltd. All rights reserved.

# Escher C Verifier (*eCv*) Reference Manual

Version 7.0, February 2017

## *Disclaimer*

The information in this publication is given in good faith but may contain errors and omissions. The contents of this document and the specifications of *eCv* are subject to change without notice.

## Contents

### 1. Getting Started

- [What \*eCv\* is intended for](#)
- [Principles](#)
- [Programming languages supported by \*eCv\*](#)
- [Installation and configuration of \*eCv\*](#)
- [Getting ready to use \*eCv\*](#)
- [Setting up your first \*eCv\* project](#)
- [Running your \*eCv\* project](#)

### 2. Making your source code compatible with *eCv*

- [Overview](#)
- [Keywords](#)
- [Restrictions](#)
- [Defining and Using Boolean types](#)
- [Pointers](#)
- [Assertions and assert.h](#)
- [Extensions to the C and C++ languages](#)
- [Common error messages](#)
- [Line and column coordinates in messages](#)
- [Suppressing warning messages](#)

### 3. Verifying your source

- [Ensuring validity](#)
- [Which preprocessor?](#)
- [Verification](#)
- [Constructs that are unverifiable](#)

### 4. Specifications

- [General Notes](#)
- [Type constraints](#)
- [Function Contracts](#)
- [Loop specifications](#)
- [Ghost declarations](#)

## **5. Additional eCv Constructs**

[Additional eCv Declarations](#)

[Additional eCv Statements](#)

[Additional eCv Expressions](#)

## **6. Predefined ghost types, functions and fields**

[Global ghost variables](#)

[Global ghost functions](#)

[Ghost fields of array pointer types](#)

[Ghost fields of the void pointer type](#)

[Ghost fields of array types](#)

[Ghost member functions of array and sequence types](#)

## **7. Support for C++ source code in eCv++**

[Supported and unsupported constructs](#)

[Additional eCv++ keywords](#)

[Casts in C++ programs](#)

[Overloaded functions and ambiguity resolution](#)

[Inheritance and final classes](#)

[Polymorphic pointers](#)

[Constructors](#)

[Member functions](#)

[Early member functions](#)

[Function overriding and hiding](#)

[Specifications of overriding functions](#)

[Operator declarations](#)

[Templates](#)

## **Appendix A - Compiler Settings**

[Microsoft Visual C++](#)

[gcc](#)

## **Appendix B - Type system of eCv**

## **Appendix C - Constructs you may see in proof output**

## **Appendix D - Verification condition types**

## **Appendix E - Language extensions for C99 and C++ 2011**

[C99](#)

[C++ 2011](#)

## **Appendix G - Differences from MISRA-C 2004 and MISRA-C 2012**

[MISRA-C 2004 guidelines not fully enforced by eCv](#)

[eCv rules that are stronger than the MISRA-C 2004 guidelines](#)

[MISRA-C 2012 guidelines not fully enforced by eCv](#)

[eCv rules that are stronger than the MISRA-C 2012 guidelines](#)



# Introduction to Escher C Verifier

**eCv is a tool for verifying programs written in a subset of C or C++. Unlike traditional static analysis tools, eCv verifies software so that it is mathematically proven not only to be free from run-time errors, but also to be correct with respect to a well-defined functional specification.**

## What eCv is intended for

eCv has been designed for developing and verifying **critical embedded software**. Therefore, eCv requires you to write your program in a safe, type-strengthened subset of C (based on MISRA-C 2004). Constructs that are not suitable for use in safety-critical embedded software - for example, dynamic memory allocation and concurrency - are either not supported at all, or supported only when they are used in certain ways. If you are looking to verify C programs that are not written with safety in mind, then eCv may not be the tool you are looking for.

Because eCv supports only a safe subset of C, the annotation language of eCv is simpler than the annotation language of more general tools, making eCv more suitable for ordinary software developers.

Unlike other formal tools for C, eCv provides a mechanism for expressing high-level software requirements and specifications as well as low-level ones. In support of this, eCv provides abstract data types such as sets and sequences, together with a wide range of operations on them.

We strongly recommend that you use eCv alongside your compiler when developing your software, rather than applying eCv only when development is believed to be complete. Indeed, we suggest running code through eCv in *Check* mode even before you compile it, to check that your code complies with the eCv safe subset of C. Verifying previously-developed code with eCv is practical only if the source code was written to a high standard *and* you are prepared to make changes to it.

## Principles

eCv uses the Verified Design-by-Contract (VDbC) paradigm and a powerful automatic theorem prover (the same theorem prover as in Escher Technologies' flagship product *Perfect Developer*). Verified Design-by-Contract builds on the widely-known design-by-contract principle by adding automated proof that contracts are fulfilled or not. All **possible** violations of contract are detected by eCv prior to compilation. This is clearly preferable to the alternative of hoping that during testing, the test cases used were sufficient to detect all contract violations.

Furthermore, eCv recognizes that many programming language constructs and library functions have implicit contracts; for example, preconditions that are required to hold in order to avoid undefined or implementation-defined behaviour. eCv verifies that these preconditions hold, too.

Preconditions and other specification annotations for eCv are expressed using some additional keywords, with bracketed arguments where needed. When you compile your program using a regular C or C++ compiler, these additional keywords are defined as macros with empty expansions. This means that the specifications are invisible to the compiler, which can still translate your source code as if the specifications were not there.

## Programming languages supported by eCv

You can write programs for processing with eCv in subsets of C90, C99, or C++, depending on what compiler you intend to use.

Any program which is a valid eCv program and which is also a valid program in two (or even all three) of those languages has the same meaning in both (all) of them, provided that the compilers you use implement implementation-dependent features of C/C++ in the same way. This makes it easier for you to switch from using a C90 compiler to using a C++ compiler, for example.

# Installation and configuration of eCv

Install *Escher Verification Studio* from the CD or download, ensuring that you include the *Escher C Verifier* sub-feature.


Load *Escher Verification Studio*.

Go to **Options** → **Editor** and configure eCv to use your preferred editor. Assuming that your editor supports syntax highlighting, we recommend that you also adjust the configuration of the editor itself to highlight the additional eCv language keywords, possibly in a different colour. See [here](#) for a list of additional keywords, and the *EditorCustomizations* subfolder of the *Escher Verification Studio* installation for preconfigured syntax definition files for some popular editors. If you don't already have an editor that supports syntax highlighting and you are running under Windows, you can find a free editor (Crimson) and an evaluation copy of an inexpensive editor (TextPad) on the *Escher C Verifier* installation CD.

Optionally, go to **Options** → **C/C++ Compilers** and create an entry for each installed C or C++ compiler that your code will be compiled with. See [Appendix A](#) for some sample compiler configurations. If a compiler supports different modes (e.g. it can compile in C mode or in C++ mode), or you wish to target more than one platform using the same compiler, you may wish to set up more than one entry for that compiler.

## Getting ready to use eCv

File *ecv.h* is provided with eCv. You will find it in folder "C:\Program Files\Escher Technologies\Verification Studio 5\Escher C Verifier\Include" (replace "C:\Program Files" by your program root directory for 32-bit applications, for example "C:\Program Files (x86)" under 64-bit Windows). You must include a directory that contains *ecv.h* in your compiler's file include path **and** in eCv's include path. Either copy *ecv.h* into the directory where your own standard header file is kept, or configure your development environment or build system to include the installed location of *ecv.h* in the include file path when you run your compiler(s).

Each C or C++ source file must directly or indirectly **#include** file "*ecv.h*". This inclusion must come before any specifications or other eCv constructs in the source file, and before including other files that contain specifications or other eCv constructs. We recommend that you **#include "ecv.h"** right at the beginning of one of your own header files (for example, you may already have a header file that defines integral types of fixed sizes taking account of the target platforms), and then **#include** that file right at the start of every .c file. 

Unless you are writing in C++, you also need to set up standard definitions for a Boolean type, or make your existing Boolean type definition understandable to eCv. See the section on [Defining and Using Boolean types](#). We suggest that you put these definitions in the same header file in which you **#include "ecv.h"**.

Depending on your source code, eCv may need to know the definitions of types *size\_t*, *ptrdiff\_t* and (unless your source is C++) *wchar\_t*. These are all defined in the standard header file **stddef.h** for C90 or C99, or **cstddef** for C++. Therefore, we suggest that you **#include** this file in the header file that includes "*ecv.h*" too.

## Setting up your first eCv project

### Command line or Project Manager?

eCv may be used from the command line, or from the graphical user interface provided by the Project Manager of *Escher Verification Studio*. To run eCv from the command line (recommended for advanced users only), you will need to set up a batch file or shell script to invoke *EscherTool*, optionally passing your C/C++ source code through your compiler's preprocessor first. The command line syntax for *EscherTool* is given in the *Escher Verification Studio User Guide*. The remainder of this document assumes that you are using the Project Manager.

To create your first project, first load the Project Manager via the *Escher Verification Studio* shortcut or the file *VerificationStudio.exe*. Next, either go to **File** → **New Project** or click on the blank-sheet button on the toolbar. If you are asked what sort of files your project will contain, select **C/C++ files**. Choose a name and folder location for your project.

## Adding and creating source files

Each project contains a list of the C/C++ source files that you want eCv to process, along with information about which compiler(s) you will be using. If you have existing C or C++ source files for the project, add them to the project (use **File** → **Add**, or the + toolbar button). If you wish to create new source files, use **File** → **Add New**, or the \*+ toolbar button. Remember that each source file must (directly or indirectly) `#include "ecv.h"`.

## Configuring the project settings

At this stage, the only item that it is essential to configure is your choice of C or C++ compiler. Select **Project** → **Settings** from the menu, or click on the cog icon in the toolbar. In the **Compiler** box, select one of the compilers you previously configured, or one of the generic ones that was created for you when *Escher C Verifier* was installed. In the **Additional include paths** box, use the **Add** button to add the path to file `ecv.h`, or to your own header file that includes `ecv.h`.

## Essential annotations

We suggest that if you added existing source files to the project, you go through them and add at least the following types of annotation before running eCv on them:

- **array** annotations on array pointer variables, parameters and fields
- **null** annotations on nullable pointer variables, parameters and fields
- preconditions on any functions that take null-terminated strings as parameters

When you are writing new source, it is best to add these annotations, along with preconditions and other specifications, as you write the code.

## Running your eCv project

Run syntax and semantic checks on all files by pressing the yellow-tick button on the toolbar. You can run checks on individual files by right-clicking on the file in the Project Manager window and selecting **Check**.

Resolve any error messages by appropriate changes to source and/or header files. See the chapter on [Making your source code compatible with eCv](#). Then you can proceed to [Verification](#).

Further instructions on using the Project Manager, together with instructions on running eCv from the command line, can be found in the *Escher Verification Studio User Guide*.

 [TOC](#)

eCv Manual, Version 7.0, February 2017.

© 2017 Escher Technologies Limited. All rights reserved.

# Making your source code compatible with eCv

## Overview

You can configure eCv to assume source code is C90, C99, C++ 1998 or C++ 2011. The choice is made in the compiler configuration dialog in the Project Manager, or using the `-gl` option if you are running eCv from the command line.

The following describes the core C90 language subset supported by eCv, in particular the restrictions placed on C constructs. Where these restrictions have equivalent or related MISRA-C 2004 rules, we quote the rule numbers.

Not all of the restrictions are rigidly enforced; some will give rise to warning (rather than error) messages if they are violated, allowing analysis to continue.

For information on constructs from C99 and C++ that eCv supports when you select on of those languages, see [Appendix E](#).

## Keywords

**eCv treats some additional words as keywords. These words may not be used as identifiers. Here is a list of them:**

---

Normal keyword	Underlying keyword	Where used	See section
any	<code>_ecv_any</code>	'any' expression	<a href="#">Collections</a>
array	<code>_ecv_array</code>	Indicates that a pointer refers to an array	<a href="#">array</a>
assert	<code>_ecv_assert</code>	Assertion statement	<a href="#">assert</a>
assume	<code>_ecv_assume</code>	Assume declaration	<a href="#">assume</a>
bool		Boolean type	<a href="#">Boolean types</a>
decrease	<code>_ecv_decrease</code>	Declares a loop variant	<a href="#">decrease</a>
exists	<code>_ecv_exists</code>	'exists' expression	<a href="#">Quantified expressions</a>
false		Boolean false value	<a href="#">Boolean types</a>
forall	<code>_ecv_forall</code>	'forall' expression	<a href="#">Quantified expressions</a>
ghost	<code>_ecv_ghost</code>	Declares that a declaration is for use in specifications only	<a href="#">ghost</a> (see also <a href="#">Ghosts</a> )
holds	<code>_ecv_holds</code>	'holds' expression	<a href="#">Disjoint expressions</a>
idiv	<code>_ecv_idiv</code>	An integer division operator that always rounds down	<a href="#">Binary operators</a>
imod	<code>_ecv_imod</code>	An integer modulus operator that always yields a non-negative result	<a href="#">Binary operators</a>
in	<code>_ecv_in</code>	Element-in-collecton operator	<a href="#">Binary operators</a>
integer	<code>_ecv_integer</code>	Unbounded integer type	<a href="#">ghost</a>
	<code>_ecv_interrupt</code>	Function specifier to indicate that the function is an interrupt service routine	(to be added)



invariant	<code>_ecv_invariant</code>	Declares a structure invariant	<a href="#">invariant</a>
keep	<code>_ecv_keep</code>	Declares a loop invariant	<a href="#">keep</a>
let	<code>_ecv_let</code>	Names a subexpression so you can refer to it in a larger expression	<a href="#">let</a>
maxof	<code>_ecv_maxof</code>	Yields the maximum value in a type	<a href="#">Type operators</a>
minof	<code>_ecv_minof</code>	Yields the minimum value in a type	<a href="#">Type operators</a>
min_sizeof	<code>_ecv_minof</code>	Yields the minimum value that sizeof could yield for the same type	<a href="#">Type operators</a>
not_null	<code>_ecv_not_null</code>	Cast a nullable pointer to a non-nullable pointer	<a href="#">not_null</a>
null	<code>_ecv_null</code>	Declares that a pointer is nullable	<a href="#">Nullable and non-nullable pointers</a>
	<code>_ecv_nullptr</code>	Null pointer literal	<a href="#">Additional eCv++ keywords</a>
old	<code>_ecv_old</code>	Selects the original value of an expression instead of the final or current value in a postcondition, loop invariant or loop variant	<a href="#">old</a>
out	<code>_ecv_out</code>	Indicates that a pointer parameter is used only to pass a value back to the caller	<a href="#">Pointers as function parameters</a>
over	<code>_ecv_over</code>	'over' expression	<a href="#">Collections</a>
pass	<code>_ecv_pass</code>	Null statement	<a href="#">pass</a>
post	<code>_ecv_post</code>	Declares a postcondition	<a href="#">post</a>
	<code>_ecv_pow</code>	Exponentiation operator	<a href="#">Binary operators</a>
pre	<code>_ecv_pre</code>	Declares a precondition	<a href="#">pre</a>
result	<code>_ecv_result</code>	Refers to function result in a postcondition	<a href="#">post</a>
returns	<code>_ecv_returns</code>	Declares a function return value	<a href="#">returns</a>
some	<code>_ecv_some</code>	Indicates any object(s) of a specified type	
spec	<code>_ecv_spec</code>	Declares that a function prototype overrides another similar one	
that	<code>_ecv_that</code>	'that' expression	<a href="#">Collections</a>
those	<code>_ecv_those</code>	'those' expression	<a href="#">Collections</a>
true		Boolean true value	<a href="#">Boolean types</a>
wchar_t		Wide character type	<a href="#">below</a>
writes	<code>_ecv_writes</code>	Declares nonlocal variables that a function writes to	<a href="#">writes</a>
value	<code>_ecv_value</code>	Refers to the value of the structure in a structure invariant, similar to (*this) in C++	
yield	<code>_ecv_yield</code>	'for..yield' and 'for..those..yield' expressions	<a href="#">Collections</a>
zero_init	<code>_ecv_zero_init</code>	Yields the value of a type obtained by setting all bits to zero	<a href="#">Type operators</a>

## Notes on keywords

If you **must** use one of the keywords in the above table as an identifier in your program (perhaps because a third-party library header file uses it) then there is a workaround. Apart from **bool**, **false**, **true** and **wchar\_t**, all the reserved words not beginning with `_ecv_` are defined in file `ecv.h` as macros equivalent to the corresponding `_ecv_` versions. After you `#include "ecv.h"` at the start of your program, you may `#undef` the keyword. So if you need to use e.g. `result` as an identifier, you can `#undef result`, and then subsequently use `_ecv_result` instead of `result` in eCv specifications.

Note: if you use keywords beginning with `_ecv_` directly, then this may result in the column numbers in eCv error, warning and informational messages being incorrect (see later in this chapter).

eCv also places a number of identifiers beginning with `_ecv_` in the global namespace, therefore you should not use identifiers beginning with `_ecv_` in your source code.

Although we have listed `wchar_t` as a reserved word, if your source code is C90 or C99 (not C++) then you are allowed to use a `typedef` in a standard header file to define `wchar_t` with its usual meaning, i.e. the type of a wide character literal.

If you are using any of the keywords `{bool, true, false}` to define your own Boolean type, you should make your own definitions conditional - see [here](#).

## Restrictions

eCv places the following restrictions on using of the C language.

### Tokens

eCv treats `:"-` as a single token, rather than the two tokens `":"` and `"-"`. This means that conditional expressions such as the following, will not be parsed correctly:

```
c ? e : -f
c ? e : --x
```

The fix is simple: ensure that there is at least one space between the colon and the minus-sign, as in the following:

```
c ? e : -f
c ? e : --f
```

### Declarations

- In any scope, you may not use the same identifier as more than one of:
  - the name of a single variable, function or parameter, and/or the name of one or more **struct** or **union** fields
  - a struct name
  - a union name
  - a typedef name
  - an enum name

So, at any location in the program, a given identifier that does not immediately follow a period has exactly one meaning. **(MISRA 5.6)**

- You may not declare an identifier in an inner scope so that it hides a declaration of the same identifier in an enclosing scope. This includes parameter names in prototypes [because prototypes may contain specifications that refer to the parameters]. **(MISRA 5.2)**  
Advice: avoid declaring **static** or **extern** variables/functions with short names that you may also want to use as parameter names.
- eCv currently requires the names of parameters in function prototypes to match exactly with the corresponding parameter names in the function definition. **(MISRA 16.4)**
- Storage-class specifiers must be placed before type qualifiers. For example, you may use **extern const ...** or **static volatile ...** , but not **const extern ...** or **volatile static ...** .
- You may not declare a **struct**, **union** or **enum** within a parameter list.
- You may not declare a **struct**, **union** or **enum** within the operand of **sizeof**.
- You may not declare a **struct**, **union** or **enum** within the type-part of a cast expression.
- Variables must be explicitly typed (i.e. no default of **int**). **(MISRA 8.2)**
- eCv currently requires that all types declared are also defined. **(MISRA 18.1)** So if your source file refers to type **struct Foo \*** then type *Foo* must be defined somewhere. If necessary, declare a dummy definition, like this:

```
#ifdef ECV
    struct Foo { int x; }
#endif
```

This dummy definition includes a member so that eCv cannot infer that all instances of Foo are equal.

- When initializing arrays of arrays, arrays of structs, structs containing arrays etc. each part of the initializer list for any array or struct must be enclosed in braces. **(MISRA 9.2)**
- When you declare a variable with static or extern linkage, if the type of the variable does not have a valid default-initialized value, then the declaration must include an initializer. For example, if you declare a static variable with non-nullable pointer type, or with struct type where the struct contains a field with non-nullable pointer type, then you must provide an initializer. Similarly, if you declare a static array whose element type does not have a valid default initial value, then you must provide an initializer for all the elements of the array.

## Pointers summary

- A pointer variable or parameter may only be given the value 0 if it has been declared nullable. See later section on pointers and arrays.
- Pointer variables in static data that are not declared nullable must have initializers.
- Pointers to arrays are distinguished from plain pointers (i.e. pointers to single values) by the keyword **array** after the \* in the corresponding declaration. See later section on pointers and arrays.
- A plain pointer may not be indexed or have any other operator applied to it other than \* or == or != . **(MISRA 17.1, 17.3)**
- A function parameter of pointer type that is used only to pass a value back to the caller should be flagged with the keyword **out**.

## Characters and character types

- eCv treats plain **char** as distinct from both **signed char** and **unsigned char** (like C++). eCv does not perform implicit type conversions to or from plain **char**. **(MISRA 6.1, 6.2)**
- The type of a character literal in eCv is **char** (as in C++), not **int** (as in C).
- eCv treats **wchar\_t** as a separate type distinct from all other types (like C++). eCv does not perform implicit type conversions to or from **wchar\_t**.
- The type of a wide character literal in eCv is **wchar\_t**.

## Types and type conversions

- eCv does not perform implicit type conversions between integral types and floating-point types. Any such conversion required must be done using an explicit cast. **(MISRA 10.1)**
- eCv generates a warning where there is an implicit type conversion and the destination type cannot contain all the values of the source type. **(MISRA 10.1, 10.2)** Exception: eCv will not warn if an integer *literal* having a signed type is implicitly converted to an unsigned type with the same or a larger number of bits, because an integer literal is non-negative, so there is no risk that it will not fit in the type.
- Where a string literal is converted implicitly into a pointer to the first character (i.e. whenever it is not used as an array initializer or the operand of **sizeof**), its type in eCv is **const char\* array** (i.e. **const char\*** as in C++, with the eCv **array** modifier), not **char\*** as in C. Note the **const**. This means that you can't modify a string literal (which would in any case amount to "undefined behaviour" in the C standard), or pass a string literal to a function that takes a **char\*** (without the **const**) parameter.
- Enum types are treated as separate types (as in C++), not as a shorthand for integers. Enum types can be converted to integers implicitly. Conversion from integral types to enum types can be done using an explicit cast - there is no implicit conversion. You cannot use the operators ++, -- or the assigning operators +=, -= etc. on enum types (they are normally allowed by the C language standards but not by the C++ standard).
- eCv uses a distinct Boolean type called **bool**, and treats the result of evaluating a relational operator as having type **bool**. The condition part of a conditional expression or **if**-statement, and the while-expression in a **while**, **do...while** or **for** loop must have type **bool**. **(MISRA 13.2)** If you already define your own Boolean

type, see [here](#) for how to make eCv understand it.

- A cast expression must do one of the following:
  - Convert between two integral type, two floating-point type, or one integral type and one floating-point type (note that plain **char** and **wchar\_t** are not treated by eCv as integral types for the purpose of this rule)
  - Convert from **char** to **signed char** or **unsigned char**
  - Convert from an integral type to **char**
  - Convert between **wchar\_t** and an integral type
  - Convert between an enumeration type and an integral type
  - Convert between **bool** and an integral type
  - Convert from **[const] [volatile] T1\* [array] [null]** to **[const] [volatile] T2\* [array] [null]**, where T1 and T2 are the same type or one of them is **void**, and elements shown here in [ ] are optional. There are some further restrictions: you can't cast from a **const**-pointer to a non-**const**-pointer, and you can't cast from a plain pointer to an **array** pointer.

Some cast-expressions other than the above are accepted by eCv but provoke a warning that verification of the program may not be sound.

## Functions

- Function declarations must adhere to the ANSI format (not the old K&R format) and have explicit return type. **(MISRA 8.2)**
- A function signature of the form **T f()** is interpreted as having no parameters, rather than unspecified parameters; i.e. it is treated as if it were **T f(void)**.
- Variable length argument lists are not supported. **(MISRA 16.1)**
- The body of a function with a non-void return type may not fall through to the end of the function (i.e. it must return a defined value). **(MISRA 16.8)**
- Return statements inside a function with a non-void return type must include a returned value. Return statements inside a function with void return type must not include a returned value. **(MISRA 16.8)**
- A function that writes to non-local variables (other than only to variables directly pointed to by any parameters of non-const pointer types) must declare this in a **writes** clause. See the section on function contracts for more details.

## Arithmetic operations

- The unary-minus operator may not be applied to an operand of unsigned type. **(MISRA 12.9)**
- eCv allows you to use mixed signed/unsigned operands in arithmetic operators and comparisons, but it will both issue a warning and generate verification conditions to ensure that the implicit type conversion(s) involved do not lose information. We recommend adding an explicit type conversion, which will suppress the warning. **(MISRA 10.1)**
- You cannot mix integral and floating operands in arithmetic expressions. This is a consequence of the fact that eCv does not provide implicit conversions from integral to floating-point types. If you want to mix integral and floating operands, you will have to cast the operand having integral type to a suitable floating type. **(MISRA 10.2)**

## Bit operations

- The underlying type of the operands of binary operators **& | ^ << >>** and unary operator **~** must be unsigned. **(MISRA 12.7)**

## Switch statements

- In a switch-statement, the end of one case may not fall through to the next case, i.e. it must end in **break**, **return**, **continue** or **goto**. (MISRA 15.2) However, you can have multiple case labels on a single statement.
- The case-labels of a switch statement must be placed on statements directly within the compound statement that forms the switch body. (MISRA 15.1) They may not be placed on nested statements. Example:

```
switch(i) {
  case 1: /* ok */
  case 2: /* ok */
    ...
    break;
  {
    case 3: /* error, case label is inside another statement */
    default: /* error, case label is inside another statement */
      ...
      break;
  }
  case 4: { /* ok */
    ...
    break;
  }
}
```

- The compound statement that forms the body of a switch statement may not contain declarations unless they are inside a further compound statement. Example:

```
switch(i) {
  int temp1; /* error, declarations not allowed here */
  case 1: {
    int temp2; /* ok, declaration is inside a compound statement */
    ...
  }
  break;
  default:
  int temp3; /* error, declarations not allowed here even in C99 or C++
mode */
  ...
  break;
}
```

## Other restricted constructs

- The operand of **sizeof** may not be a character literal or enumeration constant. The reason is that C and C++ give different results for this.
- You may not use the unary **&** operator to take the address of a field of a union or any sub-part thereof.
- eCv generates a warning if an unsuffixed integer literal is implicitly unsigned. (MISRA 10.6)
- In any **goto** statement, the destination label must be later in the preprocessed source file than the **goto** keyword, and must be in the same block as the goto keyword or in an enclosing block. So backwards jumps and jumps into blocks are not permitted. When processing C99 or C++ source, a goto statement may not jump over any declarations that are in the same block as the destination label.

## Side effects

- The operand of **sizeof()** may not have side effects. (MISRA 12.3)
- Any given variable may be read or written at most once between each pair of consecutive sequence points; except that a variable may be read as part of the calculation of the new value that is to be written to it in an assignment expression. (MISRA 12.2)
- At most one volatile variable may be read or written, and only once, between each pair of consecutive sequence points. (MISRA 12.2)
- The three operands of a conditional expression may not have side effects.

## Unsupported constructs

- The **offsetof(...)** macro is not supported, because it is defined in terms of unsupported type conversions.

## (MISRA 20.6)

- Calls to the `setjmp` and `longjmp` functions/macros are not supported. (MISRA 20.7)

## Additional restrictions enforced by the verifier

- When any of the operators `<` `<=` `>` `>=` is used with pointer operands, they must be array pointers that point into the same array. (MISRA 17.3)
- The value assigned to any entity of an `enum` type must be not lower than the lowest value declared in the corresponding `enum` declaration, and not higher than the highest value declared in the `enum` declaration. One consequence of this is that if you have any static variables of enumeration type, either the enumeration type must have a named member whose value is zero, or the static variable must have an initializer.
- A member of a union may not be read unless the last assignment to the union was done via the same member. So unions cannot be used to convert between types or to pack or unpack data structures. Their sole function must be to represent data of different types at different times or in different situations.
- All variables must be initialized before use. (MISRA 9.1)
- Every array pointer value (including any intermediate values in pointer arithmetic expressions) must address an element that is within the bounds of an array, or be one past the last element of an array, or be null (if it is declared nullable).
- Every access to an array element (whether by indexing or by dereferencing) must be within the bounds of the array.
- Null pointers may not be dereferenced.
- In any type conversion, whether explicit or implicit, the destination type must be able to hold the value being converted without loss of information or a change in the interpretation of the value; except that for conversions from floating-point types to integral types, the foregoing applies only to the integral part after truncation or rounding of the fractional part.
- If the left operand of the right-shift operator has a signed type, its value may not be negative.
- The result of any integral arithmetic operation or left-shift operation must be representable in the result type without any modulo-N wrap round, even if the operands are unsigned.
- When using the integer `/` and `%` operators, the second operand must be positive.
- The right operand of a shift operator must lie between zero and one less than the number of bits in the promoted operand.

## Defining and Using Boolean types

C90 does not provide a Boolean type. C99 provides a boolean type called `_Bool`, and if you `#include "stdbool.h"` then it is also called `bool` and the corresponding literals are called `false` and `true`. C++ provides a Boolean type, calling it `bool` and the corresponding literals `false` and `true`.

**eCv follows the C++ standard** in order to provide stronger typing than C90. Therefore, in order that you can use the Boolean type in your code in a manner that your compiler will accept, you should do one of the following:

(a) Add the following to your standard header:

```
#if !defined(__ECV__) && !defined(__cplusplus)
/* we're neither running under eCv nor compiling as C++, so we need to
define the Boolean type */
#if defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L)
/* we're compiling as C99 so use the definition in stdbool.h */
#include <stdbool.h>
#else
/* compiling as an older version of C */
/* define the Boolean type ourselves in a manner compatible with C99
and C++ */
typedef enum { false = 0, true = 1 } bool;
#endif
#endif
```

and then use the names **bool**, **false** and **true** in your code.

(b) If you are already using some names other than {**bool**, **false**, **true**} to denote Boolean types and values, then do something like the following (this example assumes that you are using **BOOL\_T**, **FALSE** and **TRUE**):

```
#if defined( __ECV__ )
    typedef bool BOOL_T;
    #define FALSE (false)
    #define TRUE (true)
#else
    /* insert your own code here, e.g. the following */
    /* typedef enum { FALSE = 0, TRUE = 1 } BOOL_T; */
#endif
```

## Pointers

Pointers in C and C++ can be troublesome in several ways:

- Zero (i.e. **NULL**) is an allowed value of every pointer type in C and in C++. In critical systems, while we may occasionally want to allow null pointers, for example in the link field of the last element of a linked list, more usually we want to **disallow null pointers**. Verification requires that anywhere we use the **\*** or **[ ]** operator on a pointer, we can be sure that it is not null.
- C and C++ do not distinguish between pointers to single variables and pointers to arrays. So, where we have a parameter or variable of type **T\***, we can't tell whether it is supposed to point to a variable or an array. If it points to a single variable, then we mustn't do pointer arithmetic or indexing on it. The verifier must be able to check this.
- Array parameters in C/C++ are passed as pointers. Aside from the problem that we can't distinguish array pointers from pointers to single variables, we also have the problem that there is no size information contained in an array pointer.
- Anywhere we use pointers to mutable data, there is the possibility of **aliasing**. In other words, there may be more than one pointer to the same data. The verifier needs to take account of the fact that changes to data made through one pointer may affect the values subsequently read through another pointer.

## Nullable and non-nullable pointers

By default, **eCv** assumes when you declare a variable, parameter or function return value as having a pointer type, the value zero (or **NULL**) is not allowed. If you wish to allow this value, you must say so by using the **null** qualifier in the declaration. Here's an example:

```
void foo(int * p, int * null q) { ... }
...
int a = 1;
foo(NULL, &a); /* error, parameter p of foo() is not nullable */
foo(&a, NULL); /* ok, parameter q was declared nullable */
```

## Array pointers

**eCv** requires array pointers to be qualified with the keyword **array**. Here's an example:

```
void copyError(const char * array msg, char * array dst, int dstSize)
{ ... }
```

The presence of the **array** keyword tells **eCv** that the *msg* and *dst* parameters point to arrays rather than single values. If you leave it out, then **eCv** will not allow you to perform indexing or any other sort of pointer arithmetic on those parameters. When you compile the code, **array** becomes a macro with an empty expansion, so your standard C or C++ compiler doesn't notice it.

In this example, we've passed the size of the destination buffer in a separate **int** parameter, so that the code can limit how many characters it writes. However, in writing specifications, we often need to talk about the size of the array that a pointer points to even when we don't have it available in a separate parameter. **eCv** treats an array

pointer like a **struct** comprising three values: the pointer itself, the *lower bound* (i.e. index of the first element), and the *limit* (i.e. one plus the index of the last element). To refer to the lower bound or limit of *dst* we use the syntax *dst.lwb* or *dst.lim* respectively. We also allow *dst.upb* (for *upper bound*), which is defined as  $(dst.lim - 1)$ . Of course, you cannot refer to these fields in code, but you can use them in specification constructs (such as preconditions, invariants, assertions) as much as you like. We call them *ghost* fields because they aren't stored.

For example, let's specify that when the *copyError* function is called, it assumes that *dst* points to an array with at least *dstsize* elements available. Here's how we can specify that:

```
void copyError(const char * array msg, char * array dst, int dstSize)
pre(dst.lim >= dstSize)
{ ... }
```

An array pointer in C/C++ may only address the first element of an array, or one-past-the-last element, or any element in between. If *p* addresses the first element of an array of *N* elements, then  $p.lwb == 0$  and  $p.lim == N$ . If it addresses one-past-the-last element, then  $p.lwb = -N$  and  $p.lim = 0$ . So *p* has implicit invariant  $p.lwb \leq 0 \ \&\& \ p.lim \geq 0$ .

Within the body of *copyError*, *eCv* will attempt to prove that all accesses to *msg* and *dst* are in bounds. For example, the expression *dst[i]* has precondition  $dst.lwb \leq i \ \&\& \ i < dst.lim$ . Also, wherever *copyError* is called, *eCv* will attempt to prove that the precondition holds. So anywhere that buffer overflow is possible, there will be a corresponding a failed proof. If all the proofs succeed, and provided that no function with a precondition is ever called by external unproven code, we know that buffer overflow will not occur.

As with plain pointers, *eCv* assumes that array pointers may not take the value zero (i.e. NULL) unless you qualify them with the **null** keyword. When you qualify a pointer declaration with both **array** and **null**, it does not matter which order you place the qualifiers in; however we suggest using **array null** rather than the other way round.

When declaring a function parameter that accepts an array of elements of some type **T**, both C and C++ allow the type to be declared as **T[ ]** as an alternative to **T\***, with the same meaning. *eCv* also allows **T[ ]** as the type of a parameter to be qualified with **null**; so declaring a parameter with type **T[ ] null** has the same meaning as declaring it with type **T\* array null**.

## The not\_null operator

Sometimes you may have an expression that has a nullable pointer type, but you know that its value is not null and you wish to use it in a context that requires a non-nullable pointer. You can do this using the **not\_null** construct.

The expression **not\_null(pointer-expression)** yields the value of *pointer-expression* (which must have nullable pointer or nullable array pointer type) as a non-nullable pointer or non-nullable array pointer, asserting that it is not null. This is equivalent to a cast from the nullable type to the non-nullable type, but avoids your C compiler or other static checker perhaps issuing a warning about a redundant cast. It also makes it clear that you are just asserting non-nullness, rather than trying to do a more general type conversion. *eCv* will generate a verification condition that *pointer-expression* is not null.

If you use an expression whose type is a nullable pointer type in a context where a non-null pointer is required, *eCv* will assume an implicit **not\_null(...)** operation around the expression, and warn you that it has done so.

## Pointers as function parameters

Parameters of pointer type are often used to pass values results between functions and their callers. When you declare a parameter of a non-const pointer type, *eCv* normally assumes that when you call the function, it both reads and writes the value that is pointed to. Therefore, if you take the address of a variable and then pass that address to a function, you must initialize that variable first. The function is permitted but not obliged to update the variable by writing through the pointer.

You can change this behaviour by flagging a parameter with the keyword **out** at the start of its declaration. This indicates to *eCv* that the pointer parameter is used only to pass a value back from the function. When you take the address of a variable and then pass that address as the actual value of an **out** parameter, you do not need to initialize the variable first. However, a function is obliged to write through all its pointer parameters that have been flagged as **out** parameters, except when the parameter is a nullable pointer and the actual parameter is null.



Here is an example:

```
void foo(int *p, out int *q, out int *r) {
    p += 1;      /* ok */
    q += 1;      /* error, out-parameter q used before it has been
initialized */
    return;      /* error, function must initialize out-parameter r before
returning */
}

...
int a, b, c, d;
foo(&a, &b, &c); /* error, 'a' has not been initialised */
foo(&b, &c, &d); /* ok, 'b' was initialized by the previous call to foo
*/
```

Only parameters of non-const pointer type (including pointer types flagged **array** and/or **null**) may be flagged **out**.

## Assertions and assert.h

If you use assertions in your program and **#include "assert.h"** in your source file, we suggest that you make this inclusion conditional like this:

```
/* #include "ecv.h " somewhere before the following */
#ifdef __ECV__
#undef assert          /* remove eCv's definition */
#include "assert.h"
#endif
```

eCv will then treat your assertions as eCv assertion statements (see [assert](#)) and try to prove they are true.

If you do not make the inclusion conditional, then the definition of **assert** in file *assert.h* may override the definition of **assert** in file *ecv.h*. If the **DEBUG** macro is defined when you run eCv, then your assertions will typically be expanded to code that performs I/O if the assertion fails. eCv will try to verify this code and will report errors if it modifies variables that were not declared in the writes-clause of the current function.

## Extensions to the C and C++ languages supported by eCv

eCv supports the following extensions to the C and C++ languages as defined in the ISO standard documents:

- Comments beginning with **//** and ending at end-of-line are permitted even in C90 mode. Comments of this form are part of the C99 and C++ language standards.
- Integer literals in binary format beginning with **0b** or **0B** are supported. The type of a binary integer literal is determined using the same rules as for hexadecimal integer literals. Note that eCv generates a warning if an un-suffixed integer literal is implicitly **unsigned**.
- The address-placement syntax for variables supported by many embedded C compilers are supported.

Many C and C++ compilers support additional extensions. Where these extensions are implemented using additional keywords (possibly followed by a bracketed argument list), it is usually possible to define these keywords as null macros when processing with eCv, so that eCv does not see those extensions and can process the source file. File **eCv.h** already contains a number of these macro definitions.

## Common error messages and what to do about them

Sometimes you may find that your C compiler accepts your source code, but eCv does not. Here are some of the most common eCv error messages that you may see under these conditions, and what they mean.

**Error! Incorrect syntax at ...** Assuming that a regular C compiler accepts your source code, then this means one of the following:

- You have used an eCv reserved word as an identifier. You must either rename the identifier, or rename the keyword. To make this sort of error easier to spot visually, we recommend that you use an editor that supports syntax highlighting, and add all the eCv keywords to its keyword list for C.
- You have declared the types of function parameters using the old K&R syntax. Use the ANSI/ISO syntax instead.
- (when this error occurs at the keyword **case** or **default**) You have written a **switch** statement that does not conform to the format supported by eCv. Case labels must appear directly in the body of the switch statement, not nested inside further compound statements.
- You have included a declaration directly in the body of a **switch** statement.
- You have followed the colon character in a conditional expression directly by a minus-sign. Resolve this by inserting a space character after the colon.
- You are using a C99 or C++ construct that is not supported by eCv.

Error! Incorrect syntax at token '`_ecv_pre`'. This can occur if you have extra brackets around the function declarator, e.g.

```
int (foo(int arg)) pre arg > 0 { ... }
```

The fix is either to remove the extra brackets:

```
int foo(int arg) pre arg > 0 { ... }
```

or to include the precondition (and any other specifications) inside the brackets:

```
int (foo(int arg) pre arg > 0 ) { ... }
```

Error! Cannot find declaration of class '`size_t`'. This will occur if your program uses the C `sizeof` keyword, but you haven't included any standard header file that defines `size_t`. eCv needs this definition so that it knows exactly what type a `sizeof` expression yields. The fix is to **#include `stddef.h`** (or some other header that correctly defines `size_t`) in your source file.

Error! Cannot find declaration of class '`ptrdiff_t`'. This will occur if your program subtracts one pointer from another, but you haven't included any standard header file that defines `ptrdiff_t`. eCv needs this definition so that it knows exactly what type a pointer difference expression yields. The fix is to **#include `stddef.h`** (or some other header that correctly defines `ptrdiff_t`) in your source file.

Error! No binary operator '`==`' defined between types '`T*`' and '`int`' (for some T, where the integer operand is literal zero, and where `==` can also be `!=`). This normally indicates that the variable or parameter of type `T*` should have been declared nullable, i.e. **`T* null`** instead of just `T*`.

Error! No binary operator '`+`' defined between types '`double`' and '`int`' (for '`+`' or some other arithmetic operator). eCv does not support mixed-mode arithmetic. Cast the integral operand to **`double`** or some other suitable floating-point type.

Error! No globals or current class members can match '`result`'. This occurs when you use an expression like `0..result` in a specification and you are using your compiler's own preprocessor. The reason is that the preprocessor treats `0..result` as a single *pp-number* token, so it doesn't recognise `result` as a macro to be expanded. To fix this, put a space between `0` and `..` or between `..` and `result`, or use `(0)` instead of `0`, or use `(result)` instead of `result`.

## Line and column coordinates in error and warning messages

If your code uses either an eCv specification macro such as `pre` or one of your own macros, and eCv detects errors either in the code resulting from the macro expansion or in code that follows the macro and its arguments on the same line, then the line/column coordinates that eCv provides in the error message may to be incorrect. This is because eCv sees the code after macro expansion.

If you put specification macros and their actual parameters all on the same line, and you do not use any other macros on that line, then the line number should be correct. If you are using the eCv preprocessor or your compiler's preprocessor preserves white space, and you use the standard eCv specification macro names, then the column numbers should also be correct.

If you use keywords starting with `_ecv_` directly, then the column number of any message that refers to a construct after that keyword but on the same line will be incorrect. This is because when eCv calculates the column number in the original source file, it assumes that any keyword starting with `_ecv_` was generated by expanding the corresponding keyword without the `_ecv_` prefix.

If you put macro parameters on more than one line, then your preprocessor may expand the macro such that everything is on one line. In that case, any error messages relating to code or specifications in the macro parameters will have a line number that relates to the source line on which the macro name appeared; and the column number may be somewhat higher than the number of columns in that line. For this reason, we suggest that, when declaring specifications, you keep each specification and its arguments on the same line, wherever possible. For example, don't use:

```
pre(a >= 0;
a < 10)
```

Instead, use either:

```
pre(a >= 0; a < 10)
```

or:

```
pre(a >= 0)
pre(a < 10)
```

## Suppressing warning messages

You can suppress a warning message (but not an error message) like this:

```
#pragma ECV ignore_warning "warning-text"
```

This will cause a single instance of a warning message that includes the specified text to be ignored. Instead, an information message will be generated that the warning has been ignored. You can list multiple comma-separated warning messages in a single pragma. Here is an example:

```
void f(int a, unsigned int b) {
#pragma ECV ignore_warning "Signed/unsigned mismatch", "Implicit
conversion of expression 'a + b'"
    int c = a + b;
}
```

Optionally, you may give an integer line offset before the first message, to specify the number of lines to be skipped before the warning message is expected. Here is another way of expressing the previous example:

```
#pragma ECV ignore_warning 2 "Signed/unsigned mismatch"
#pragma ECV ignore_warning 1 "Implicit conversion of expression 'a + b'"
void f(int a, unsigned int b) {
    int c = a + b;
}
```

eCv will warn you if you use the ignore-warning pragma but no matching warning is generated.

 [TOC](#)

# Verifying your source

## Ensuring that verification is valid for your target environment

You must make sure that eCv and your compiler interpret the source code in the same way. In particular:

- eCv assumes (and does not check) that header files are always parsed in the same `#define` environment, i.e. when parsing a set of files, it need only consider one instance of each header file. So don't use `#ifdef`, `#ifndef` or `#if` inside a header file to make it do different things depending on which file is including it.
- eCv only sees your program in a state in which `__ECV__` is defined. So don't make parts of your program conditional on whether `__ECV__` is defined, unless you really know what you are doing!
- eCv passes the search paths you configure in your project and in eCv's global compiler settings to the preprocessor; project-specific paths first, then compiler-specific ones. Make sure that when you actually compile the code, the compiler uses exactly the same search path order, in case there are files with the same name in different places on the search path.
- Some compilers define additional macros automatically, and most allow additional macro definitions to be provided on the compiler command line. Add these additional macro definitions to the project settings, so that eCv sees the same set of macro definitions as the compiler (plus the definition of `__ECV__`, which is added automatically when you run eCv).
- The meaning of a C program, and therefore the verification conditions that eCv generates, depends on various compiler/platform details, such as the sizes of the various integral types and the behaviour of integer division. You must ensure that the details you have configured in the eCv Project Manager for the C/C++ compiler you have told it you are using match the actual compiler you use. If you intend to compile for more than one platform, you must run eCv verification separately for each platform, unless you know that the configuration parameters are the same for all of them. You can set up multiple configurations in a single project to allow for using different compilers.

**Note:** in order to determine the maximum and minimum values that can be stored in variables of the integral types of C, eCv assumes that your compiler/platform uses two's-complement representation of signed integral types, and plain binary representation of unsigned integral types. If this is not the case, then eCv may give incorrect results.

## Which preprocessor?

When you verify your source files, you can use either eCv or the target compiler to preprocess them, depending on how you have configured the chosen C or C++ compiler in the Project Manager. If you choose to let eCv preprocess them, you can use the standard header files supplied with eCv, or you can attempt to use the standard header files supplied with your compiler. Here are some guidelines to help you choose:

- **The easiest and best approach is usually to use eCv to preprocess the source and use the header files supplied with eCv.** To achieve this, configure the eCv include-file path in the Project Manager. Make sure this path includes the directory containing the standard header files, but not the directory containing your compiler's standard header files. If you need to have both directories in the include path, put the eCv header files directory earlier in the path.
- If you wish to use your own compiler to preprocess the source, you must configure the Project Manager to do this for your chosen compiler (use **Options** → **C/C++ Compilers**).
- eCv does not perform MISRA checks relating to use of the preprocessor, when you are not using the eCv preprocessor.
- Although it is possible to use eCv to preprocess files but pick up the header files supplied with your compiler, this may be problematic for two reasons. First, compiler-supplied header files may use language extensions that cause eCv to fail to parse the file (although it may be possible to define the associated keywords as null

macros so that eCv ignores them). Second, header files supplied with some compilers (notably gcc) depend on a large number of macros that the preprocessor normally defines, and you need to accurately replicate these macro definitions in order to ensure the correct behaviour.

- If you use the eCv standard header files, then only definitions in the C standard library will be available. If your software depends on non-standard definitions in the header files supplied with your compiler, these will not be available when running eCv.
- The header files supplied with eCv include function contracts. If you do not use these header files then the contracts will not be available. You may therefore need to add contracts to your program for standard library functions, in order that code that calls these functions can be verified.
- If you use your own compiler to preprocess the source, then you can use the platform check program supplied with eCv to check that the type sizes you have configured in the eCv compiler parameters match the values in the compiler-supplied **limits.h** file.
- If your compiler's preprocessor does not follow the ISO standard, or if you are using features whose behaviour is undefined (e.g. multiple # and/or ## tokens in a single macro body such that the meaning depends on the order in which they are evaluated), then the behaviour of the eCv preprocessor and your compiler's preprocessor may differ, so that the code verified by eCv is not the same as the code seen by your compiler.

## Verification

When you have resolved any errors (and, optionally, warnings) produced by eCv when you Check your program, press the green-tick button on the toolbar to verify your source. Expect a lot of verification warnings if you haven't yet annotated your source code with specifications. Resolve verification errors by adding preconditions and other specifications.

The sort of specifications you need to add depend on what you want to verify:

- For functions that you want to verify, you need to write at least preconditions, writes-clauses and loop invariants. You may also need to write postconditions, if they are called by other functions that you want to verify.
- For functions (including library functions) that you do not want to verify at the present stage, but which are called from functions that you **do want** to verify, you need to write preconditions, writes-clauses, and sometimes postconditions.
- You may also want to write assertions and postconditions to describe properties you expect to hold.
- You need only write loop variants where you wish to prove that a loop terminates.

You can run verification on individual files by right-clicking on the file in the Project Manager window and selecting **Verify**.

Note that you don't always have to run a Check before running a Verify, since Verify will start by checking anyway.

## Constructs that are unverifiable or compromise verification

eCv is unable to verify code that contains certain constructs as detailed below. Where the integrity or verification results may be compromised, eCv will generally issue a warning message.

### Casts between pointer types

eCv assumes strong typing, therefore it cannot verify code that contains casts or implicit conversions between pointers to different types. The exception is that conversions to **void\*** do not make code unverifiable.

### Casting away const

eCv assumes that variables annotated by **const** are immutable. Casting away **const** violates this assumption.

However, if you cast away **const** so that you can pass a pointer to a function that takes a non-**const** parameter, and the function does not actually write through that parameter, validity of eCv verification is not affected.

## Casting away volatile

eCv tracks the value of non-volatile variables, but not the values of volatile variables. If you cast a volatile-qualified pointer to a non-volatile-qualified pointer, then the variable tracking performed by eCv will not function correctly, and the integrity of verification is compromised.

## Calls to memcpy and memset

eCv can only reason about calls to the standard library function *memcpy* when at least one of the following is true:

- The third parameter is zero;
- The first two operands were originally pointers of type *T\** (prior to being converted to **void\***) and the third parameter is equal to *sizeof(T)*;
- The first two operands were originally pointers of type *T\* array* (prior to being converted to **void\***), they each point to the start of an array, and the third parameter is an exact multiple of *sizeof(T)*.

eCv can only reason about calls to the standard library function *memset* when at least one of the following is true:

- The third parameter is zero;
- The first operand was originally a pointer of type *T\** (prior to being converted to **void\***), the second parameter is zero, and the third parameter is equal to *sizeof(T)*;
- The first operand was originally a pointer of type *T\* array* (prior to being converted to **void\***) and it points to the start of an array, the second parameter is zero, and the third parameter is an exact multiple of *sizeof(T)*.

In other cases, the validity of verification results is not affected, but expect eCv to find some verification conditions unprovable.

**Note that using *memset* to initialize objects that include pointers and/or floating-point fields is neither portable nor verifiable by eCv.** This is because the bit patterns used to represent null pointers and floating-point zeros are implementation-defined, not necessarily all zeros.

## Suppressing verification of included files

Sometimes you may wish to suppress verification of certain *#include* files, for example vendor-supplied include files. eCv maintains a list of the names (with full paths) that you do not wish to verify. There are two ways of adding file names to this list:

- By putting the following line in files that you do not wish to be verified:

```
#pragma ECV noverify
```

This adds the current file to the do-not-verify list. For all **#include** directives after the first occurrence of this pragma in a file, those included files will also be added to the do-not-verify list.

- By putting the following in a file that contains an **#include** directive for the file you do not wish to verify, prior to that **#include** directive:

```
#pragma ECV noverifyincludefiles
#include "file1.h"
#include "file2.h"
#pragma ECV verifyincludefiles
```

In this example, file1.h and file2.h are added to the do-not-verify list, as are any files that they **#include**.

Note: once a file has been added to the do-not verify list, it will never be verified, even if it is also included from  
Page 22 of 62

another context in which there is no pragma to suppress verification. The Verification Report produced by EscherTool includes a list of files for which verification was suppressed.

[▲ TOC](#)

**eCv Manual, Version 7.0, February 2017.**

**© 2017 Escher Technologies Limited. All rights reserved.**

# eCv specifications

## General Notes

All expressions within eCv specification constructs must have NO side effects. So the following are not permitted in specifications:

- assignment expressions
- operators ++ and -- (both prefix and postfix)
- assigning operators (e.g. +=)
- calls to functions that have side-effects, i.e. functions with explicit or implicit non-empty writes-clauses
- reading or writing volatile variables

Where a specification macro takes an expression list, the expressions must be separated by a semicolon, not by a comma. This is because macros in C90 and C++ must have a fixed number of arguments.

Example:

```
pre(n >= 0; n <= 10)      /* correct */
pre(n >= 0) pre (n <= 10) /* correct */
pre(n >= 0, n <= 10)     /* incorrect */
```

Within specifications, you may use ghost functions and ghost members. For example, if *myArray* is a parameter of array type, then *myArray.lwb* gives the lowest valid index (usually 0), and *myArray.upb* gives the highest valid index (usually one less than the number of elements in the array).

Similarly, if *s* is a parameter of type **char \* array**, then *isNullTerminated(s)* expresses the condition that there is a valid non-negative index into *s* such that the corresponding element of *s* is the null character. See the list of ghost members and predefined ghost functions later in this document.

## Type constraints

You can declare a constrained type by using an **invariant** clause within a **typedef** declaration. Here is an example:

```
typedef int invariant(value in 0..100) percent;
```

The argument of **invariant** is an expression that constrains on the values of this type, using the keyword **value** to refer to any such value. The constraint may refer to constants but not to variables.

The semantics of constrained types are as follows, in which the term *underlying type* refers to the type that precedes the **invariant** clause:

- A value of a constrained type can be converted implicitly to the underlying type. In particular, this conversion occurs whenever a value of a constrained type is subject to the "usual arithmetic conversions".
- A value of the underlying type can be converted implicitly to the constrained type. However, eCv generates a verification condition to ensure that the value is permitted by the constraint.
- If a static variable or a field or element of a static variable has a constrained type, it must either have an initializer or the constraint must allow the default static-initialized value of the underlying type.
- A pointer to a constrained type may not be cast to or from any other type, other than another type that is a synonym for it. In the above example, a *percent\** may not be cast to or from a *int\**. However, if we also declare:



```
typedef percent percent2;
```

then *percent2* is a synonym for *percent* and we may convert between *percent\** and *percent2\** in either direction, explicitly or implicitly. If instead we declare:

```
typedef int invariant(value in 0..100) percent2;
```

then *percent2* and *percent* are treated as different types, and there is no conversion from *percent\** to *percent2\** or vice versa. Likewise if we declare:

```
typedef percent invariant true percent3;
```

then *percent3* is not a synonym for *percent* and there is no conversion between *percent\** and *percent3\**.

Note that you can avoid a lot of potential aliasing problems by using constrained types, since a pointer to a constrained type cannot be aliased to a pointer to any other type that is not a synonym for that type.

## Function contracts

Function contracts are placed after the function parameter list, but before the opening brace "{" of the body if it is a function definition, or before the terminating semicolon if it is a function prototype.

You may declare any or all of writes-clauses, preconditions, recursion variants, returns-expressions and postconditions, but they **must** be declared in that order. The syntax of these is:

- writes(writes-expression-list)
- pre(expression-list)
- decrease(expression-list)
- returns(expression)
- post(expression-list)

You may declare more than one writes-clause, precondition, postcondition or variant, which is equivalent to declaring a single, longer list. For example:

```
pre (n >= 0)  
pre (n <= 10)
```

means the same as

```
pre (n >= 0; n <= 10)
```

Keep the arguments of each instance of a specification on a single line as far as possible, to avoid getting misleading line numbers in error and warning messages.

### writes(writes-expression-list)

The writes-clause specifies what nonlocal variables the function writes. A writes-clause may contain the following elements, separated by semicolon:

- Normal lvalue expressions of non-volatile types, indicating that those expressions may be written to.
- Expressions of the form **some** (*type-name*), indicating that any nonlocal values of the designated type may be written. This form should only be used when the expressions that are written cannot be individually enumerated. For example, a function that iterates through a list updating every node in the list would specify **writes** (**some** (*node\_t*)) where *node\_t* is the type of the list node. [Note: the semantics of **some** are not fully implemented in the initial release of eCv, therefore programs using **some** may not be completely verifiable.]
- The special form **volatile**, indicating that the function writes and/or reads unspecified volatile variables. It is

not necessary to list the variables individually.

If you don't provide a writes-clause, then a default one is constructed. The default will be such that for any parameter declaration of the form  $T *p$  or  $T *array\ p$  where  $T$  is not qualified by **const** or **volatile**, the function is assumed to write to  $*p$ . If your function writes to any other nonlocal variables (for example, static variables), then you must declare **all** the nonlocal variables it writes to in a writes-clause.

If you have a function with a parameter of the form  $T *p$  that doesn't write to  $*p$ , then if the function is under your control, we recommend you add the **const** qualifier. If this is not possible (for example, it is a third-party library function that you cannot change), use an explicit writes-clause to prevent eCv from assuming that the function writes to  $*p$ . You can use the form `writes ()` to indicate that a function writes no nonlocal variables at all.

## **pre(expression-list)**

Preconditions describe the constraints that the caller must satisfy when calling the function. eCv requires that all function preconditions be declared explicitly.

## **decrease(expression-list)**

Recursion variants are only used in recursive function specifications, and allow eCv to prove that the recursion terminates. See *Perfect Developer Language Reference Manual* for details. Note that a function may have a recursive specification even if its implementation is not recursive.

## **returns(expression)**

Returns-specifications describe the value that a function returns. They are not allowed when the function return type is **void**). The specification `returns (e)` is equivalent to `post(result == e)` with the important exception that the expression inside `returns (...)` is permitted to make recursive calls to the function being specified, whereas expressions inside `post(...)` are not. This allows you to write a recursive specification for the return value of a function, even when the return value is computed by iteration instead of recursion.

## **post(expression-list)**

Postconditions describe conditions that the function guarantees hold when it returns. In postconditions, you may use the keyword **result** to refer to the value returned by the function. A non-void function may have both a returns-specification and a postcondition - for example, the postcondition might assert additional properties of the function result and/or describe side-effects of the function.

## **Where to put function contracts**

If a function has **extern** linkage (i.e. it is not declared **static**), then you will normally define the function in a `.c` file and provide a prototype for it in a `.h` file. You should do the following:

- Declare the writes-clause (if applicable) and any preconditions and postconditions for the function in the prototype. This means that when you verify another `.c` file that `#includes` the file containing the prototype, the specifications are available.
- **#include** the `.h` file in the `.c` file (this is normal practice anyway, so that the prototype gets checked against the definition). If you don't do this, then the preconditions and postconditions will not be available when the function itself is verified.
- If the specification is recursive (i.e. the returns-specification calls the function recursively), you must declare a variant in the prototype. If only the function body is recursive, you may declare a variant in either the prototype or the definition, but not in both.

eCv will give an error message if, when processing a `.c` file and all the other files that it `#includes`, it finds a function definition with specifications and there is a prototype for that function in a different file.

If you need to provide specifications for functions declared in a third-party header file, but you do not wish to add specifications or the **array** or **null** keywords to that header file, you can declare prototypes for the same functions

with added specifications, **array** and **null** keywords in your own header file. Each such function declaration should be prefixed with the **spec** keyword to tell eCv that this overrides the other one. eCv will nevertheless check that each pair of declarations is compatible, ignoring the missing **array** and **null** keywords in the third-party file.

eCv provides a number of header files corresponding to the standard C header files for this purpose. For example, file **ecv\_string.h** provides specifications for functions in the standard header file **string.h**. So if your program includes **string.h**, you must also include **ecv\_string.h**. Note that including **ecv\_string.h** alone is not sufficient to make the declarations in **string.h** available. This is to ensure that your compiler sees its own versions of the declarations. You do not need to make the inclusion of **ecv\_string.h** and similar files conditional on **#ifdef \_\_ECV\_\_** because this is already done inside the file.

If a function is local to a single file (i.e. **static** linkage), then you must declare the specifications in the prototype, if it has one. If the function has no prototype, then declare the specifications in the function definition.

Specifications for C++ class member functions whose implementation is defined outside the class declaration must be attached to the declaration of the member function declaration within the class declaration.

## Loop specifications

Loop specifications must be placed between the loop header and the loop body, like this:

```
for (...)
loop-specifications
{
    statements
}

while (...)
loop-specifications
{
    statements
}

do
loop-specifications
{
    statements
} while (...)
```

You may declare any or all of writes-clauses, loop invariants, and loop variants, but they must be declared in that order. The syntax of these is:

- writes(writes-expression-list)
- keep(expression-list)
- decrease(expression-list)

You may declare more than one writes-clause, loop invariant, or loop variant, which is equivalent to declaring a single, longer list. For example:

```
keep(n >= 0)
keep(n <= 10)
```

means the same as

```
keep(n >= 0; n <= 10)
```

Keep the arguments of each instance of a specification keyword on a single line as far as possible, to avoid getting misleading line numbers in error and warning messages.

## Loop writes clause

eCv needs to know what variables a loop modifies. If a loop modifies only the local variables of the function it occurs

in, and modifies them directly rather than via a pointer, then eCv can generally work this out for itself. However, if a loop modifies anything else, or anything via a pointer or array pointer, then a writes-clause for the loop must be given explicitly. For example, consider the following:

```
void setArray(int * array a, size_t size, int k)
{ size_t i;
  for (i = 0; i != size; ++i) {
    a[i] = k;
  }
}
```

The loop modifies elements of the array *a* which is not a local variable, therefore a writes-clause is needed. The loop modifies all elements of the array, therefore the appropriate expression is *a.all*:

```
void setArray(int * array a, size_t size, int k)
{ size_t i;
  for (i = 0; i != size; ++i)
    writes(a.all)
    {
      a[i] = k;
    }
}
```

Note: if a loop modifies any elements of a local array, and no writes-clause is given, then it will be assumed that the loop modifies all elements of the array. You can use a loop invariant to describe elements of the array that do not change.

## Loop invariant

eCv requires you to write a *loop invariant* for every loop. A loop invariant is a Boolean expression that depends on all the variables modified by the loop, and is true when the loop is first entered, at the start and end of each iteration of the loop, and when the loop terminates. Typically, it comprises two parts.

The most important part of the loop invariant is a generalization of the state that the loop is intended to achieve. It needs to be written so that it is easy to establish this part of the invariant before the loop starts, by suitable initialization of variables; yet it becomes exactly the desired state when the loop terminates. We'll refer to the desired state when the loop terminates as the *loop postcondition*.

For example, suppose we have an array *a* of integers, and we want to set every element in that array to *k*. Here's a function to do that:

```
void setArray(int * array a, size_t size, int k)
pre(a.lwb == 0)
pre(a.lim == size)
post(forall j in 0..(size - 1):- a[j] == k)
{ size_t i;
  for (i = 0; i != size; ++i)
    writes(a.all)
    {
      a[i] = k;
    }
}
```

The first precondition says that *a* is a regular pointer to the start of an array. The second one says that the number of elements is given by *size*.

The postcondition says that when the function returns, for all indices *j* in the range 0 to  $(size - 1)$ , *a[j]* is equal to *k*.

In this case, the loop comprises the entire body of the function, so the loop postcondition is the same as the function postcondition. In fact, loop postconditions are very frequently **forall** expressions, especially for loops that iterate over an array.

We need to generalize the **forall** expression in the postcondition here so that it is true at the start of every iteration of the loop. The state when we are about to commence the *i*<sup>th</sup> iteration will be that we've already set all elements from zero to *i-1* (inclusive) to *k*, but not the elements from *i* onwards. We can express this with a slight modification

to the **forall** expression, like this:

```
forall j in 0..(i - 1) :- a[j] == k)
```

The upper bound of the **forall** has been changed from  $(size - 1)$  in the postcondition to  $(i - 1)$  here. This gives us exactly what we need for the loop invariant:

- The loop initialization sets  $i$  to zero, so the initial bounds of the **forall** are  $0 \dots -1$  (that is, from zero up to minus one). This is an empty range (because  $-1$  is less than  $0$ ), and a **forall** over an empty range is **true**. So the loop invariant is established at the start of the first iteration.
- During each iteration, we set the  $i$ th element to  $k$  **and** we increment  $i$ . So the invariant is preserved. For example, after the first iteration, the **forall** has range  $0..0$  so it just tells us that  $a[0] == k$ , which is exactly right. After the second iteration, the range is  $0..1$  so it says that  $a[0]$  and  $a[1]$  have value  $k$ , which again is exactly right.
- The loop terminates when  $i == size$ , because we wrote the while-condition as  $i != size$ . If we replace  $i$  by  $size$  in the invariant, we get exactly the desired postcondition.

eCv expects the loop invariant to be written between the loop header and the body, like this:

```
void setArray(int * array a, size_t size, int k)
pre(a.lwb == 0)
pre(a.lim == size)
post(forall j in 0..(i - 1) :- a[j] == k)
{ size_t i;
  for (i = 0; i != size; ++i)
    writes(a.all)
    keep(forall j in 0..(i - 1) :- a[j] == k)
    { a[i] = k;
    }
}
```

eCv uses the keyword **keep** to introduce the invariant, because we're **keeping** the invariant true.

Unfortunately, this is not yet enough to allow eCv to verify the loop. Using the above, eCv reports that it is unable to prove that  $a[j]$  is in-bounds in the loop invariant, or that  $a[i]$  is in-bounds in the loop body.

So, we need another component of the loop invariant, to ensure that these accesses are always in bounds. To make sure that  $a[i]$  is in bounds, we need to constrain  $i$  to be in the range  $0..(size - 1)$  at that point. But we can't constrain  $i$  to this range in the invariant, since when the loop terminates,  $i$  ends up with the value  $size$ , which is just outside this range. Instead, we constrain  $i$  to the range  $0..size$ . The body is not executed when  $i == size$ , so that constraint is sufficient to guarantee that  $a[i]$  is in bounds in the body. The code then looks like this:

```
void setArray(int * array a, size_t size, int k)
pre(a.lwb == 0)
pre(a.lim == size)
post(forall j in 0..(i - 1) :- a[j] == k)
{ size_t i;
  for (i = 0; i != size; ++i)
    writes(a.all)
    keep(i in 0..size)
    keep(forall j in 0..(i - 1) :- a[j] == k)
    { a[i] = k;
    }
}
```

Previously, eCv reported that it was unable to prove that  $a[j]$  was in bounds in the original **keep** clause. Putting the new **keep** clause before the original one allows eCv to assume that  $i$  is in  $0..size$  in the second **keep** clause. That is enough for it to prove that  $a[j]$  is in bounds, because  $j$  takes values from  $0$  to  $i - 1$ .

With the above loop specifications, eCv is able to prove that this function meets its specification, *if* the loop terminates. To prove that it terminates, we'll need to provide a *loop variant*.

Note: if the while-part of a for-loop or while-loop has side effects, then the loop invariant refers to the state **before** those side-effects take place.

## Proving loop termination

The easiest way to prove that a loop that is designed to terminate actually **does** terminate is to use a *loop variant*. A loop variant in its simplest form is a single expression that depends on variables changed by the loop. It has the following properties:

- Its type has a defined lower bound, a finite number of values, and a total ordering on those values;
- Its value decreases on every iteration of the loop.

If we can define such a variant, then we know that the loop terminates, because from any starting value the lower bound must be reached after a finite number of iterations – after which it can't decrease any more.

To ensure that the first of these properties is met, `eCv` only allows loop variant expressions to have integer, Boolean, or enumeration type. For integer variants, `eCv` assumes a lower bound of zero and will need to prove that such a variant is never negative. For Boolean expressions, the lower bound is **false**, and **true** is taken to be greater than **false**. For expressions of an enumeration type, the lower bound is the lowest enumeration constant defined for that type.

In order to show that our example loop terminates, we need to add a loop variant in the form of a **decrease** clause. In this case, it is simple to insert a loop variant based on the loop counter:

```
void setArray(int * array a, size_t size, int k)
pre(a.lwb == 0)
pre(a.lim == size)
post(forall j in 0..(i - 1) :- a[j] == k)
{ size_t i;
  for (i = 0; i != size; ++i)
    writes(a.all)
    keep(i in 0..size)
    keep(forall j in 0..(i - 1) :- a[j] == k)
    decrease(size - i)
    { a[i] = k;
    }
}
```

The expression  $size - i$  meets the needs of a loop variant in `eCv` because:

- It has one of the allowed types (i.e. integer);
- It is always  $\geq 0$  inside the loop body (actually, it is  $\geq 0$  immediately after the loop terminates too, although `eCv` doesn't require that);
- It decreases on every iteration (because  $i$  increases while  $size$  remains constant).

`eCv` will try to prove that  $size - i$  is never negative, and that  $size - i$  decreases from one iteration to the next. The first of these is easily proven from the invariant  $i$  in  $0..size$ . The second is easily proved because the loop increments  $i$  at the end of each iteration but leaves  $size$  alone.

For a for-loop whose header increments a loop counter from a starting value to a final value, we can always use a loop variant of the form  $final\_value - loop\_counter$ , provided the loop body doesn't change  $loop\_counter$  or  $final\_value$ .

For some loops, each iteration may make progress towards termination in one of several ways. For example, you could write a single loop that iterates over the elements of a two-dimensional array. Each iteration might move on to the next element in the current row, or advance to the next row if it has finished with the current one. In cases like this, defining a single variant expression for the loop can be awkward. So `eCv` allows you to provide a list of variant expressions. Each iteration of the loop must decrease at least one of its elements in the list, and may not increase an element unless an element earlier in the list decreases. So it must either decrease the first element, or keep the first element the same and decrease the second element, or keep the first two elements the same and decrease the third; and so on.

Note: if the while-part of a for-loop or while-loop has side effects, then the loop variant is computed **after** those side-effects take place.

# Ghost declarations

Sometimes it is easier to write a specification if you declare additional constants, variables and function prototypes (with associated contracts) that are referred to only in specifications. Such a declaration is called a *ghost* declaration. To tell eCv that a declaration is ghost, and to avoid your compiler generating code for ghost declarations, such a declaration is enclosed in the **ghost(...)** macro. Here are some example ghost declarations:

```
ghost(  
    int max3(int a, int b, int c)  
    post(result >= a && result >= b && result >= c && (result == a || result  
    == b || result == c));  
)  
  
ghost(const int maxReading = 1000;)
```

There are some special rules for ghost declarations, as follows:

- A ghost function declaration may not have a body. It must be a prototype, and to be useful it must have a contract specification.
- Ghost declarations may use the predefined ghost type **integer**, which is an integral type with no bounds; and the predefined ghost collection types **\_ecv\_set<T>**, **\_ecv\_bag<T>** and **\_ecv\_map<T1, T2>** (see the Library Reference section of the *Perfect Developer Language Reference Manual* for details of these collection types)
- Ghost functions may have array return types, for example **int[ ]**

You may refer to ghost declarations in specification contexts and in other ghost declarations, but not in code. The final semicolon at the end of the ghost declaration (before the closing parenthesis that completes the **ghost** macro invocation) is optional.

Sometimes, for specification purposes it is useful to pass additional ghost parameters to non-ghost functions or constructors. This can be done by declaring a ghost parameter list immediately after the main parameter list, like this:

```
void foo(int a, bool b) ghost(int c, int d)
```

When such a function is called, actual ghost parameters must be provided using a similar syntax:

```
foo(aa, bb) ghost(cc, dd);
```

[TOC](#)

eCv Manual, Version 7.0, February 2017.  
© 2017 Escher Technologies Limited. All rights reserved.

# Additional eCv constructs

## Additional eCv declarations

The following additional types of declaration are available. They may not appear inside functions, and their scope is the file in which they are declared.

In the following, a *spec-expression-list* is a list of one or more *spec-expressions* separated by semicolons. A *spec-expression* is any expression that has no side effects. A *spec-expression* may refer to ghost declarations.

**assert( <spec-expression-list> )**  
**assert( ( <parameter-list> ) : <spec-expression-list> )**

The first form causes eCv to generate verification conditions that all the expressions in the *spec-expression-list* are true. Whether proven or not, each expression in the list will be assumed true when attempting to prove that the following expressions are true, and when proving subsequent verification conditions in the same scope.

In the second form, the parameter list must not be empty and all the parameters must be named. The prover generates verification conditions that all the expressions in the *spec-expression-list* are true for all possible values of the parameters.

You can also use **assert** as a statement (see below).

Note: if you wish to **#include "assert.h"** in your source file so that you can do run-time checking of assertions in a debug build, then we suggest you introduce any global assertions with the keyword **\_ecv\_assert** instead of **assert**, otherwise your compiler will report a syntax error. Alternatively, enclose them within **#ifdef \_\_ECV\_\_ ... #endif**.

**assume( <spec-expression-list> )**  
**assume( ( <parameter-list> ) : <spec-expression-list> )**

The first form causes the prover to assume that each expression in the *spec-expression* is true, without generating a verification condition. Useful when you want to provide information that eCv has no way of finding out for itself. Here is an example:

```
struct Foo {
    int x;
    double y;
}

assume(sizeof(struct Foo) <= 2 * min_sizeof(struct Foo))
```

Although eCv can calculate a minimum value for the size of a structure, it cannot calculate a maximum value because the amount of padding inserted by the compiler is unknown. This means that eCv is normally unable to prove that calculations involving the size of a structure do not overflow. In this example, we use an **assume** declaration to put an upper limit on **sizeof(struct Foo)**, thereby avoiding this problem.

The second form causes the prover to assume that each expression in the *spec-expression* is true, for all possible values of the parameters.

You can also use **assume** as a statement (see below).

## Additional eCv statements

**assert(<spec-expression-list>);**



Causes `eCv` to generate a verification condition that each *spec-expression* is true at that point, and then to assume it is true when proving subsequent verification conditions.

Note: if you wish to **#include "assert.h"** in your source file so that you can do run-time checking of assertions in a debug build, then we suggest the following:

- Make the inclusion of **assert.h** conditional (see chapter 2);
- Use only one *spec-expression* in the *spec-expression-list*;
- Avoid using ghost functions and fields in this *spec-expression*.

This will avoid assertions that are legal in `eCv` but not in the standard definition of the **assert** macro. You can still write `eCv`-only assertions by using **`_ecv_assert(<spec-expression-list>)`** to introduce an assertion that is visible to `eCv` but not to your compiler, even when you have included **assert.h**.

### **assume(<spec-expression-list>);**

The main use of **assume** is as described above; however **assume** can also be used within a statement list, in a similar way to **assert**. When **assume** is used in this way, each expression in *spec-expression-list* is assumed true without generating a verification condition.

This is useful when you want to provide information that `eCv` has no way of finding out for itself.

### **ghost(<statement;>)**

Used to introduce a statement that is invisible to the compiler, typically for the purpose of changing the value of a ghost variable. A ghost statement may not change the value of any non-ghost variables.

### **pass;**

A "do nothing" statement, which can be useful when, for instance, you want to provide an empty loop body.

## **Additional eCv expressions**

In specifications and within the declarations of ghost functions, you can use some special `eCv` expression types as well as side-effect-free expressions from the C language. The [not-null](#) expression (which can be used in normal code as well) may be useful, too.

### **Binary operators**

#### **<cast-expression\_1> .. <cast-expression\_2>**

Builds a sequence (i.e. an array) of values comprising *cast-expression\_1*, *cast-expression\_1+1*, *cast-expression\_1+2* and so on up to *cast-expression\_2*. If *cast-expression\_1* > *cast-expression\_2*, the sequence is empty. The expressions must both be of integral type (in which case they are promoted to **integer**, which is the type of unbounded integers), or of type **char**, or of type **wchar\_t**, or of the same enumeration type.

We suggest using a space before and after the `".."` operator to avoid the possibility that the C preprocessor treats the whole construct as a preprocessing number. In particular, if *cast-expression\_1* is an integer literal, *cast-expression\_2* starts with a macro name (for example, **result**), and there are no spaces, then the preprocessor will not recognize the macro.

#### **<logical-or-expression\_1> => <logical-or-expression\_2>**

The `=>` operator means implication, so that `a => b` means "a implies b". This is equivalent to `(!a || b)` but is easier to read in some specification contexts, such as preconditions.

The precedence of `=>` is lower than `||`, so any `&&` or `||` operators in the operands are evaluated before the implication

is evaluated.

### **<relational-expression> in <shift-expression>**

Returns **true** if and only if the left operand occurs in the right operand. The right operand must be an array, set or bag whose element type is the same as the type of the left operand. Or the right operand can be a map whose domain type is the same as the type of the left operand.

### **<multiplicative-expression> idiv <cast-expression>**

### **<multiplicative-expression> imod <cast-expression>**

### **<multiplicative-expression> \_ecv\_pow <cast-expression>**

**idiv** is a version of the integer division operator `/` that always rounds down (i.e. towards minus infinity). **imod** is a version of the integer modulus operator `%` that always returns a non-negative result. Both of these have the precondition that the second operand is greater than zero.

**\_ecv\_pow** is an operator that raises the left-hand operand to the power of the right-hand operand, equivalent to the `^` operator in *Perfect*. The types of the operands may be (integer, integer), (real, real) or (real, integer).

## Compound literals

### **(<type>){<initializer-list>}**

This is the compound literal expression from C99. When eCv is run in C90 or C++ mode, it can be used in specifications and other ghost contexts. It constructs a value of *type* from the values in *initializer-list*. For example, the expression `(int[3]){45, 32, 12}` yields an array of three integer values. Don't be misled by the word "literal", the expressions in *initializer-list* do not need to be constant-expressions. You may not declare new structs, unions or enums inside *type*.

## Disjoint expressions

### **disjoint(<expression\_1>, <expression\_2>, ...)**

Yields **true** if the storage associated with each expression in the argument list does not overlap with the storage associated with any other expression in the list. Each expression may be an lvalue or an expression of the form "**some**(*type-name*)" (see also *writes-clauses* in function specifications). To be meaningful, at least one of the expression must be of the form `*p` where *p* is a pointer, or `a[i]` or `a.all` where *a* is an array pointer. Used mostly in function preconditions and structure invariants, to state that some parameters or fields of pointer type don't alias the same memory, or don't point into certain static variables, or never point into structures of certain types.

## Holds expression

### **<relational-expression> holds <member-name>**

This yields **true** if *expression* was last assigned a value through *member-name*. The type of *expression* must be a union, and *member-name* must be one of the members of that union.

## 'Old' operator

### **old <postfix-expression>**

Within a postcondition, any mention of a variable or other object normally refers to the value of that variable or object when the function returns. [Exception: any mention of a function parameter in a postcondition always refers to the initial value of that parameter, because changes to the values of parameters are not visible to the caller of the function.] However, it is often useful to refer to initial values in a postcondition as well, so that the postcondition can describe how final values relate to initial values. You can do this by applying the **old** operator to the expression whose initial value you are interested in.

Here is an example:

```
static unsigned int count;

void updateCount(unsigned int *p)
writes(count; *p)
pre(p != &count)
pre(count + *p <= maxof(unsigned int))
post(count == old count + old(*p))
post(*p == 0)
{
    count += *p;
    *p = 0;
}
```

You can also use the **old** operator in a loop invariant or loop variant, to refer to the value of an expression just before the loop was entered for the first time.

## Named subexpressions

( let <identifier-1> = <expression-1>; let <identifier-2> = <expression-2>; ... ; <expression> )

This allows you to construct an expression by first naming a subexpression and then using that name as many times as you wish, thereby avoiding the need to write out the subexpression in full each time. For example, to compute the fourth power of *x* in a ghost context we could use:

```
( let square = x * x; square * square )
```

The scope of each name is the remainder of the parenthesised expression in which it is declared.

## Type operators

### default(<type-name>)

Yields the value of the type when it has default initialization, for example the initial value of a static variable that was declared without an initializer. For integral types this is also the value corresponding to a bit pattern of all zeros. This is *not* necessarily true for pointer types and floating-point types.

### maxof(<type-name>)

Yields the maximum value of the specified type, which must be an enumeration or unconstrained integral type, or an alias for such a type.

### minof(<type-name>)

Yields the minimum value of the specified type, which must be an enumeration or unconstrained integral type, or an alias for such a type.

### min\_sizeof(<type-name>)

Yields the lower limit of *sizeof(type-name)*. If *sizeof(type-name)* is known exactly by eCv, then *min\_sizeof(type-name)* is equal to *sizeof(type-name)*. For a structure, *min\_sizeof(struct S)* is the sum of the sizes of all the fields of *S*, that is, the size that a *struct S* would have if there is no padding.

### zero\_init(<type-name>)

Yields the value of type-name corresponding to zero initialization. Same as **default(type-name)** for integral and character types; undefined (i.e. platform-dependent) for other types.

## Quantified expressions

In the following, a *collection-expression* is an expression with type  $T[]$ ,  $\text{set}\langle T \rangle$  or  $\text{bag}\langle T \rangle$  for some type  $T$ . A *bool-expression* is an expression that has type **bool**.

**forall <type> <identifier> :- <bool-expression>**

Yields **true** if, for every value of *identifier* in *type*, *bool-expression* is true.

**forall <identifier> in <collection-expression> :- <bool-expression>**

Yields **true** if, for every value of *identifier* in the array or collection *expression*, *bool-expression* is true.

**exists <type> <identifier> :- <bool-expression>**

Yields **true** if, for some value of *identifier* in *type*, *bool-expression* is true.

**exists <identifier> in <collection-expression> :- <bool-expression>**

Yields **true** if, for some value of *identifier* in the array or collection *expression*, *bool-expression* is true.

## Operations on collections

**that <identifier> in <collection-expression> :- <bool-expression>**

Yields the single element in *collection-expression* for which *bool-expression* is true. There must be exactly one such element.

**those <identifier> in <collection-expression> :- <bool-expression>**

Selects those elements in *collection-expression* for which *bool-expression* is true, yielding a new collection containing those elements. If the collection is an array, then the elements in the result appear in the same order as they did in the original.

For example, the following expression:

```
those x in arr :- x >= 0
```

yields an array comprising the non-negative elements of *arr*.

**for <identifier> in <collection-expression> yield <expression>**

Yields a new collection obtained by mapping the elements of *collection-expression* using *expression*. The original collection is unchanged. If the collection is an array, then the elements in the result appear in the same order as the elements they were derived from appeared in the original.

For example, if *arr* has type **int**[ ] then the expression:

```
for x in arr yield x + 1
```

yields an array of type **integer**[ ] with the same number of elements as the original, in which each element is one greater than the corresponding element in *arr*.

**for those <identifier> in <collection-expression> :- <bool-expression> yield <expression>**

Yields a new collection obtained by selecting those elements of *collection-expression* for which *bool-expression* is true and mapping them using *expression*. The original collection is unchanged. If the collection is an array, then the elements in the result appear in the same order as the elements they were derived from appeared in the original.

For example, if *arr* has type **int**[ ] then the expression:

```
for those x in arr :- x >= 0 yield x + 1
```

first selects the non-negative elements of *arr* and then adds 1 to them, yielding an array of type **integer[ ]**.

### **<binary-operator> over <collection-expression>**

Applies the specified binary operator sequentially over the elements of the collection, starting from the element at index 0. For example, the expression `+ over arr` where *arr* has type `int[ ]` yields the sum of the elements. If the sequence has only one element, then the result is the value of that element. If the operator is known to eCv and has a left-identity value declared, then it is permissible for the sequence to be empty, in which case the value is the left identity. Otherwise, this construct has the precondition that the sequence is not empty. Operators `+` and `*` over integral types and floating types all have a left identity element known to eCv.

[▲ TOC](#)

**eCv Manual, Version 7.0, February 2017.**

**© 2017 Escher Technologies Limited. All rights reserved.**

# Predefined ghost types, functions and fields

A ghost type, function or field is one that you can refer to in specifications, but not in code. As well as the predefined ones that eCv provides, you can [declare your own](#) ghost types, variables, functions and function parameters, and you can declare ghost fields of your own struct and union types.

The predefined ghost fields of pointers and array pointers are read-only. Even though ghost fields are not stored at run-time, eCv can track their values and reason about them.

## Global ghost variables

Variable name and type	Description	Where declared
<code>_ecv_seq&lt;_ecv_VolatileBase&gt;</code> <code>_ecv_vol_ops</code>	Records the sequence of reads from and writes to volatile variables performed by the program	Built in

## Global ghost functions

Where these functions have names not beginning with `_ecv_`, they are declared as macros in the header files indicated, in terms of intrinsic functions whose names begin with `_ecv_`. You can therefore rename any whose names clash with your own identifiers.

Function name and signature	Description	Where declared
<code>bool _ecv_isNullTerminated(const char * array str)</code>	Yields <b>true</b> if there is a null in <i>str</i> with an index between zero and the upper bound of <i>str</i> . Typically used in function preconditions to state that a parameter of type <code>char* array</code> or <code>const char* array</code> must be a null-terminated string.	Built in
<code>bool _ecv_allBytesAre(const void* array p, int val, size_t num)</code>	Yields <b>true</b> if the first <i>n</i> characters of memory addressed by <i>p</i> have the value <i>val</i> . Used to describe the semantics of <i>memset</i> and similar functions.	Built in
<code>bool _ecv_equalBytes(const void* array p, const void* array q, size_t num)</code>	Yields <b>true</b> if the first <i>n</i> characters of memory addressed by <i>p</i> and <i>q</i> are correspondingly equal. Used to describe the semantics of <i>memcpy</i> and similar functions.	Built in
<code>template&lt;typename T&gt; T _ecv_readVolatile(T* p) writes(_ecv_vol_ops)</code>	Read from the volatile variable at <i>*p</i> , record the read in <code>_ecv_vol_ops</code> and return the value read.	Built in
<code>template&lt;typename T&gt; T _ecv_writeVolatile(T* p, T val) writes(_ecv_vol_ops)</code>	Write <i>val</i> to the volatile variable at <i>*p</i> , record the write in <code>_ecv_vol_ops</code> and return <i>val</i> .	Built in

## Ghost fields of array pointer types

Each array pointer type `T* array` (where *T* is any type) has the following ghost fields:

Field name	Type	Description

all	$T[]$	All the elements of the array that is pointed to, including elements with a negative index (if any). If <i>ap</i> is an array pointer, then <i>ap.all</i> is a bit like <i>*ap</i> except that it returns the value of the whole array, instead of just the value of the array element that <i>ap</i> points to. <b>Important:</b> if you write a function postcondition involving <i>a.all</i> where <i>a</i> is a function parameter with array pointer type, then the postcondition is required to apply to the whole array including any negative indices. If the code of your function does not take account of possible negative indices, then the postcondition is likely to be unprovable. You can get round this by specifying that <i>a</i> is a pointer to the start of an array (so that there are never any negative indices), for example by including <i>a.lwb == 0</i> in the function precondition.
base	$T^*$ array	A pointer to the same array re-based to point to its first element (so that <i>offset</i> is zero). You can test whether two pointers <i>a</i> and <i>b</i> point into the same array using the expression <i>a.base == b.base</i> .
count	integer	The count of all elements in the array, equal to <i>offset + lim</i> .
indices	seq<integer>	The sequence <i>lwb..upb</i> , i.e. a sequence comprising all the valid indices of the array pointer.
lim	integer	One greater than the highest valid index. For any array pointer that was not created by pointer arithmetic, this is the number of elements in the array. Never less than zero.
lwb	integer	The lowest valid index, equal to <i>(-offset)</i> .
offset	integer	The offset of the pointer into the array relative to the start of the array, as a number of elements. Zero for any array pointer that was not created by pointer arithmetic. Always between zero and <i>count</i> inclusive.
upb	integer	The highest valid index, equal to <i>lim - 1</i> . Never less than -1.

## Ghost fields of the void pointer type

The void pointer type **void\*** has the following ghost fields:

Field name	Type	Description
all	Universal object type	The whole object that the pointer points to or into.
lim	integer	The size of the object in characters, not including any part of it that has a negative offset from the pointer. If <i>pv</i> has type <b>void*</b> then $((\text{char}^*)pv).lim == pv.lim$ . Never less than zero.
offset	integer	The offset of the pointer into the object relative to the start of the object, as a number of characters. Zero for any <b>void*</b> pointer that was obtained by converting a non-array pointer or a pointer to the start of an array.

## Ghost fields of array types

Each array type  $T[]$  (where *T* is any type) has the following ghost fields:

Field name	Type	Description
count	integer	The count of elements, i.e. the same value as <i>lim</i> .
indices	seq<integer>	The sequence <i>0..upb</i> , i.e. a sequence comprising all the valid indices of the array pointer.
lim	integer	One greater than the highest valid index, i.e. the number of elements in the array.
lwb	integer	The lowest valid index. Always zero.
upb	integer	The highest valid index, equal to <i>lim - 1</i> .

Note: there is no ghost member **all** for array types, because this would just return a copy of the array itself.

## Ghost member functions of array and sequence types

Each array type  $T []$  or sequence type  $\text{seq}<T>$  (where  $T$  is any type) has the following ghost member functions:

Function name	Preconditions	Description of return value
$\text{seq}<T>$ drop(integer $n$ )	$0 \leq n; n \leq \text{lim}$	A sequence comprising the array elements with the first $n$ removed
bool isndec()	Element type $T$ is a comparable type	<b>true</b> if and only if the elements are in nondecreasing order
bool isninc()	Element type $T$ is a comparable type	<b>true</b> if and only if the elements are in nonincreasing order
$T$ max()	Element type $T$ is a comparable type; $\text{lim} \neq 0$	The maximum element
$T$ min()	Element type $T$ is a comparable type; $\text{lim} \neq 0$	The minimum element
$\text{seq}<T>$ permndec()	Element type $T$ is a comparable type	A sequence comprising the elements of the array sorted into nondecreasing order
$\text{seq}<T>$ permninc()	Element type $T$ is a comparable type	A sequence comprising the elements of the array sorted into nonincreasing order
$\text{seq}<T>$ rev()		A sequence comprising the elements of the array in reverse order
$\text{seq}<T>$ slice(integer $start$ , integer $len$ )	$0 \leq start; 0 \leq len; start + len \leq \text{lim}$	A sequence comprising $len$ elements from the array starting at index $start$ . Equivalent to $\text{drop}(start).\text{take}(len)$
$\text{seq}<T>$ take(integer $n$ )	$0 \leq n; n \leq \text{count}$	A sequence comprising the first $n$ elements of the array

Other ghost members of array and sequence types are available, as detailed in the *Library Reference* section of the *Perfect Developer Language Reference Manual* (look up class **seq of X**).

 [TOC](#)

eCv Manual, Version 7.0, February 2017.

© 2017 Escher Technologies Limited. All rights reserved.



# Support for C++ source code in eCv++

## Supported and unsupported constructs

Escher C++ Verifier (or eCv++ for short) provides formal verification of programs written in subsets of C++ as defined by the 1998 (ISO 2003) and ISO 2011 standards. The subset is carefully designed to meet the safety requirements of high-integrity projects, for example projects written to meet IEC61508 SIL 3 to 4, or DO-178C DAL A and B. Many C++ constructs are excluded from the subset, and other constructs are included but restricted. Nevertheless, the eCv++ subset of C++ provides several advantages over C, such as encapsulation, information hiding, and object-oriented software development.

The eCv++ subset of C++ was derived by starting from the eCv subset of C and adding only those features of C++ that are well-understood, are unambiguous, introduce little or no risk when they are used, and are of obvious benefit in the context of high-integrity embedded software.

The major features of C++ that are included in the current eCv++ subset are:

- Comments introduced by `//`
- Declarations that are not at the start of a block
- **const\_cast<>**, **static\_cast<>** and **reinterpret\_cast<>** operators
- Function overloading and default parameter values
- Classes, including single inheritance, and member variables, functions, operators (with restrictions, see below), and constructors
- Virtual functions and dynamic binding
- **extern "C" { ... }** declarations
- Class, struct, union and enum names are recognized as type names (i.e. no need to introduce them with 'struct' or 'union' or 'enum' every time you refer to them)
- The first clause of a for-loop header may be a declaration instead of an expression
- C++ initialization syntax for variables
- **bool** type
- **wchar\_t** is a type, not a typedef
- Reference types
- Templates with class parameters (with restrictions, see below)
- Namespaces

Features of C++ that are being considered for a future version of the eCv++ subset are:

- Templates with non-type (e.g. integer) parameters
- Multiple interface inheritance
- Placement **new** operator

Features of C++ that are not supported, and not currently planned to be supported include:

- **new** and **delete** operators (because dynamic memory allocation is normally prohibited in software for critical embedded systems)
- Most of the Standard Template Library (because it depends on dynamic memory allocation for its

implementation)

- Destructors
- User-defined copy constructors
- User-defined assigning operators
- User-defined '==', '->' and unary '\*' operators
- Type-conversion operators
- Exceptions
- Pointer-to-member
- Multiple inheritance where more than one base is not an interface
- **mutable** member variables

A small number of features from the C++ 2011 ISO standard (C++'11) are supported, even when the compiler type is set to C++'98, as follows:

- **final** applied to a class declaration
- **override** and **final** keywords applied to method declarations in derived classes
- **nullptr** keyword
- **static\_assert**
- Scoped enumerations (**enum class**)

## Additional eCv++ keywords

When processing C++ code, in addition to the keywords listed in Chapter 2 and all the C++'98 keywords (whether supported by eCv++ or not), the following words are treated as keywords:

Normal keyword	Underlying keyword	Where used	See section
early	<code>_ecv_early</code>	Indicates that a member function does not depend on dynamic binding	<a href="#">Early functions</a>
final	<code>_ecv_final</code>	Indicates that a class may not be inherited from, or that virtual function may not be overridden in a descendant class	<a href="#">Function overriding and hiding</a>
from	<code>_ecv_from</code>	Qualifies the declaration of a pointer or reference to indicate that it may point or refer to a class derived from the one indicated in the type	<a href="#">Polymorphic pointers</a>
nullptr	<code>_ecv_nullptr</code>	Null pointer expression	<a href="#">Null pointer expression</a>
override	<code>_ecv_override</code>	Indicates that the function being declared overrides an inherited virtual function	<a href="#">Virtual functions</a>

The **final** and **override** keywords have the same meaning in eCv as the corresponding identifiers in C++'11. However, eCv treats them as keywords in all contexts, rather than in certain contexts only as C++'11 does. When the source file is compiled by a C++ compiler that implements an earlier version of the standard, they are defined in `ecv.h` so as to expand to nothing, which makes them invisible to the compiler. Like other eCv keywords, you can **#undef** them and use keywords `_ecv_final` and `_ecv_override` instead if they conflict with identifiers used in your program.

The **nullptr** keyword has the same meaning as the corresponding C++'11 keyword. It signifies a null value that can be converted implicitly to any nullable pointer type, but not to an integral type. For your convenience, it is defined in eCv as **(0)** when you are compiling the source using a compiler that implements an older C++ standard.

## Casts in C++ programs

When you set the source language to C++, a warning will be generated if you use a C-style cast expression that is not equivalent to a **static\_cast**. You are expected to use **reinterpret\_cast** or **const\_cast** to express these more dangerous types of cast. Example:

```
int a = 1;
int *pa;
const int c = 2;
unsigned char b;
unsigned char *pb;

b = (unsigned char)a;           // OK, cast is equivalent to static_cast
pb = (unsigned char*)&a;       // error, need to use reinterpret_cast here
pa = (int*)&c;                  // error, need to use const_cast here
```

## Overloaded functions and ambiguity resolution

When a function call is potentially ambiguous because the name of the called function has been overloaded, eCv does not apply the standard C++ disambiguation rules, which are complicated and sometimes give surprising results. Instead, eCv applies the stricter rule that the call must match exactly the argument types of one of the candidate function declarations, with no implicit type conversions required. Otherwise, the call is considered ambiguous and an error is reported. Example:

```
void foo(const int*, short);
void foo(int*, int);
int i;
...
foo(&i, 1L);           // OK in C++, ambiguous in eCv++
```

## Inheritance and final classes

eCv++ supports single, public inheritance only. You may declare a class **final** to indicate to eCv++ that no other class inherits it. This makes verification by eCv++ easier in some cases. Therefore, you are recommended to declare classes **final** where possible. The syntax used by eCv++ to indicate that a class is final is the same as in C++11. Example:

```
class Foo final {
    ...
};

class Derived final : public Base {
    ...
};
```

## Polymorphic pointers

Given a class *Foo*, in eCv++ the type *Foo\** means a pointer to an object of exactly class *Foo*. If you wish to declare a pointer that may point to *Foo* or any class derived from *Foo*, then you must qualify the pointer declaration with the eCv++ keyword **from**. If the pointer is also declared nullable, then **from** precedes **null** in the declaration. Here are some examples:

```
class AbstractBase {
public:
    virtual int foo(int) = 0;           // Pure virtual function declared, so
AbstractBase is an abstract class
};

class ConcreteBase {
public:
    virtual int foo(int);
};

class Derived1 : public AbstractBase {
public:
    int foo(int) override;           // Overrides the only pure virtual function,
so Derived is a concrete class
};
```

```

class Derived2 : public ConcreteBase {
public:
    int fool(int) override;
};

AbstractBase* pa1;           // Error, there can be no objects exactly of
type AbstractBase
AbstractBase* from pa2;     // OK, pa2 may point to objects of types
derived from AbstractBase
ConcreteBase* pc1;         // OK, pc1 may point to objects whose type is
exactly ConcreteBase
ConcreteBase* from null pc2; // OK, pc2 may be null or point to objects
whose type is or is derived from ConcreteBase

```

Within the specification and body of a nonstatic non-const member function of a class C, the type of the **this** pointer is **C\* from** unless C is declared **final**, in which case it is just **C\***. For **const** nonstatic member functions, the type of **this** is **const C\* from** if C is not declared **final**, and **const C\*** if it is. Within the body of a constructor of class C, the type of **\*this** is always **C\*** whether or not C is declared **final**.

## Constructors

A constructor must initialize all fields of **\*this**. If a constructor is declared with a writes-clause, you should not include **\*this** in the writes-clause, because it is implicitly assumed. However, any other variables (including static class member variables) directly or indirectly written by a constructor must be mentioned in its writes-clause.

A constructor declared without a writes-clause is assumed to write its non-const pointer parameters according to the same rules as for global and static functions, as well as **\*this**. Therefore, if a constructor takes a non-const pointer parameter that is used only to initialize member data, you should specify **writes()** to indicate that it does not write to the object addressed by the parameter.

If a constructor is defined with an empty body, then the constructor postcondition will be inferred from the base and member initializers. You may provide an explicit postcondition instead if you prefer, for example to allow you to specify both the state achieved by the initializers and some additional properties of the state that you expect to hold. If the body is not empty, no postcondition is inferred.

Member initializers must be specified in the same order in which the members are declared. This is so that any side-effects of the initializers will occur in left-to-right order.

A constructor may not call any nonstatic member functions of the current class or its bases other than functions declared **early** (see [Early member functions](#)). A constructor may not pass **this** to any function it calls, other than implicitly to an early member function of the same class or a base class.

Any constructor that can be called with no arguments (a 'default constructor') may not have side-effects. For example, it may not modify static or global variables, or static variables of any class (including its own).

Any constructor that can be called with exactly one argument must be declared **explicit**. This is to prevent such constructors from introducing new implicit type conversions.

## Member functions

If a nonstatic member function has an explicit writes-clause, then just as with a non-member function, you must specify in the writes clause everything that it directly or indirectly modifies - including any parts of **\*this**. You may include **\*this** in the writes clause, or just some member variables as required. If a nonstatic member function has no writes-clause, then it is assumed to write through any non-const pointer parameters (as for a non-member function), and if the function is not declared **const** then it is also assumed to write **\*this**.

## Early member functions

A non-virtual nonstatic member function may be declared **early** to indicate that it does not depend on dynamic binding. An early function may not call any other member functions other than early functions, and may not store the

**this**-pointer in any non-local variables, nor pass it as a parameter to another function. Example:

```
class Base {
    int m_x, m_y;
public:
    virtual int foo1(int) = 0;

    int foo2(int) early {
        m_x = 0; // OK
        foo1(); // Error - call to non-early function not
allowed
    }

    int foo3(int) const early {
        return m_x + m_y; // OK
    }
};
```

Taken together, these restrictions ensure that the result of calling an early function does not depend on dynamic binding of the **this**-pointer. This means that early functions may safely be called from constructors and invariant declarations.

If a class is declared **final**, or it has no direct or inherited virtual member functions, or all its virtual member functions have been declared **final**, then all the member functions of that class cannot depend on dynamic binding of **this** and are therefore implicitly **early**.

## Function overriding and hiding

Within the declaration of a derived class, whenever you override a virtual function inherited from the parent class, you must indicate your intention to override the inherited member using the **override** keyword. Declaring the overriding member **virtual** as well is not necessary but is permitted. You may also indicate that the function is not overridden in any child of the current class using the **final** keyword. eCv++ does not allow you to declare a member in a derived class that hides a non-virtual member of the same name in a base class.

The syntax for using **override** and **final** is the same as in C++'11. Here are some examples:

```
class Base {
public:
    virtual int foo1(int);
    virtual int foo2(int);
    virtual int foo3(int) const;
    int foo4(int);
};

class Derived : public Base {
public:
    int foo1(int); // Error, missing 'override'
    int foo2(int) override; // OK
    int foo3(int) const override final; // OK
    int foo4(int); // Error, hides foo4(int) in class Base
};
```

## Specifications of overriding functions

Where a function overrides a virtual function inherited from a base class, there are restrictions on its specification, as follows:

- The overriding function inherits the writes-clause of the overridden function and **may not have its own writes-clause**. For a non-const overriding function, this means that it is not permitted to write any member variables declared in its class, unless the overridden function implicitly or explicitly writes **\*this**. Therefore, when a non-const virtual function is declared with an explicit writes-clause, the clause should normally include **\*this**.
- If the overriding function is declared *without* a precondition, then it inherits the precondition of the overridden function. If the overriding function is declared *with* a precondition, then the precondition of the overridden function must imply the precondition of the overriding function. A verification condition is generated to ensure that this is the case.

- If the overriding function is declared *without* a postcondition, then it inherits the postcondition of the overridden function. If the overriding function is declared *with* a postcondition, then the postcondition of the overriding function must imply the postcondition of the overridden function. A verification condition is generated to ensure that this is the case.

In short, an overriding function may **weaken the precondition** and/or **strengthen the postcondition** of the overridden function. Taken together, these restrictions implement the Liskov Substitution Principle (LSP), which is a necessary condition for the safe use of object-oriented methods in critical software. See for example the DO-332 Object-Oriented Technology and Related Techniques supplement to DO-178C.

Sometimes, when declaring an overriding function, it is useful to compose a new postcondition by taking the inherited postcondition and adding some extra conditions. The token "..." may be used as one of the postconditions of an overriding function to represent the inherited postconditions.

## Operator declarations

User-defined operators are supported, with the following restrictions at present:

- They must be nonstatic **const** class members
- They must have empty writes-clauses (whether explicit or implicit)
- Only the following operators may be declared: + - ~ \* / % & | ^ << >> < > <= >=

User-defined assigning operators and type conversion operators are not supported.

## Templates

Class and function templates are supported, with the following restrictions at present:

- Template parameters must be type parameters. Other sorts of template parameters (e.g. integer) are not supported.
- The instantiation of a template may not depend on the existence of particular members of the types with which it is instantiated, or functions etc. that accept those types. In C++ parlance, this means that a template definition must not have any dependent names.
- Specialization and partial specialization are not supported.

For an example of verifying a class template with eCv++, see queue.cpp in the installed example files.

 [TOC](#)

**eCv Manual, Version 7.0, February 2017.**  
**© 2017 Escher Technologies Limited. All rights reserved.**

# Appendix A - Compiler Settings

The following are sample compiler configuration (in the Options → C/C++ Compilers settings). They are used only to run the preprocessor.

Note: eCv may be supplied with some predefined compiler configurations, so you may not need to set these up yourself.

## Microsoft Visual C++ or Visual C++ Express

- executable is "C:/Program Files/Microsoft Visual Studio ###/vc/bin/cl.exe" where ### depends on the version you installed (8 for 2005, 9.0 for 2008 and 10.0 for 2010)
- additional path for executable files is "C:/Program Files/Microsoft Visual Studio ###/Common7/IDE"
- language is C90 or C++ as desired (C99 is not supported)
- char is signed unless the /J compiler option is used, in which case it is unsigned
- wchar\_t is unsigned
- sizes for char, short, int, long, long long are 8, 16, 32, 32, 64 respectively
- sizes for float, double, long double are 32, 64, 64 respectively
- pointer size is 32 bits when using the 32-bit compilers, or 64 bits when using the 64-bit compilers
- integer division rounds towards zero
- include path intro is "/I" (that's capital i as in Include)
- include path separator is " /I" (that's capital i as in Include; note the extra space at the start)
- #define intro is "/D"
- #define separator is " /D" (note the extra space)
- preprocessor command for C90 is: "/P /nologo /Za /TC \$d \$i \$x \$f".
- preprocessor command for C++ is: "/P /nologo /Za /TP \$d \$i \$x \$f".

Under 64-bit Windows, replace "Program Files" in all the above by "Program Files (x86)".

## gcc

- executable is "cpp" (this runs the preprocessor directly) - if you are running under Windows via MinGW or Cygwin, then use "cpp.exe" and provide the full path
- additional path for executable files - under Windows this should be the folder containing file cc1.exe
- language is C90, C99 or C++ as desired
- for modern versions of gcc, the sizes for various types and the signedness of plain char and wchar\_t are as defined by the Application Binary Interface for the platform you are targeting (the signedness of plain char can be overridden by compiler options *-funsigned-char* and *-fsigned-char*, and wchar\_t can be forced to be equivalent to unsigned short int with *-fshort-wchar*)
- integer division rounds towards zero
- include path intro is "-I" (that's capital i as in Include)
- include path separator is " -I" (that's capital i as in Include; note the extra space at the start)
- #define intro is "-D "

- #define separator is " -D " (note the extra space at the start)
- preprocessor command for C90 is: "-x c -std=c90 -o \$o \$d \$i \$x \$f".
- preprocessor command for C99 is: "-x c -std=c99 -o \$o \$d \$i \$x \$f".
- preprocessor command for C++ is: "-x c -std=c++98 -o \$o \$d \$i \$x \$f".

**Important:** some of the header files for gcc use non-standard syntax of the form `__attribute__((...))`. In order to hide this syntax from eCv, file `eCv.h` **#defines** `__attribute__((_x))` as a null macro when both `__ECV__` and `__GNUG__` are defined. However, certain gcc-related header files (e.g. `_mingw.h`) contain the line `#undef __attribute__`. You will need to remove any such lines, or surround them with `#ifndef __ECV__ ... #endif`.

 [TOC](#)

**eCv Manual, Version 7.0, February 2017.**

**© 2017 Escher Technologies Limited. All rights reserved.**



# Appendix B - Type system of eCv

The type system of eCv is based on C, with these changes:

- The type 'pointer to array of T' is distinct from 'pointer to T'. The syntax 'T\*' means 'pointer to T'. To specify 'pointer to array of T', use 'T\* array'. You may not apply indexing or pointer arithmetic to non-array pointers.
- Pointers (including pointers to arrays) are not nullable by default (i.e. zero is not an allowed value). You can specify that null is an allowed value using the type qualifier 'null', e.g. 'T\* null' or 'T\* array null'. The order of qualifiers 'array' and 'null' (and any other type qualifiers you use) is not significant. There is an implicit type conversion from each plain pointer type to the corresponding nullable pointer type, but not the other way round. To convert the other way, use 'not\_null(e)' which asserts that e is not null and converts its type from 'T\* null' to 'T\*', or from 'T\* array null' to 'T\* array'.
- Union types have an additional *ghost* field called the discriminant, which remembers the name of field the union was last assigned through. This field cannot be referred to at run-time (because it isn't stored), however it can be referred to in specifications via the **holds** operator. You may only access a union member if it was most recently assigned via the same member.
- **bool** is a separate type
- **char** is a separate type, not an alias for **signed char** or **unsigned char**
- **wchar\_t** is a separate type, not an alias for some other integral type
- Each enumeration declaration creates a distinct type, not an alias for an integral type

## Semantic differences between eCv and C90

- The type of a string literal in eCv is **const char\* array** (like C++ with the addition of **array**) instead of **char\***. Therefore, you cannot provide a string literal in a context where a value of type **char\*** is required. This difference is backwards-compatible in the sense that any legal use of a string literal in eCv is also a legal use in C and has the same semantics.
- The type of a wide string literal is **const wchar\_t\* array** instead of **wchar\_t\***.
- The type of a character literal in eCv is **char** (like C++) instead of **int**. If you want to use a character literal in a situation where a value of type **int** is required, you must cast it explicitly to type **int**. This change is backwards-compatible in that any legal use of a character literal in eCv is also a legal use in C, although some compilers might generate a warning that an explicit cast of a character literal to type **int** is redundant. You can define a char-literal-to-int conversion macro to avoid any such warnings. Note that in eCv it is illegal to use a character literal as the operand of **sizeof**, because this would yield a different result from C.

[▲ TOC](#)

eCv Manual, Version 7.0, February 2017.

© 2017 Escher Technologies Limited. All rights reserved.

# Appendix C - Constructs you may see in proof output

The math used by the prover is different from C maths, so you may see some expressions that are neither part of standard C nor part of the eCv specification extensions. Here is a list of [some of] them. You should not see these in error messages, or anywhere else other than in proof/unproven output files - let us know if you do.

## Binary operators ./ and .%

These are round-towards-zero versions of the / and % integer operators. Plain / and % in proofs refer to the version of integer division that always rounds down and the version of modulo that yields a result having the same sign as the divisor.

## Unary operator #

This means the number of elements in the collection operand. For an array *arr*, *#arr* equates to *arr.count*.

## Subscripted variables

A variable of the form **foo**<sub>543,21</sub> means the value of variable **foo** at line 543 column 21 in the .i file that was generated by the preprocessor. Similarly, **heap**<sub>543,21</sub> means the value of the heap at that location.

## "\$r.value" expressions

The expression **p.\$r.value** where **p** is a simple pointer is the prover's internal representation of **\*p**. The expression **ap.\$r.value** where **ap** is an array pointer is the prover's internal representation of **ap.all**, i.e. all the elements in the array to which **ap** points.

[▲ TOC](#)

eCv Manual, Version 7.0, February 2017.

© 2017 Escher Technologies Limited. All rights reserved.

# Appendix D - Verification condition types

Verification condition types generated by eCv, and their meanings.

See separate document **Verification Conditions Generated by Escher Verification Studio**.

 [TOC](#)

eCv Manual, Version 7.0, February 2017.

© 2017 Escher Technologies Limited. All rights reserved.

# Appendix E - Language extensions for C99 and C++ 2011

If you are using a C99 compiler, then as well as constructs in the C90 subset supported by eCv, you may use the following C99 constructs:

- comments introduced by // (these are allowed by eCv even in C90 mode)
- declarations do not have to be at the start of a compound statement (however, eCv does not allow declarations directly in the cases of a switch-statement)
- **inline** storage class when declaring functions
- `_Bool` type (provided that you set up the correct definitions so that it is equivalent to eCv's **bool** type)
- compound literals (but not using named member notation)
- the first clause of a for-loop header may be a declaration instead of an expression

If you are using a C++ 2011 compiler, then as well as constructs in the C++ 1998 (ISO C++ 2003) subset supported by eCv, you may use the following C++ 2011 constructs:

- **final** reserved identifier when declaring a class
- **final** and **override** reserved identifiers when overriding an inherited virtual function
- **nullptr** keyword

Note that eCv treats **final** and **override** as keywords when processing C++ source, instead of reserved identifiers as specified in the C++ 2011 language standard. Therefore, they may not be used as normal identifiers. These keywords and the **nullptr** keyword are recognized even when the compiler type is set to C++ 1998.

[▲ TOC](#)

eCv Manual, Version 7.0, February 2017.  
© 2017 Escher Technologies Limited. All rights reserved.

# Appendix F

## Main differences between eCv in C'90 or C'99 mode and MISRA-C

Here are lists of the main differences between the C subset accepted by eCv and MISRA-C. Numbers in brackets are the corresponding MISRA rule numbers.

### MISRA-C 2004 guidelines not fully enforced by eCv

The MISRA-C 2004 standard has 141 rules constraining how the C language may be used. Compliance with about 128 of these rules can in principle be checked by source code analysis, with formal verification needed in the case of a few rules. eCv currently checks 55 MISRA-C 2004 rules in full and a further 16 rules in part. Future versions of eCv may support additional MISRA compliance checks.

Here are the MISRA-C 2004 rules that eCv allows to be violated without generating a warning or error message, when the compiler type is set to C90 and MISRA-C 2004 checking has been enabled in the project settings.

- eCv does not fully enforce rule 1.1 because it supports a few language extensions, such as **inline** functions, **long long int** type, and the syntax used by many embedded C compilers to place variables at particular addresses.
- Rules 1.3, 1.4 and 1.5 are outside the scope of source code static checking.
- eCv does not enforce rule 2.1.
- eCv permits comments introduced by // (2.2).
- eCv does not enforce rule 2.4.
- The documentation rules (3.x) are outside the scope of source code static checking. However, the behaviour of integer division is included when specifying the compiler/platform behaviour (3.3).
- eCv does not enforce rules 5.4, 5.5, 5.6 and 5.7.
- eCv enforces rule 6.1 and 6.2 to the extent that conversions in either direction between other integral types and plain char must be made explicit.
- eCv does not enforce rule 6.3.
- eCv does not require prototypes for functions whose definitions are visible at each place where they are called (8.1).
- eCv permits objects to be defined in header files, in particular inline function declarations (8.5).
- eCv does not enforce rules 8.6, 8.7, 8.8, 8.9, 8.10, 8.11 or 8.12.
- eCv does not enforce rule 9.3.
- eCv enforces part (a) of rules 10.1 and 10.2 but not parts (b), (c) or (d).
- eCv does not enforce rules 10.3, 10.4 or 10.5.
- eCv does not enforce rules 12.1, 12.4, 12.5 or 12.10.
- eCv cannot enforce rule 12.11 because it depends on detailed knowledge of the behaviour of the compiler you will be using. Wrap-around will not occur when using the eCv preprocessor.
- eCv does not enforce rule 12.13.
- eCv allows assignment-expressions to be used within expressions yielding a Boolean value (13.1), provided

that an assignment-expression is not used where a Boolean value is required. For example, "(a = b) && c" is forbidden, but "(a = b) == 0" is permitted.

- eCv allows floating point expressions to be tested for equality and inequality (13.3).
- eCv treats a for-loop as if it were an equivalent while-loop, therefore it does not place additional restrictions on the expressions in a for-loop header (13.4, 13.5) or on modifying for-loop variables (13.6).
- eCv does not prohibit Boolean operations whose results are invariant, however they will typically lead to "given false" warnings during verification, indicating unreachable code (13.7).
- eCv does not guarantee to detect unreachable code (14.1), however it will typically lead to "given false" warnings during verification.
- eCv does not check that a non-null statement either has a side effect or causes control flow to change (14.2).
- eCv does not check rule 14.3. You can use **pass** to introduce a null statement explicitly.
- eCv supports **continue** (14.5).
- eCv allows multiple **break** statements in a loop (14.6).
- eCv allows multiple **return** statements in a function (14.7).
- A **case** clause in a **switch** statement need not end in "**break**;" if it cannot fall through, for example if it ends in an if-statement for which both branches end in a jump statement (15.2).
- eCv only allows recursive functions if a recursion variant is declared (16.2).
- eCv does not enforce rule 16.3.
- eCv treats a function or function prototype declared with an empty parameter list as having no parameters; it does not insist on the parameter list being declared as "(void)" (16.5).
- eCv does not warn about pointer parameters that could have been declared **const** but were not (16.7).
- eCv does not enforce rule 16.9.
- eCv does not check that error information returned by a function is tested, provided the specification is met (16.10).
- eCv permits pointer arithmetic on array pointers (17.4), however verification of pointer arithmetic typically requires additional annotation, therefore pointer arithmetic should be avoided as far as possible.
- eCv does not place a limit on pointer indirection, however compliance with the MISRA limit of 2 is recommended (17.5).
- eCv does not enforce rule 17.6.
- eCv permits unions, provided they are not used to convert between different types (18.4).
- eCv does not enforce preprocessing rules 19.1, 19.2, 19.4, 19.5, 19.7, 19.9, 19.10, 19.13 or 19.15.
- eCv partially enforces rule 20.1 but does not enforce the remaining library rules (20.x).

#### Notes:

1. The commenting rules (2.x) and preprocessor rules (19.x) are only checked if you are using eCv's own preprocessor, not if you are using your compiler's preprocessor.
2. Compliance with rules 1.2, 12.2, 12.8, 17.2, 17.3 and 21.1 can only be ensured if verification is run and the verification conditions corresponding to these rules are successfully discharged by the theorem prover.

## eCv rules that are stronger than the MISRA-C 2004 guidelines

- eCv has a Boolean type **bool**. The underlying type of an operator expression in which the operator is a comparison or logical operator is **bool**. Implicit conversions between **bool** and other types other than 1-bit unsigned int bit fields generate warnings.

- eCv has strongly-typed enumerations. The underlying type of an enumeration constant or a variable of enumeration type is that type, not **int**. Implicit conversions from enumeration types to **int**, **long int** and **long long int** do not generate warnings. Implicit conversions from integral to enumeration types *do* generate warnings.
- Type **wchar\_t** is treated as a separate type by eCv, not as a **typedef** for some integral type. It is treated in a similar way to plain **char**, i.e. all conversions between **wchar\_t** and other types must be explicit.
- Pointers to arrays must be qualified by the **array** keyword when they are declared.
- Pointers that are allowed to take the null pointer value must be qualified by the **null** keyword when they are declared.
- eCv will warn about conversions between different pointer types, except for conversions to **void\***.
- eCv does not allow any part of a conditional expression to have side effects.
- eCv does not allow you to take the address of a member of a union, or of any part of a member of a union.
- eCv does not allow you to use the **sizeof** operator with a character literal operand.

## MISRA-C 2012 guidelines not fully enforced by eCv

The MISRA-C 2012 standard has 16 directives and 143 rules constraining how the C language may be used. Compliance with most rules can in principle be checked by source code analysis, with formal verification needed in some cases. Compliance with some of the directives can also be partially checked by source code analysis and formal verification, although in most cases additional information is needed. eCv currently checks 68 MISRA-C 2012 rules in full, and a further 6 rules and one directive in part. Future versions of eCv may support additional MISRA compliance checks.

Here are the MISRA-C 2012 directives and rules that eCv allows to be violated without generating a warning or error message, when the compiler type is set to C90 or C99 and MISRA-C 2012 checking has been enabled in the project settings.

- Directives 1.1, 2.1, 3.1, 4.2 are outside the scope of formal verification
- The eCv verifier enforces directive 4.1 in respect of integer arithmetic overflow, pointer arithmetic, array bound errors, function parameters (provided the functions are specified with the correct preconditions) and pointer dereferencing. It does not verify absence of overflow of floating point computations. eCv does not support dynamic memory.
- eCv enforces rule 1.1 in respect of syntax (apart from the extensions supported by eCv) and constraints, but not in respect of implementation limits.
- eCv supports a small number of common language extensions (rule 1.2)
- When code is unreachable, the eCv verifier will often warn about it by way of a "given false" message, but this is not guaranteed (rules 2.1 and 2.2)
- eCv does not check rules 2.3, 2.4, 2.5, 2.6, 2.7
- eCv does not check rule 3.2
- eCv does not check rule 4.1
- eCv does not check rules 5.1, 5.2
- eCv does not check rule 5.4. The eCv preprocessor considers all characters in a macro name to be significant.
- eCv does not check rules 5.5, 5.8, 5.9
- eCv does not check rule 7.1
- eCv does not check rules 8.4, 8.5, 8.6, 8.7, 8.9, 8.10, 8.12, 8.13, 8.14
- eCv does not support C99 designated initializers, therefore rules 9.4, 9.5 are not applicable
- eCv does not check rules 10.6, 10.7, 10.8
-

- eCv does not check rule 11.9
- eCv enforces rule 12.1 only partially, for example in most cases it does not warn if an operand of a Boolean operator is not parenthesized
- eCv does not check rule 12.3
- eCv does not check rules 13.3, 13.4, 13.5
- eCv does not check rules 14.1, 14.2, 14.3
- eCv does not check rules 15.1, 15.4, 15.5
- eCv does not fully enforce rule 16.3 because it allows a switch-case to be terminated by a return- or continue-statement
- eCv does not fully enforce rule 17.2 because it allows a function to be directly recursive if a recursion variant was specified, and does not check for indirect calls
- eCv does not check rules 17.7, 17.8
- eCv allows and understands pointer arithmetic (rule 18.4)
- eCv does not check rules 18.5, 18.6
- eCv does not check rule 19.2, however it requires that unions are used only for the purpose of storing different values at different times, not for performing type conversions
- eCv does not check rules 20.1, 20.2, 20.7, 20.8, 20.10, 20.11, 20.12
- eCv checks rule 21.1 in respect of #define but not in respect of #undef
- eCv does not check the remaining rules in section 21
- eCv does not check the section 22 rules

#### Notes:

1. The commenting rules (3.x) and preprocessor rules (20.x) are only checked if you are using eCv's own preprocessor, not if you are using your compiler's preprocessor.
2. Compliance with some rules can only be ensured if verification is run and the verification conditions corresponding to these rules are successfully discharged by the theorem prover.

## eCv rules that are stronger than the MISRA-C 2012 guidelines

- The essential type of an enumeration constant or a variable of enumeration type is always that type (not **int**), even if the type is an anonymous enum type.
- Type **wchar\_t** is treated as a separate type by eCv, not as a **typedef** for some integral type. It is treated in a similar way to plain **char**, i.e. all conversions between **wchar\_t** and other types must be explicit.
- Pointers to arrays must be qualified by the **array** keyword when they are declared.
- Pointers that are allowed to take the null pointer value must be qualified by the **null** keyword when they are declared.
- eCv will warn about conversions between different pointer types, except for conversions to **void\***.
- eCv does not allow conditional expressions to have side effects.
- eCv does not allow you to take the address of a member of a union, or of any part of a member of a union.
- eCv does not allow you to use the **sizeof** operator with a character literal operand.





# Appendix G

## Main differences between eCv in C'90 or C'99 mode and MISRA-C

Here are lists of the main differences between the C subset accepted by eCv and MISRA-C. Numbers in brackets are the corresponding MISRA rule numbers.

### MISRA-C 2004 guidelines not fully enforced by eCv

The MISRA-C 2004 standard has 141 rules constraining how the C language may be used. Compliance with about 128 of these rules can in principle be checked by source code analysis, with formal verification needed in the case of a few rules. eCv currently checks 55 MISRA-C 2004 rules in full and a further 16 rules in part. Future versions of eCv may support additional MISRA compliance checks.

Here are the MISRA-C 2004 rules that eCv allows to be violated without generating a warning or error message, when the compiler type is set to C90 and MISRA-C 2004 checking has been enabled in the project settings.

- eCv does not fully enforce rule 1.1 because it supports a few language extensions, such as **inline** functions, **long long int** type, and the syntax used by many embedded C compilers to place variables at particular addresses.
- Rules 1.3, 1.4 and 1.5 are outside the scope of source code static checking.
- eCv does not enforce rule 2.1.
- eCv permits comments introduced by // (2.2).
- eCv does not enforce rule 2.4.
- The documentation rules (3.x) are outside the scope of source code static checking. However, the behaviour of integer division is included when specifying the compiler/platform behaviour (3.3).
- eCv does not enforce rules 5.4, 5.5, 5.6 and 5.7.
- eCv enforces rule 6.1 and 6.2 to the extent that conversions in either direction between other integral types and plain char must be made explicit.
- eCv does not enforce rule 6.3.
- eCv does not require prototypes for functions whose definitions are visible at each place where they are called (8.1).
- eCv permits objects to be defined in header files, in particular inline function declarations (8.5).
- eCv does not enforce rules 8.6, 8.7, 8.8, 8.9, 8.10, 8.11 or 8.12.
- eCv does not enforce rule 9.3.
- eCv enforces part (a) of rules 10.1 and 10.2 but not parts (b), (c) or (d).
- eCv does not enforce rules 10.3, 10.4 or 10.5.
- eCv does not enforce rules 12.1, 12.4, 12.5 or 12.10.
- eCv cannot enforce rule 12.11 because it depends on detailed knowledge of the behaviour of the compiler you will be using. Wrap-around will not occur when using the eCv preprocessor.
- eCv does not enforce rule 12.13.
- eCv allows assignment-expressions to be used within expressions yielding a Boolean value (13.1), provided

that an assignment-expression is not used where a Boolean value is required. For example, "(a = b) && c" is forbidden, but "(a = b) == 0" is permitted.

- eCv allows floating point expressions to be tested for equality and inequality (13.3).
- eCv treats a for-loop as if it were an equivalent while-loop, therefore it does not place additional restrictions on the expressions in a for-loop header (13.4, 13.5) or on modifying for-loop variables (13.6).
- eCv does not prohibit Boolean operations whose results are invariant, however they will typically lead to "given false" warnings during verification, indicating unreachable code (13.7).
- eCv does not guarantee to detect unreachable code (14.1), however it will typically lead to "given false" warnings during verification.
- eCv does not check that a non-null statement either has a side effect or causes control flow to change (14.2).
- eCv does not check rule 14.3. You can use **pass** to introduce a null statement explicitly.
- eCv supports **continue** (14.5).
- eCv allows multiple **break** statements in a loop (14.6).
- eCv allows multiple **return** statements in a function (14.7).
- A **case** clause in a **switch** statement need not end in "**break**;" if it cannot fall through, for example if it ends in an if-statement for which both branches end in a jump statement (15.2).
- eCv only allows recursive functions if a recursion variant is declared (16.2).
- eCv does not enforce rule 16.3.
- eCv treats a function or function prototype declared with an empty parameter list as having no parameters; it does not insist on the parameter list being declared as "(void)" (16.5).
- eCv does not warn about pointer parameters that could have been declared **const** but were not (16.7).
- eCv does not enforce rule 16.9.
- eCv does not check that error information returned by a function is tested, provided the specification is met (16.10).
- eCv permits pointer arithmetic on array pointers (17.4), however verification of pointer arithmetic typically requires additional annotation, therefore pointer arithmetic should be avoided as far as possible.
- eCv does not place a limit on pointer indirection, however compliance with the MISRA limit of 2 is recommended (17.5).
- eCv does not enforce rule 17.6.
- eCv permits unions, provided they are not used to convert between different types (18.4).
- eCv does not enforce preprocessing rules 19.1, 19.2, 19.4, 19.5, 19.7, 19.9, 19.10, 19.13 or 19.15.
- eCv partially enforces rule 20.1 but does not enforce the remaining library rules (20.x).

#### Notes:

1. The commenting rules (2.x) and preprocessor rules (19.x) are only checked if you are using eCv's own preprocessor, not if you are using your compiler's preprocessor.
2. Compliance with rules 1.2, 12.2, 12.8, 17.2, 17.3 and 21.1 can only be ensured if verification is run and the verification conditions corresponding to these rules are successfully discharged by the theorem prover.

## eCv rules that are stronger than the MISRA-C 2004 guidelines

- eCv has a Boolean type **bool**. The underlying type of an operator expression in which the operator is a comparison or logical operator is **bool**. Implicit conversions between **bool** and other types other than 1-bit unsigned int bit fields generate warnings.

- eCv has strongly-typed enumerations. The underlying type of an enumeration constant or a variable of enumeration type is that type, not **int**. Implicit conversions from enumeration types to **int**, **long int** and **long long int** do not generate warnings. Implicit conversions from integral to enumeration types *do* generate warnings.
- Type **wchar\_t** is treated as a separate type by eCv, not as a **typedef** for some integral type. It is treated in a similar way to plain **char**, i.e. all conversions between **wchar\_t** and other types must be explicit.
- Pointers to arrays must be qualified by the **array** keyword when they are declared.
- Pointers that are allowed to take the null pointer value must be qualified by the **null** keyword when they are declared.
- eCv will warn about conversions between different pointer types, except for conversions to **void\***.
- eCv does not allow any part of a conditional expression to have side effects.
- eCv does not allow you to take the address of a member of a union, or of any part of a member of a union.
- eCv does not allow you to use the **sizeof** operator with a character literal operand.

## MISRA-C 2012 guidelines not fully enforced by eCv

The MISRA-C 2012 standard has 16 directives and 143 rules constraining how the C language may be used. Compliance with most rules can in principle be checked by source code analysis, with formal verification needed in some cases. Compliance with some of the directives can also be partially checked by source code analysis and formal verification, although in most cases additional information is needed. eCv currently checks 68 MISRA-C 2012 rules in full, and a further 6 rules and one directive in part. Future versions of eCv may support additional MISRA compliance checks.

Here are the MISRA-C 2012 directives and rules that eCv allows to be violated without generating a warning or error message, when the compiler type is set to C90 or C99 and MISRA-C 2012 checking has been enabled in the project settings.

- Directives 1.1, 2.1, 3.1, 4.2 are outside the scope of formal verification
- The eCv verifier enforces directive 4.1 in respect of integer arithmetic overflow, pointer arithmetic, array bound errors, function parameters (provided the functions are specified with the correct preconditions) and pointer dereferencing. It does not verify absence of overflow of floating point computations. eCv does not support dynamic memory.
- eCv enforces rule 1.1 in respect of syntax (apart from the extensions supported by eCv) and constraints, but not in respect of implementation limits.
- eCv supports a small number of common language extensions (rule 1.2)
- When code is unreachable, the eCv verifier will often warn about it by way of a "given false" message, but this is not guaranteed (rules 2.1 and 2.2)
- eCv does not check rules 2.3, 2.4, 2.5, 2.6, 2.7
- eCv does not check rule 3.2
- eCv does not check rule 4.1
- eCv does not check rules 5.1, 5.2
- eCv does not check rule 5.4. The eCv preprocessor considers all characters in a macro name to be significant.
- eCv does not check rules 5.5, 5.8, 5.9
- eCv does not check rule 7.1
- eCv does not check rules 8.4, 8.5, 8.6, 8.7, 8.9, 8.10, 8.12, 8.13, 8.14
- eCv does not support C99 designated initializers, therefore rules 9.4, 9.5 are not applicable
- eCv does not check rules 10.6, 10.7, 10.8
-

- eCv does not check rule 11.9
- eCv enforces rule 12.1 only partially, for example in most cases it does not warn if an operand of a Boolean operator is not parenthesized
- eCv does not check rule 12.3
- eCv does not check rules 13.3, 13.4, 13.5
- eCv does not check rules 14.1, 14.2, 14.3
- eCv does not check rules 15.1, 15.4, 15.5
- eCv does not fully enforce rule 16.3 because it allows a switch-case to be terminated by a return- or continue-statement
- eCv does not fully enforce rule 17.2 because it allows a function to be directly recursive if a recursion variant was specified, and does not check for indirect calls
- eCv does not check rules 17.7, 17.8
- eCv allows and understands pointer arithmetic (rule 18.4)
- eCv does not check rules 18.5, 18.6
- eCv does not check rule 19.2, however it requires that unions are used only for the purpose of storing different values at different times, not for performing type conversions
- eCv does not check rules 20.1, 20.2, 20.7, 20.8, 20.10, 20.11, 20.12
- eCv checks rule 21.1 in respect of #define but not in respect of #undef
- eCv does not check the remaining rules in section 21
- eCv does not check the section 22 rules

#### Notes:

1. The commenting rules (3.x) and preprocessor rules (20.x) are only checked if you are using eCv's own preprocessor, not if you are using your compiler's preprocessor.
2. Compliance with some rules can only be ensured if verification is run and the verification conditions corresponding to these rules are successfully discharged by the theorem prover.

## eCv rules that are stronger than the MISRA-C 2012 guidelines

- The essential type of an enumeration constant or a variable of enumeration type is always that type (not **int**), even if the type is an anonymous enum type.
- Type **wchar\_t** is treated as a separate type by eCv, not as a **typedef** for some integral type. It is treated in a similar way to plain **char**, i.e. all conversions between **wchar\_t** and other types must be explicit.
- Pointers to arrays must be qualified by the **array** keyword when they are declared.
- Pointers that are allowed to take the null pointer value must be qualified by the **null** keyword when they are declared.
- eCv will warn about conversions between different pointer types, except for conversions to **void\***.
- eCv does not allow conditional expressions to have side effects.
- eCv does not allow you to take the address of a member of a union, or of any part of a member of a union.
- eCv does not allow you to use the **sizeof** operator with a character literal operand.

