Xcpp Macro Preprocessor

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

Xcpp includes a powerful macro preprocessor. This paper describes the preprocessor directives with example usage.

1 Overview

The xcpp macro preprocessor is applied to the source code before subsequent compilation by a standard C/C++ compiler. The directives always begin with the `@` character. In the code examples presented in this paper red syntax colouring is used for these directives and blue for the standard C++ keywords.

To illustrate the preprocessor self contained code examples are shown before and after translation. The appendices define the grammar using EBNF.

2 @def directive

2.1 Simple macros

A macro can be defined and then referenced any number of times *further down* in the translation unit:

```
@def m = 100

// This is a comment
int f() { return m+m/2; }
```

Before translation

```
// This is a comment
int f() { return 100+100/2; }
```

After translation

We say that macro `m` has been *invoked* twice in the above example. Often a macro is defined in a header file so it can be invoked from multiple source files. We refer to the substitution string `100` as the *body* of `m`.

The translation maintains existing indentation and comments. Macro definitions are stripped away, and all invocations of a macro after the point of definition result in substitution by the body.

2.2 Macro arguments

In similar fashion to C/C++ `#define` macro definitions, xcpp macros can take any number of formal arguments. By default the macro expander performs substitution without regard for the semantics of C/C++ programs. The following example shows how inappropriate usage can lead to surprises because of a lack of bracketing of expressions:

```
@def add(x,y) = x+y
int f() { return 3*add(5,2); }
```

Before translation

```
int f() { return 3*5+2; }
```

After translation

2.3 Macro return types

The return type of a macro can be specified. In the following example it is stipulated that `add(x,y)` returns an `int`. This causes xcpp to perform calculations at compile time in order to coerce the return value.

```
@def int add(x,y) = x+y
int f() { return 3*add(5,2); }
```

Before translation

```
int f() { return 3*7; }
```

After translation

This can have a number of advantages:

- It can avoid the problems with macro expanded expressions needing additional brackets in order to be processed correctly;
- It helps to self document the intent;
- The xcpp compiler validates the type, and provides useful error messages when type checking fails; and
- The generated code is more succinct.

2.4 Macro argument types

The types of the formal arguments of a macro can optionally be specified. This leads to eager evaluation of the arguments by the xcpp preprocessor at the point of invocation. The advantages described above for strong typed return values also apply to strongly typed formal arguments. Here is an example:

```
@def multiply(int x,int y) = x*y
int f() { return multiply(1+2,4); }
```

Before translation

```
int f() { return 3*4; }
```

After translation

The following example illustrates strong typing of both the return value and arguments of a macro:

```
@def int multiply(int x,int y) = x*y
int f() { return multiply(1+2,4); }
```

Before translation

```
int f() { return 12; }
```

After translation

2.5 Rules for delimiting the body

In the directive `@def x = y`, we call `y` the *body*. The body is delimited in a number of different ways. In the following example, macros named `w1`,`w2` and `w3` are all equivalent:

```
@def w1 = 5
@def w2 = {5}
@def w3 =
{
    5
}
int f() { return w1+w2+w3; }
```

Before translation

```
int f() { return 5+5+5; }
```

After translation

The extent of the body is determined before considering its macro expansion. The xcpp preprocessor uses the lexical scanner to help determine the extent of the body. This allows it to ignore braces inside // or /*...*/ comments or in single or double quoted strings. From a position just after the '=', it scans past space and tab characters on that line. If the next character is not a linefeed or left brace then it sets the body to be all the remaining characters on that line (not including the linefeed). Otherwise the body is assumed to be an indented block of text delimited (non-inclusively) by braces. Nested braces are allowed within the block, as long as braces pair up correctly (xcpp counts braces to determine the end of the block). The body doesn't include the final linefeed - i.e. on the last line immediately before the final right brace.

In rare circumstances when there's a need to workaround the counting of braces, `@unstr` can be used (this directive is described in section 6.3). For example:

```
@def leftBrace = @unstr('{')
@def rightBrace = @unstr('}')
@def m = rightBrace@@leftBrace@@leftBrace
const char* f()
{
    {m}}
    return @str(m);
}
```

Before translation

```
const char* f()
{
    {}{{}}
    return "}{{";
}
```

After translation

2.6 Indenting of replacement text

The xcpp preprocessor applies an indentation to an entire block of text under macro substitution. The effect is that the output of the preprocessor is often conveniently formatted. For example:

```
@def m(x,y,b) =
{
    if (x < y)
    {
        return b;
    }
}
@def loop(i,n,body) =
{
    for (int i=0 ; i < n ; ++i)
    {
        body
    }
}
bool f(int* x, int nx, int* y, int ny)
{
    loop(i,nx,loop(j,ny,m(x[i],y[j],true)))
    return false;
}
```

Before translation

```
bool f(int* x, int nx, int* y, int ny)
{
    for (int i=0 ; i < nx ; ++i)
    {
        for (int j=0 ; j < ny ; ++j)
        {
            if (x[i] < y[j])
            {
                return true;
            }
        }
    }
    return false;
}
```

After translation

2.7 Nested macro definitions

Macro definitions can be nested. The following example shows how macro `m` defines a *local* macro called `y`. Local macros are not accessible outside the scope of the containing macro.

```
@def m(x) =
{
    @def y = 2
    x+y
}
int f()
{
    int y = 3;
    return m(1)+y;
}
```

Before translation

```
int f()
{
    int y = 3;
    return 1+2+y;
}
```

After translation

2.8 Back-quoted arguments to macros

An argument to a macro can be back-quoted in order to ensure it is parsed as a single indivisible argument to bind to the formal argument of the macro. For example, it is necessary to

back-quote the argument `map<K,V>` because it contains commas, and the macro expander doesn't count angled brackets to delimit terms when a macro is invoked.

```
@def m(type) = void f(const type&);
template <class K,class V>
m(‘map<K,V>‘)
```

Before translation

```
template <class K,class V>
void f(const map<K,V>&);
```

After translation

3 Printing to stdio

The directive `@print(x)` writes the macro expanded form of `x` to stdio during the execution of the xcpp macro preprocessor. This could for example be used to display warning messages to the programmer. `@println` is the same as `@print` except that a line-feed is written afterwards.

Alternatively, text can be written to stdio using the `@runpython` or `@defpython` directives (these directives are described in section 8):

```
@runpython
{
    print ’Hello, world’
    for i in range(4):
        print i
}
```

4 Aborting the preprocessor

There are many different error conditions that can cause the xcpp preprocessor to abort with some failure indication. Error messages are displayed in a similar form to the Microsoft Visual C++ compiler.

The directives in this section are specifically aimed at aborting the xcpp preprocessor.

4.1 @assert directive

The `@assert(x)` directive macro expands `x` then evaluates it as an expression that must be implicit convertible to `bool`. If false then the xcpp preprocessor aborts with an error message on the command line. The `@assert(x)` directive is stripped from the generated output. i.e. it macro expands into nothing.

4.2 @assertfails directive

This directive macro expands into nothing. It verifies that macro expansion of `x` generates a macro expansion error. The error is written to stdio. The directive aborts the xcpp preprocessor if macro expansion of `x` doesn't generate an error. The only purpose of this directive is to properly unit test the xcpp preprocessor.

```
// Unit tests of xcpp preprocessor

// No conversion from string to bool
@assertfails( @(bool("x")) )

// No conversion from string to int32
@assertfails( @(int("x")) )

// No conversion from multi character
// string to char
@assertfails( @(char("xx")) )

// Invalid argument to log
@assertfails( @(log(0)) )
@assertfails( @(log(-1)) )

// Type mismatch (no implicit conversion
// between bool and int)
@assertfails( @(1 == true) )

// No implicit conversion from int to bool
@assertfails( @if(0){} )
```

4.3 @fail directive

`@fail(x)` aborts the xcpp preprocessor displaying an error message obtained by macro expanding `x` then evaluating as an expression that must be implicit convertible to a string.

5 @if-@else directive

`@if-@else` directives can be used for conditional macro expansion. Informally, the syntax is:

```
@if     (b1) {x1}
@elseif (b2) {x2}
@elseif (b3) {x3}
...
@elseif (bn) {xn}
@else        {y}
```

There may be zero or more `@elseif` directives. The `@else` directive is optional. The `b1,...,bn` are macro expanded before being evaluated as boolean valued expressions. This for example allows for nested `@if-@else` expressions, such as in the following example:

```
@def min(x,y) =
{
    @if(x < y) {x} @else {y}
}
int f() { return min(min(3,7),min(1,6)); }
```

Before translation

```
int f() { return 1; }
```

After translation

6 String conversion directives

Sometimes a macro has bound to some text, and we need to put it in double quotes so it looks like a C/C++ string literal. At other times it can be useful to remove the quotes. `@str` and `@strx` allow for adding the quotes, and `@unstr` allows for removing them.

6.1 @str directive

The `@str(x)` directive expands into a double quoted string obtained by first macro expanding `x` then converting it to a double quoted string. The argument to the `@str` directive must always be enclosed in round brackets. The argument may itself contain round brackets as long as they pair up correctly (the preprocessor counts bracket tokens from a lexical scan to determine the extent of the argument). For example:

```
@def min(x,y) =
{
    @if(x < y) {x} @else {y}
}
const char* f()
{
```

\a	bell
\b	backspace
\f	formfeed
\n	linefeed
\r	carriage return
\t	horizontal tab
\v	vertical tab
\xhh	hexadecimal value
\0	null character
\\	backslash
\'	single quote
\"	double quote

Table 1: Escape characters

```
    return @str(Minimum is min(3,4));
}
```

Before translation

```
const char* f()
{
    return "Minimum is 3";
}
```

After translation

Non-printable characters, backslashes, single and double quote characters and so forth are escaped as required for string literals in C/C++ (see Table 1). For example:

```
@def a = @unstr('\a')
@def b = @unstr('\b')
@def f = @unstr('\f')
@def n = @unstr('\n')
@def r = @unstr('\r')
@def t = @unstr('\t')
@def v = @unstr('\v')
@def ff = @unstr('\xff')
@def z = @unstr('\0')
const char* s =
  @str(a b f n r t v ff z '\0' "w");
```

Before translation

```
const char* s =
  "\a \b \f \n \r \t \v \xff \0 \'\\0\' \"w\"";
```

After translation

If the `@str` token and the subsequent left bracket token appear on the same line then all white space characters between the containing brackets are significant - i.e. are assumed to be part of the text to be represented as a C/C++ literal. For example:

```
const char* s = @str( 1

  2   );
```

Before translation

```
const char* s = " 1\n\n  2   ";
```

After translation

Otherwise if the `@str` token and the subsequent left bracket token appear on different lines then the text to be converted is assumed to be an indented block. The position of the first non-whitespace character defines the indent position of the block. The block is converted to a string literal without the white space characters to the left of this indent position and without the leading and trailing linefeeds (i.e. that appear just after the opening bracket and just before the closing bracket). Note that trailing white space characters on a given line are significant. For example:

```
const char* s =
  @str
  (
        abc
         def

        ghi
  );
```

Before translation

```
const char* s =
  "abc\n def\n\nghi";
```

After translation

6.2 @strx directive

The directive `@strx` is the same as `@str` except that linefeeds cause the string to be broken up into separate strings. This is useful given that the C/C++ compiler concatenates adjacent string literals.

The strings are formatted into a nicely tabbed block of code ready to be compiled by the C/C++ compiler.

```
@def min(x,y) =
{
    @if(x < y) {x} @else {y}
}
const char* f()
{
    static const char* s =
        @strx
        (
            This is a sentence
            stating that the
            minimum of 3,4 is
            min(3,4)
        );
    return s;
}
```

Before translation

```
int f()
{
    static const char* s =
        "This is a sentence\n"
        "stating that the\n"
        "minimum of 3,4 is\n"
        "3";
}
```

After translation

6.3 @unstr directive

To process `@unstr(x)`, `x` is macro expanded and this is assumed to produce a valid single or double quoted string. `@unstr(x)` macro expands into a version of the string with the quotes removed. Also characters that were escaped are "un-escaped". E.g. `\n` is replaced by a real linefeed character.

```
@unstr("//") This is a C++ style comment!
```

Before translation

```
// This is a C++ style comment!
```

After translation

7 Scope

7.1 Namespace stack

Consider a procedural execution model of the xcpp preprocessor. A file is typically translated by processing its text from start to finish. Typically text is processed by copying it verbatim from input to output.

C/C++ comments are copied verbatim without further processing. It follows that `@def` directives can be commented out easily. E.g.

```
@def m = 1
//@def m = 2
int f() { return m; }
```

Before translation

```
//@def m = 2
int f() { return 1; }
```

After translation

Note that the comment appeared in the output. To strip a comment from the output, precede the C/C++ comment with `@`. i.e. use `@//...` or `@/*...*/`. For example:

```
@// This comment is stripped from the output
@def m = 1

int f() { return m; }
```

Before translation

```
int f() { return 1; }
```

After translation

The xcpp preprocessor uses a stack of namespaces to record macro definitions. Macro names are looked up by searching each namespace in turn, starting from the top of the stack. Therefore names in an inner scope hide names in an outer scope. When translation first begins at the top of the file, the stack is initialised with a single entry, which is the global namespace for macro definitions. Note therefore that macros at the outermost scope are recorded in the global namespace.

When a `@def` directive is processed the macro is recorded in the namespace at the top of the stack. The body of a macro is skipped over without being processed. Therefore, at this time the nested `@def` directives are ignored. Processing the body of a macro only occurs when the macro is invoked (if ever).

When a macro is invoked a new namespace is pushed onto the top of the stack in preparation for processing the body of the macro. It is popped when the processing of the body is completed. The effect is that the execution of the macro creates a local namespace for nested `@def` directives.

7.2 Macros are expanded in the context of the caller

A macro is expanded in the context of the stack of namespaces defined at the point of invocation, not the point of definition. In the following example, `m` is *invoked* in a context where `n = 2`. The fact that `n = 1` at the point of definition of `m` is irrelevant.

```
@def n = 1
@def m = n
@def n = 2
int f() { return m; }
@def n = 3
int g() { return m; }
```

Before translation

```
int f() { return 2; }
int g() { return 3; }
```

After translation

This concept of expanding a macro in the *context of the caller* (and not the callee) makes it possible to write very flexible macros without the need to explicitly parameterise with large numbers of formal arguments.

7.3 @nakeddef directive

The `@nakeddef` directive is the same as the `@def` directive except that when the macro is invoked no local namespace is pushed onto the namespace stack. The effect is that its local, nested macro definitions are added to the namespace of the caller. For example:

```
@nakeddef m1 =
{
    @def n = 1
}
@nakeddef m2 =
{
    @def n = 2
}

m1
int f() { return n; }

m2
int g() { return n; }
```

Before translation

```
int f() { return 1; }
int g() { return 2; }
```

After translation

7.4 Local scope in directives

Most directives introduce private namespaces for local macros. For example, local macros can be defined in either the boolean condition or the body of an `@if` directive, or within a `@(...)` directive:

```
@def int a = 1
@if
(
    @def a = true
    a
)
{
    @def b = 2
    int c = 10;
    int f() { return @(@def c = {3} a+b+c); }
}
```

Before translation

```
int c = 10;
int f() { return 6; }
```

After translation

7.5 @scope directive

The `@scope{x}` directive defines a local scope for macro definitions. `x` is macro expanded as if no `@scope` directive was defined. This is useful for limiting the scope of macros, to avoid accidental invocation outside their intended usage.

```
@scope
{
    @def m(T) =
    {
        T min(T x1, T x2)
        {
            return x1 < x2 ? x1 : x2;
        }
    }
    m(int)
    m(double)
}
int m(1);
```

Before translation

```
int min(int x1, int x2)
{
    return x1 < x2 ? x1 : x2;
}
double min(double x1, double x2)
{
    return x1 < x2 ? x1 : x2;
}
int m(1);
```

After translation

8 Python directives

The directives `@runpython` and `@defpython` provide access to a Python interpreter. The intention is for complex macros to be defined using Python, which represents a well supported and well documented language, allowing the xcpp preprocessor to be simpler.

8.1 @runpython directive

The `@runpython(x)` directive macro expands `x` then runs it under the python interpreter. The directive itself is stripped from the generated output. i.e. it macro expands into nothing. The purpose is normally to allow for definitions of functions and variables and so forth within the python interpreter, ready for subsequent use by the `@defpython` directive.

```
@runpython
{
    # This is python!

    # Fibonacci series:
    # the sum of two elements
    # defines the next
    def getfibnumbers(max):
        a,b  =  0,1
        s = []
        while b < max:
            s.append(b)
            a,b =  b,a+b
        return s
}
```

8.2 @defpython directive

Informally the `@defpython` directive is of the form

```
@defpython macroname(a1,...,an) = y
```

It defines a macro (in a similar fashion to the `@def` directive). The substitution string is macro expanded then executed as a python expression that must evaluate to an `int`, `float` or `string`. The `@defpython` directive itself is stripped from the generated output. i.e. it macro expands into nothing.

The following example assumes the previous `@runpython(x)` example has already been executed in order to define the `getfibnumbers` function in the Python interpreter:

```
@defpython getfib(int x) =
{
    # This is python!

    # A python expression that
    # evaluates to a string
    str(getfibnumbers(x))
}
const char* f()
{
    return @str(getfib(6));
}
```

Before translation

```
const char* f()
{
    return "[1, 1, 2, 3, 5]";
}
```

After translation

8.3 Practical examples using Python

Python provides very convenient string handling functions that can readily be made available as macros. For example:

```
@defpython int mStringLength(s) =
{
    len(@str(s))
}

@defpython mGetSubString(s,int i1,int i2) =
{
    (@str(s))[i1:i2]
}

@defpython mGetCharInString(s,int i) =
{
    (@str(s))[i]
}

@defpython mToUpper(s) =
{
    @str(s).upper()
}

@defpython mToLower(s) =
{
    @str(s).lower()
}

@defpython mCapitaliseFirstLetter(s) =
{
    @str(s).capitalize()
}

@defpython mEatLeadingWhitespace(s) =
{
    @str(s).lstrip()
}

@defpython mEatTrailingWhitespace(s) =
{
    @str(s).rstrip()
}

@defpython string mCentreJustify(s,int width) =
{
    @str(s).center(width)
}

@defpython mLeftJustify(s,int width) =
{
    @str(s).ljust(width)
}
```

```
@defpython mRightJustify(s,int width) =
{
    @str(s).rjust(width)
}

const char* f()
{
    return
        @str(mCentreJustify(mToUpper(hello),11));
}
```

Before translation

```
const char* f()
{
    return "   HELLO   ";
}
```

After translation

9 Enabling and disabling macro expansion

The three directives in this section can be used to prevent or force macro expansion.

9.1 @@ directive

The `@@` directive is stripped away from the output. It acts as a delimiter for the lexical scanner. For example putting `@@` in the middle of an identifier causes the lexical scanner to see it as two distinct tokens.

```
@def x = 1
@def y = 2
@def xy = 3
int f()
{
    return x@@y;
}
```

Before translation

```
int f()
{
    return 12;
}
```

After translation

9.2 @quote directive

The `@quote(x)` directive macro expands literally to `x` (i.e. without macro expanding `x`). This is useful when we want to disable macro expansion within some scope.

```
@def min(x,y) =
{
    @if(x < y) {x} @else {y}
}
const char* f()
{
    return @str(@quote(min(3,4)) = min(3,4));
}
```

Before translation

```
const char* f()
{
    return "min(3,4) = 3";
}
```

After translation

9.3 @[] directive

The `@[x]` directive is like the inverse of the `@quote` directive. It outputs the result of macro expanding the result of macro expanding `x` (i.e. macro expansions are applied twice). It follows that `@[@quote(y)]` is equivalent to `y`.

In the example below, the expression `v@@1` has managed to defeat the macro expander (i.e. preventing invocation of `v1`), but the `@[]` directive is able to force the macro substitution to take place anyway.

```
@def v1 = 3
const char* f()
{
    return @str(v@@1 = @[v@@1]);
}
```

Before translation

```
const char* f()
{
    return "v1 = 3";
}
```

After translation

Another example:

```
@def min(x,y) =
{
    @if (x < y) {x}
    @else       {y}
}
@def x1 = 10
@def x2 = 20
int f()
{
    return @[x@@min(1,2)];
}
```

Before translation

```
int f()
{
    return 10;
}
```

After translation

10 Expressions

The xcpp preprocessor allows for evaluation of expressions in a similar manner to the C/C++ compiler. The operators and their precedence are shown in Table 2. Expressions are evaluated in the following circumstances:

- Assignment to a variable in the `@let` directive;
- Initialising a typed formal argument when invoking a macro;
- Calculating the return value when invoking a macro that has a typed return value;
- The `@()` directive (see below); and
- Evaluation of the boolean expression in `@if` directives.

The supported types are `bool`, `int`, `double`, `char` and `string`. Unlike C/C++ there is no implicit conversion from `int` to `bool`.

!	Logical negation
~	Bitwise negation
-	Unary minus
+	Unary plus
^^	Power
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
<<	Bitwise shift left
>>	Bitwise shift right
<	Comparison less-than
<=	Comparison less-than-or-equal-to
>	Comparison greater-than
>=	Comparison greater-than-or-equal-to
==	Comparison equal-to
=!	Comparison not-equal-to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Ternary conditional (if-then-else)

Table 2: Operators

10.1 @() directive

The `@(x)` directive forces compile time evaluation of expressions. Note that `x` is macro expanded before being evaluated as an expression.

```
int A[] =
{
    @(2*2*2*2), @(2^^4), @(1 << 4),
    @(0xabcd & 0xf), @(3 | 5), @(27 % 4),
    @("x" == "y"), @("x" < "y"+'z'),
    @(1<2 && 2<4), @(!(1+1 < 2) ? 10 : 20)
};
```

Before translation

```
int A[] =
{
    16, 16, 16,
    13, 7, 3,
    false, true,
    true, 10,
};
```

After translation

10.2 Special functions

The trigonometric functions `sin`, `cos` and `tan` are available. Also the exponential `exp`, and natural logarithm `log` (base e).

```
@def double e = exp(1)
double f() { return e;  }
```

Before translation

```
double f() { return 2.7182818284590451; }
```

After translation

11 @import directives

The xcpp preprocessor copies C/C++ preprocessor commands such as `#pragma`, `#include` and `#define` verbatim from input to output. In all other respects it ignores these directives (and trusts their processing to the standard C/C++ compiler).

To make the xcpp preprocessor process and access the directives from another file, a `@import` directive must be used. This is essentially the same as C/C++ `#include`, and takes the path to a file to be included at that location in double quotes. In fact an import directive is translated to a corresponding `#include` directive by the xcpp preprocessor in preparation for a standard C/C++ compiler. For example:

```
#include "blah.h"
@import "mydir/myfile.h"
int f() { return 1; }
```

Before translation

```
#include "blah.h"
#include "mydir/myfile.h"
int f() { return 1; }
```

After translation

There is a crucial conceptual distinction between a `#include` and a `@import`. A `#include` indicates that the contents of the file are to be inserted at that location and processed *in that context*. In C/C++ the processing of the file could be influenced by earlier `#define` directives. For example, consider a header file:

```
#ifdef X
typedef int Y;
#else
typedef double Y;
#endif
```

The processing of this file is influenced by whether a `#define X` directive appears before it. Some programmers exploit this capability – by including the same file from different places, causing it to generate variable output.

This practice is rejected in the xcpp preprocessor because its macro expansion capabilities are easily powerful enough to eliminate the need for such techniques, and assuming every file has a consistent translation allows the preprocessor to *cache* the namespace of macro definitions obtained by processing the file. This allows for significant performance gains in the xcpp preprocessor for large projects.

The word *import* is intended to indicate that the directive causes the importing of one namespace into another. The implementation is able to cache namespaces in memory and an import directive avoids the need to physically copy entries from one namespace data structure into another. The effect is that all header files behave like pre-compiled headers.

12 Macro variables

12.1 Macro reassignment

A macro can be redefined (or "reassigned") making it like a variable that changes value during the execution of the xcpp preprocessor. No warnings are emitted by xcpp when a macro is redefined:

```
@def m = "hello world"
const char* f() { return m; }
@def m = 200
int g() { return m; }
```

Before translation

```
const char* f() { return "hello world"; }
int g() { return 200; }
```

After translation

Even though `@def` can be used to redefine a macro, making it like a variable available during the execution of the xcpp processor, it is not generally suitable for this purpose because the macro is always local to the scope in which the `@def` directive appears. In the following example, there is no redefinition of `m`, because the body of the `@if` directive creates a *local scope* for macro definitions:

```
@def m = 1
@if (m > 0)
{
    @def m = m+1
}
int f() { return m; }
```

Before translation

```
int f() { return 1; }
```

After translation

In any case there would be problems with infinite recursion. The following crashes the xcpp preprocessor with a stack overflow because the redefinition of `m` is imposed *before* processing its body (this is intentional because recursion can be very useful – see section 14):

```
// Oops : crashes the xcpp preprocessor!
@def int m = 1
@def int m = m+1
int f() { return m; }
```

12.2 @let directive

To allow macros to be used like variables the `@let` directive can be used. This directive has two syntactic forms. When the type is specified, it introduces a name in that scope. Otherwise, it binds to an existing name by searching upwards through nested scopes for a previously defined variable with that name.

```
@let int m = 0
@let m = m+1
int f() { return m; }
@if (m > 0) @let m = m+1
int g() { return m; }
```

Before translation

```
int f() { return 1; }
int g() { return 2; }
```

After translation

12.2.1 String variables

String literals can be given in single or double quotes (and this has no effect on the meaning). The binary '+' operator can be used to concatenate strings. However, string variables macro-expand without the containing quotes! This leads to some surprises. To get the quotes the `@str` directive must be used.

```
@let string m = "hello" + " "
@let m = @str(m) + 'world'
@let m := m !!!
const char* f() { return @str(m); }
```

Before translation

```
const char* f() { return "hello world!!!"; }
```

After translation

There is a special syntax to assign a string variable to macro-expanded text. The directive `@let x := y` is equivalent to `@let x = @str(y)`.

12.3 @while directive

The `@while` directive allows for simple loops. For example:

```
// Calculate 1 + 2 + ... + n
@def int mSum(int n) =
{
    @let int sum = 0
    @let int i = 1
    @while (i <= n)
    {
        @let sum = sum + i
        @let i = i + 1
    }
    sum
}
@assert( mSum(4) == 10 )

// Generate "1+2+...+n"
@def string mGenerateSum(int n) =
{
    @let string sum := 1
    @let int i = 2
    @while (i <= n)
    {
        @let sum := sum+i
        @let i = i + 1
    }
    @str(sum)
}
@assert(@str(mGenerateSum(4)) == "1+2+3+4")
```

13 @for directive

The `@for` directive allows for simple loops. The body of the directive is repeatedly macro expanded for different values of the loop variable(s), taken in sequence from the *iteration list* which is a comma separate list of values enclosed in square brackets.

A *tuple* of loop variables is supported, by using a comma separated list of identifiers in round brackets, and corresponding tuple values in the iteration list.

The loop variables can be regarded as macros that have local scope over the body of the `@for` directive.

```
@for (i in [1,2,3])
{
    f(i);
}
@for ((i) in [(1),(2),(3)])
{
    f(i);
}
@for ((i,j) in [(1,7),(2,-1),(3,4)])
{
    g(i,j);
}
```

Before translation

```
f(1);f(2);f(3);
f(1);f(2);f(3);
g(1,7);g(2,-1);g(3,4);
```

After translation

The iteration list is initially parsed as a single block of text enclosed in square brackets. This text is then macro expanded

according to the macros defined at the point of invocation of the `@for` directive. Only then is the result parsed as a comma separated list. This approach makes the following example possible:

```
const char* s =
    @strx
    (
        @def makelist(int n) =
        {
            @if (n>1) {makelist(n-1),n}
            @else     {1}
        }
        @for ( i in [makelist(4)] )
        {
            @for ( j in [makelist(i)] )
            {
                i + j = @(i+j)

            }
        }
      );
```

Before translation

```
const char* s =
    "1 + 1 = 2\n"
    "2 + 1 = 3\n"
    "2 + 2 = 4\n"
    "3 + 1 = 4\n"
    "3 + 2 = 5\n"
    "3 + 3 = 6\n"
    "4 + 1 = 5\n"
    "4 + 2 = 6\n"
    "4 + 3 = 7\n"
    "4 + 4 = 8\n"
```

After translation

14 Recursive macros

Macros may call themselves recursively (as long as they terminate). The following example shows how we can use a macro to calculate factorial at compile time. This example makes use of strong typing on both the argument and return type.

```
@def int factorial(int n) =
{
    @if (n <= 1) {1}
    @else        {n*factorial(n-1)}
}
int f()
{
    return factorial(5);
}
```

Before translation

```
int f()
{
    return 120;
}
```

After translation

Now consider that the return type on the factorial macro isn't specified:

```
@def factorial(int n) =
{
    @if (n <= 1) {1}
    @else        {n*factorial(n-1)}
}
int f()
{
    return factorial(5);
}
```

Before translation

```
int f()
{
    return 5*4*3*2*1;
}
```

After translation

If we don't strongly type the argument then we end up with an erroneous definition because of lack of bracketing:

```
@def factorial(n) =
{
    @if (n <= 1) {1}
    @else        {n*factorial(n-1)}
}
int f()
{
    // oops!
    return factorial(5);
}
```

Before translation

```
int f()
{
    // oops!
    return 5*5-1*5-1-1*5-1-1-1*1;
}
```

After translation

15 Example usage of the macro expander

As a practical example, the macro expander has been found very useful for generating test code. Some of the unit test files expand into over 100 000 lines of C++ code in order to fully flex important parts of the ceda core.

Here is a concocted example of the macro expander at work to give some idea of what's available for automated code generation.

```
void UnitTests()
{
    @for ((type,val) in [(int,10), (char,
        'c'), (double, 2.71828)])
    {
        std::cout << @str(Testing with type
            val) << std::endl;
        type x@@type = val;
        RunUnitTest(x@@type);
    }
}
```

Before translation

```
void UnitTests()
{
    std::cout << "Testing with int 10" <<
        std::endl;
    int xint = 10;
    RunUnitTest(xint);

    std::cout << "Testing with char \'c\'" <<
        std::endl;
    char xchar = 'c';
    RunUnitTest(xchar);

    std::cout << "Testing with double 2.71828"
        << std::endl;
    double xdouble = 2.71828;
    RunUnitTest(xdouble);
}
```

After translation

A Xcpp preprocessor grammar

We use [1] to define the EBNF syntax.

A.1 Lexical scanner

Let `printableChar` denote a printable ASCII character. The following describes the lexical scanner (with the caveat that white space and comment processing hasn't been described – this is done as normal for C/C++).

```
letter =
  'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
  'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
  'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
  'V' | 'W' | 'X' | 'Y' | 'Z' |
  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
  'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
  'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
  'v' | 'w' | 'x' | 'y' | 'z';

digit =
  '0' | '1' | '2' | '3' | '4' |
  '5' | '6' | '7' | '8' | '9';

identifier =
  ('_' | letter), {'_' | letter | digit};

boolLiteral = 'false' | 'true';

digSeq = digit, {digit};

integerLiteral = digSeq;

exp = ('E'|'e'), ['+'|'-'], digSeq;

floatLiteral =
  digSeq, exp |
  (digSeq, '.' | [digSeq], '.', digSeq), [exp];

hexDigit = digit |
  'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
  'a' | 'b' | 'c' | 'd' | 'e' | 'f';

hexLiteral = '0x', hexDigit, {hexDigit};

escapeChar =
  'r' | 'n' | 'a' | 'b' | 'f' | 't' |
  'v' | 'O' | '"' | "'" | '\' | '?';

stringChar =
  printableChar - escapeChar |
  '\', escapeChar;

stringLiteral =
  '"', {stringChar}, '"' |
  "'", {stringChar}, "'";
```

Grammar for lexical scanner

A.2 Expressions

The xcpp preprocessor must evaluate expressions under various circumstances. For example, in coercions for typed arguments to macros, typed return values from macros, in the `@()` directive, and the boolean expressions used in `@if-@else` directives.

```
expression = assignmentExpr

assignmentOp =
  '=' | '&=' | '|=' | '^=' | '<<=' | '>>=' |
  '&&=' | '||=' | '^^=' |
  '+=' | '-=' | '*=' | '/=' | '%=';

assignmentExpr =
  {varRef, assignmentOp}, conditionalExpr;

varRef = unaryExpr;

conditionalExpr =
  logicalOrExpr, { '?', expression, ':',
      logicalOrExpr };

logicalOrExpr =
  logicalAndExpr, { '||', logicalAndExpr };

logicalAndExpr =
  bitwiseOrExpr, { '&&', bitwiseOrExpr };

bitwiseOrExpr =
  bitwiseXorExpr, { '|', bitwiseXorExpr };

bitwiseXorExpr =
  bitwiseAndExpr, { '^', bitwiseAndExpr };

bitwiseAndExpr =
  equalityExpr, { '&', equalityExpr };

equalityExpr =
  relationalExpr, { ('==' | '!='),
      relationalExpr };

relationalExpr =
  shiftExpr, { ('<' | '<=' | '>' | '>='),
      shiftExpr };

shiftExpr =
  additiveExpr, { ('>>' | '<<'), additiveExpr
      };

additiveExpr =
  multExpr, { ('+' | '-'), multExpr };

multExpr =
  powExpr, { ('*' | '/' | '%'), powExpr };

powExpr =
  unaryExpr, { '^^', unaryExpr };

unaryExpr =
  {'+' | '-' | '~' | '!'}, postfixExpr;

unaryFnName =
  'len' |
  'bool' |  'int' | 'double' |
  'char' | 'string' |
  'is_bool' | 'is_int' | 'is_double' |
  'is_char' | 'is_string' |
  'sin' | 'cos' | 'tan' | 'exp' | 'log';

postfixExpr =
  primaryExpr, { '[', expression, ']' };

literal =
  boolLiteral |
  integerLiteral |
  hexLiteral |
  floatLiteral |
  stringLiteral;

primaryExpr =
  identifier - unaryFnName |
  unaryFnName, '(', expression, ')' |
  literal |
  '(', expression, ')';
```

Grammar for expressions

A.3 Grammar of @def directive

```
type =
  'bool' | 'int' | 'double' |
  'char' | 'string';

argType = type;
argName = identifier;
```

```
  arg = [argType], argName;

  returnType = type;
  macroName = identifier;

  defDirective =
    '@def', [returnType], macroName,
    ['(', [arg, { ',', arg, ')' }], ')'],
    '=',
    value;
```

Grammar for @def directive

More informally we could define the syntax as follows:

```
@def [return-type] x [([type] a1,...,[type] an)] = y
```

This defines a macro named `x`. The directive itself expands into nothing in the output. `x` must be a C/C++ style identifier. The macro named `x` is added to the local namespace. This may replace an existing definition of `x` in the local namespace. There is no support for overloading of macros - even on arity. The body of a macro introduces a new scope (i.e. namespace). This namespace contains the names of the formal arguments of that macro.

The return type is optional. If given it must be `bool`, `int`, `double`, `char` or `string`. If no return type is provided then when `x` is invoked, `x` macro expands into the result of macro expanding `y`. Otherwise, if a return type is provided then after macro expanding `y`, it is evaluated as an expression that must be implicit convertible to the return type. The invocation of `x` is then substituted by the string representation of the evaluated result.

The argument list is optional. Each formal argument must be an identifier. Formal argument names cannot be repeated. A formal argument can optionally be preceded by a type which must be `bool`, `char`, `int`, `double` or `string`. When a formal argument is not typed, the formal argument binds directly to the text provided in the invocation of the macro. Otherwise, when a formal argument is typed, the text provided for the formal argument in the invocation of the macro is first macro expanded then evaluated as an expression that must be implicit convertible to the type of the formal argument. The formal argument binds to the string representation of the evaluated result.

References

[1] ISO/IEC 14977:1996(E). *Information technology - Syntactic metalanguage - Extended BNF*, First edition 1996-12-15. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.