What is the expression language used by the Resource Compiler for non-preprocessor expressions?

devblogs.microsoft.com/oldnewthing/20230313-00

March 13, 2023

Raymond Chen

I noted some time ago that the Resource Compiler's preprocessor is not the same as the C preprocessor: Although it supports the same expression language, it does not support directives like `#pragma`.

There is a second expression language used by the Resource Compiler, and that's the one used to define resources. Surprisingly, this expression language is different from the preprocessor expression language.

For one thing, it has a much reduced set of operators, and no operator precedence. All binary operators are left-associative, and parentheses can be used for grouping.

`+` *a*	Unary plus
`-` *a*	Unary minus
`~` *a*	Unary bitwise negation
a `+` *b*	Binary addition
a `-` *b*	Binary subtraction
a `|` *b*	Binary bitwise OR
a `&` *b*	Binary bitwise AND
a `| NOT` *b*	Binary bit clear (*a* `& ~` *b*)
`0x` *dddd*	Hexadecimal constant
`0o` *dddd*	Octal constant
dddd	Decimal constant

The `NOT` operator looks weird, but the idea is that you can write something like

```
WS_OVERLAPPEDWINDOW | NOT WS_MINIMIZEBOX
```

The `WS_OVERLAPPEDWINDOW` style is a composite style:

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED     | \
                             WS_CAPTION        | \
                             WS_SYSMENU        | \
                             WS_THICKFRAME     | \
                             WS_MINIMIZEBOX    | \
                             WS_MAXIMIZEBOX)
```

Appending a `| NOT WS_MINIMIZEBOX` means that you want all of the styles that come with `WS_OVERLAPPEDWINDOW` except for `WS_MINIMIZEBOX`.

Note that all operators have equal precedence, so `1 | 1 - 1` parses as `(1 | 1) - 1`, which is 0. On the other hand, in C, the `-` operator has higher precedence than `|`, so the C expression `1 | 1 - 1` parses as `1 | (1 - 1)`, which is 1.

Since the preprocessor directives follow C expression rules, you can get into an odd situation where the same expression comes out to a difference value depending on the context in which you use it:

```
#if 1 | 1 - 1
DLG_MYDIALOG DIALOG 1 | 1 - 1, 0, 42, 42
BEGIN
END
#endif
```

The `1 | 1 - 1` in the `#if` statement is a preprocessor directive, and it evaluates to 1, so the `#if` condition is truthy, and the contents generate a dialog resource. On the other hand, when the compiler sees the `1 | 1 - 1` in the `DIALOG` statement, it is evaluated as a resource constant, which produces zero.

Note also that the resource expression language lacks many operators, most notably multiplication.

Another oddity is that the resource expression language uses `0o` as the octal prefix rather than a simple `0` like in C. This means that `010` evaluates to 8 when used in a preprocessor expression, but it evaluates to 10 when used in a resource expression. This is particularly troublesome if you put the definition in a header file that is shared with C/C++ code:

```
// resource.h
#define DLG_AWESOME 010

// contoso.rc
#include "resource.h"

DLG_AWESOME DIALOG ...

// contoso.cpp
#include "resource.h"

DialogBox(instance, MAKEINTRESOURCE(DLG_AWESOME), ...);
```

Even though they are both using `DLG_AWESOME`, the resource compiler sees it as a resource expression, and the `010` is the decimal constant 10. On the other hand, the C++ compiler sees it as as a C++ expression, and the `010` is an octal constant which evaluates to 8.

Moral of the story: Don't use leading zeroes.

There's another quirk of the resource compiler expression language, and that's the concept of the "initial value".

In many cases, the value you are specifying comes with an initial value, and the expression you provide is implicitly treated as combining with the initial value as if by logical "or" (`|`). In other words, if you write some expression `e`, the Resource Compiler acts as if you had actually written

```
initial | e
```

Note the absence of parentheses around the `e`.

Here is the table of initial values for various control types:

Control	Underlying control	With initial style	
`PUSHBUTTON`	Button	`BS_PUSHBUTTON`	`| WS_TABSTOP`
`DEFPUSHBUTTON`	Button	`BS_DEFPUSHBUTTON`	`| WS_TABSTOP`
`CHECKBOX`	Button	`BS_CHECKBOX`	`| WS_TABSTOP`
`AUTOCHECKBOX`	Button	`BS_AUTOCHECKBOX`	`| WS_TABSTOP`
`STATE3`	Button	`BS_3STATE`	`| WS_TABSTOP`
`AUTO3STATE`	Button	`BS_AUTO3STATE`	`| WS_TABSTOP`
`RADIOBUTTON`	Button	`BS_RADIOBUTTON`	

`AUTORADIOBUTTON`	Button	`BS_AUTORADIOBUTTON`	
`GROUPBOX`	Button	`BS_GROUPBOX`	
`PUSHBOX`	Button	`BS_PUSHBOX`	`| WS_TABSTOP`
`LTEXT`	Static	`SS_LEFT`	`| WS_GROUP`
`RTEXT`	Static	`SS_RIGHT`	`| WS_GROUP`
`CTEXT`	Static	`SS_CENTER`	`| WS_GROUP`
`ICON`	Static	`SS_ICON`	
`EDITTEXT`	Edit	`ES_LEFT | WS_BORDER`	`| WS_TABSTOP`
`LISTBOX`	ListBox	`LBS_NOTIFY | WS_BORDER`	

These initial styles are documented in the Resource Compiler documentation. For each control, look at the section of the documentation that says "If you do not specify a style, the default style is…" That default style is the value that your custom style is combined with.

It didn't fit in the above table, but dialogs themselves have an initial style of `WS_CAPTION` if you gave the dialog a `CAPTION`.

The "initial value" trick makes the `NOT` behavior a little more comprehensible, because it lets you write things like

```
    LTEXT "Hello", -1, 4, 80, 160, 10, NOT WS_GROUP | SS_SUNKEN
```

The `NOT WS_GROUP` combines with the initial value to produce a net result of

```
SS_LEFT | WS_GROUP | NOT WS_GROUP | SS_SUNKEN
```

Translating this into C, you get

```
((SS_LEFT | WS_GROUP) & ~WS_GROUP) | SS_SUNKEN
```

which is `SS_LEFT | SS_SUNKEN`. The `NOT WS_GROUP` removes the `WS_GROUP` style that came by default with the `LTEXT` statement.

Bonus chatter: Why are there two different expression languages?

The resource language is designed for use in resource statements. The `NOT` syntax, in particular, is handy for removing styles from the initial styles that come with certain controls.

The preprocessor language is designed for use in preprocessor statements. And the preprocessor language matches the C programming language because preprocessor statements commonly occur in header files which are consumed by both the Resource

Compiler and the C/C++ compiler. It would be a very difficult bug to track down if a `#if` statement in a header file evaluated differently depending on whether you included it from a C/C++ file or from a resource file.