N2657: **Outer**

Javier A. Múgica - javier@aerotri.es

February 4th, 2021

Problem

It is very difficult, if not impossible, to declare an identifier to be used in a macro in such a way that it will surely not conflict with some existing identifier:

```
#define copystr8(x, y) do{ \
        char *px = x; const char *py = y; \
        while(*py != '\0') *px++ = *py++; *px = '\0'; \
}while(0)


const char s1[] = "String 1";
char s2[20], px[20];

copystr8(s2, s1); // OK
copystr8(px, s1); // Wrong
```

This has always been a problem, and now that tools for generic programming are being proposed (e.g.: typeof, or the document N2638) a solution for it seems more necessary.

Proposed solution

To introduce the *identifier-resolver* `_Outer`. Its intended use is as in this example:

```
#define copystr8(x, y) do{ \
        char *px = _Outer (x); \
        const char *py = _Outer (y); \
        while(*py != '\0') *px++ = *py++; *px = '\0'; \
}while(0)
```

So that after a call like `copystr8(px, s1);` the expansion of the line declaring `px` would be

```
        char *px = _Outer (px);
```

and the inner `px` gets initialized properly.

Because the arguments passed to macros can be more or less complicated text it is necessary that the syntax allows expression like `_Outer (ptr + pos[n])`, say, where the identifier resolver should apply to all identifiers within. Furthermore, constructions like `_Outer _Outer ( _Outer ptr + pos[n] )`, which may arise from nested macro invocations should also be valid.

The “Example 1” below uses the operator typeof(), though it is not part of the language yet, in the expectation that it will be added.

The additions to the current text of the standard appear in blue (except for the syntax hilighting of the examples). This document also incorporates a proposal for fixing the first paragraph in the “Semantics” section of 6.5.1. In order to keep the two proposals clearly apart, the additions because of the latter are written in green, and deletions in ~~red~~. (The current proposal requires no deletions beyond a trivial one at the very end due to section renumbering).

Proposed wording

6.4.1. Primary expressions

Syntax

(Add _Outer *to the list of keywords)*

6.5.1. Primary expressions

Syntax

primary-expression:
semantic-identifier
identifier-resolver-selection
constant
string-literal
(*expression* **)**
generic-selection

Semantics

A semantic identifier which does not evaluate to a constant is a primary expression, provided it resolves to an identifier which has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[132)] An undeclared identifier is a violation of the syntax.

[...]

[132)] ~~Thus, an undeclared identifier is a violation of the syntax.~~

6.5.1.1. Semantic identifier and identifier resolver selections

Syntax

semantic-identifier:
[*identifier-resolver-seq*]$_{opt}$ *identifier*
identifier-resolver-seq:
_Outer [*identifier-resolver-seq*]$_{opt}$
identifier-resolver-selection:
identifier-resolver-seq **(***expression***)**

Constrains

A semantic identifier in which the identifier-resolver sequence is not empty is a *modified identifier*. A modified identifier may appear wherever an ordinary identifier (6.2.3) may, except when such appearance would constitute its declaration.

Semantics

In a construction of the form

```
_Outer sem-id
```

the *identifier resolver* _Outer may modify the entity the semantic identifier *sem-id* refers to. It makes it refer to the entity it would have referred to just before the beginning of the innermost scope. Therefore, if *sem-id* still includes identifier resolvers the rule applies recursively. If it be required to determine the entity a modified identifier refers to at file scope the behaviour is as if an undeclared identifier had appeared at the place the original modified identifier (the one that caused the requirement to evaluate a modified identifier at file scope) is present.

As a result of this process the modified identifier is said to *resolve* to an identifier.

An identifier-resolver selection *identifier-resolver-seq* **(***expression***)** evaluates to **(** *expression2* **)**, where *expression2* is the same as *expression* except that:

i) All identifier-resolver selections present in *expression* have *identifier-resolver-seq* prepended.

ii) All semantic identifiers in *expression* that appear at places where a modified identifier is allowed and which are not within the scope of an identifier-resolver selection have *identifier-resolver-seq* prepended.

EXAMPLE 1

```
#define DIR_SEP '/'
#define remove_filename(x) do{ \
        typeof(*(x)) *px = _Outer(x); while(*px != '\0') px++; \
        while(px != _Outer(x) && *px != DIR_SEP) px--; \
        if(*px == DIR_SEP) px[1] = '\0'; \
}while(0)

/* Suppose str points to a block of memory where several strings
are stored, the first one at the position pointed to by str, another one
at str+pos[px]. */

char *str;
size_t pos[100];
unsigned int px;

/* ... */

remove_filename(str);
remove_filename(str + pos[px]);
```

After the invocation **remove_filename(str)** is replaced by its expansion, the **_Outer(x)** tokens from the macro definition become **_Outer(str)**, which according to the rules for the evaluation of an identifier-resolver selection evaluates to **(_Outer str)**, which is itself a primary expression, for it matches the syntax **(** *expression* **)**. Within it, *expression* is **_Outer str**, which according to the rules for the resolution of a modified identifier evaluates to a semantic identifier which refers to the object **str** would refer to just before the beginning of the innermost scope, i. e., of the **{** which opens the **do{ }** block. This is just the only declared str, and **_Outer** would not have been necessary.

After the invocation **remove_filename(str + pos[px])** the replacement is **_Outer(str + pos[px])**, which evaluates to

_Outer str + _Outer pos[_Outer px]

and the **_Outer** that precedes **px** is necessary. There, **_Outer px** resolves to an identifier which designates the unsigned int from the declaration **unsigned int px**.

The **px** within the **typeof** operator does not need to be preceded by **_Outer** because after expansion:

typeof(*(px)) *px = _Outer(px)

by the time the first **px** is seen the scope of the **px** being declared has not yet begun (6.2.1 - 7, 6.7.6).

EXAMPLE 2

```
int i;  // 1st
int main(void){
        enum A{a, e, i = 6}; // 2nd
        {
        int i, j; //3rd

        j = _Outer  _Outer _Outer i;  // The innermost _Outer i is to be resolved at file scope: undecl. identifier
        i = _Outer j;          // Undeclared_identifier
        _Outer (int k[i]);     // OK. k is not affected by _Outer because it is a declarator. VLA having 6 elements.
```

```
        }
}
```

EXAMPLE 3

```
int i;
typedef unsigned int uint;
_Outer i = 2; // Error. _Outer at file scope

int h(int n);
int f(int _Outer i); // Constrain violation: i is a declarator
int g(void){
        int _Outer i; // Constrain violation: i is a declarator
        int uint, f;
        int k = _Outer uint; // Constrain violation, equivalent to int k = unsigned int;

        f = _Outer f(f);          // OK
        f = _Outer (f)(f);        // The same as previous line
        f = h(f) + _Outer h(f);  // OK
        f = _Outer (f(i));        // Uses file-scope i
        {
                goto _Outer a; // Constrain violation. An ordinary identifier is not allowed here
        }
        a: ;
}
```

EXAMPLE 4

```
_Outer (int a = _Outer _Outer (x + _Outer y) + z)
```

Evaluates to

```
int a = _Outer _Outer _Outer (x + _Outer y) + _Outer z
```

which in turn evaluates to

```
int a = _Outer _Outer _Outer x + _Outer _Outer _Outer _Outer y + _Outer z
```

6.5.1.~~1~~2 Generic selection

Alternatives

To introduce a construction that would defer the beginning of the scope of an identifier being declared to the end of the instruction or to the next sequence point. For example:

```
char * _Deferred px = px; // The second px does not refer to the object the declared px will refer to
                          // because the latter's scope has not yet begun.
```

This has the disadvantage that the outer object cannot be referred to beyond the initialization of the inner one. Example 1 featured a case of that use.

Comments

• Two `_Outer` may be needed if the macro opens another block. For example (suppose uint is unsigned int):

```
#define mmul_vectorn(A,B,v,a,c) if(a){ \
        memset(A, 0, a*sizeof(double)) \
        const double *pB = _Outer B, *pC = _Outer v; \
        double *pA = _Outer A; \
        uint ic = _Outer(c); \
        while(ic){ ic--; \
```

```
            double aux = *pC++; \
            uint ia = _Outer _Outer (a); do ia--, *pA++ += *pB++ * aux; while(ia >= 1); \
            pA -= _Outer _Outer (a); \
    }}
```

This seems to be not a problem. I think it is preferable to keep the effect of `_Outer` clear than to attempt a smarter definition that would likely bring in much more problems than it would solve.

• The previous example also serves to illustrate the need of the `_Outer(x)` syntax versus `_Outer x` in same cases. The arguments passed to the macro in place of `a` and `c` could be constants. Suppose for instance `c` to be `6`. The two syntax would yield, respectively,

```
    uint ic = _Outer(6);
    uint ic = _Outer 6;
```

The second one is not valid, since `6` is not an identifier. The existence of the `_Outer(x)` variant relieves the definition of *semantic-identifier* from the need to deal with those cases. It would be a little messy since the same sequence of tokens can match both `_Outer` *identifier* and `_Outer` *constant*. This happens when the identifier is an enumeration constant. Also, `_Outer 6` looks weird. There being no need for it, it seemed better not to permit `_Outer` to precede constants like this. It is the most superfluous since macro parameters are most often enclosed in parentheses in its replacement text.

• The syntax has been defined with the care that

```
_Outer1 ( _Outer2 (expr ) + _Outer3 x)
```

expands to

```
_Outer1  _Outer2 (expr ) + _Outer1 _Outer3 x
```

While it wouldn't make any difference if the order of the `_Outer` identifier resolvers were reversed, it would in the (unlikely) event that other identifier resolvers were added to the syntax, and in any case the present expansion is what would be expected.

• In "Example 1", the declaration is `typeof(*(x)) *px` and not `typeof(x) px` because `x` could be an array.