Document number	P2142R1
Date:	2020-05-08
Audience	SG17 EWG Incubator
Reply-to	James (Jim) Buckeyne < d3ck0r at gmail > (AKA d3x0r https://github.com/d3x0r)

I. Table of Contents

II. Introduction

- Allow `.` operator to operate on pointers.
- Introduction

III. Motivation and Scope

- Modernizes method of member access to what basically every other language uses. Makes portability between different langugae codebases slightly easier. Simplifies the knowledge required to start programming in C, if there's just one operator, that works the same to access members of an object.

IV. Impact On the Standard

- All valid code previously written still behaves the same way.
- code generated with the new standard is potentially incompatible with previous compilers.

V. Design Decisions

- See remainder of this gist
- Introduction
- C++ Modification discussion

VI. Technical Specifications

- see remainder of this; there is a rough example implementation against a GCC 9.x compiler.

VII. Acknowledgements

- None known, no previous proposals have been mentioned.

VIII. References

- C++ 17 Standard draft
  - Relavent sections

IV. Changes

- R1
  - update contact information.
  - include link to original document.

C Standards Proposal

External Link

CXX Standards Proposal

This original, latest document, also version history External Link.

In section 8.5.1.5 2)

```
add: if the first expression of (dot) is a pointer [to an object], then E1.E2 is converted to (*(E1)).E2 .
```

more

Introduction

In C/C++, there are two operators `.` and `->` used to access members of `struct`, `class` and `union` types, as available. These operators are specified such that they are always paired with a single lvalue type; for example, if the left hand expression is a pointer to a `struct`, `class`, or `union`, then the operator `->` MUST be used. There is no occasion where `.` and `->` may be interchanged; although, C++ does allow overriding the `->` operator, and this existing behavior does not change.

Other languages (JS,Rust,Ruby,Python,Go,C#,Java,Kotlin,Swift,ObjC), indirectly derived from C have come to just use '.' as an operator; although, they also don't have a static struct instance that's a constant, and the behavior of '.' is really to dereference the left hand expression always. A leading cause of confusion in first learning C is why `.` and `->` are different, when they both just get the value from a structure.
Another side effect of having different operators is migrating code from a place where you have an instance of a structure you're passing by reference to read information into, and migrating to perhaps use a pointer to the buffer instead, and all the code has to be changed from '.' to '->' when it's really the same operation. It should be noted, though, that with the introduction of smart pointers, which provide a -> operator to appear like pointers, would still have to use the -> symbol, because the '.' operator would continue to work as normal on the base structure instance.

This is a simple example of current behavior.

```

struct s {
   int a;
};

void f( void ) {
    struct s S;
    struct s *P = &S;
    S.a
    P->a
}
```

The internal operation of '.' and '-> are nearly identical; `S.a` is just 'take the base address of the structure, add an offset to it'; or a shorthand (mem+offset) where mem is variable depending on where the function's execution frame is located. On the other hand, `P->a` is also (mem + offset), only the value comes from the lvalue, instead of just being the address of the lvalue.

It could be that the actual operation of these operators depends on the type of the left hand operator.

```
 struct t {
   struct s *pS;
};

struct v {
   struct t T;
};

struct v * pV;

void f2(void) {
    // there are no times where ->-> is immediately followed with but must always
    // be followed by an identifier, same with '.' except as a parameter '...'; which
    // is an entirely diffferent operator.
   pV.T.Ps //...
   pV->T->Ps //...
}
```

One immediate argument was 'clarity' that you know more information if the operators remain the same. That you would know that the above `T` was a pointer or a struct (whether that information itself is of any use is deatable).

What about the other method of dereferencing? `(* identifier)`

```
 void g( void ) {
    static struct s S;
    (*pV).T.pS = &S;
    pV->T.pS = &S;
}
```

In either case above the resulting type of the lvalue is either a `struct` or `union` or a `pointer to a struct` or a `pointer to a union`, and the token after the operator(s) is(are) an() identifier(s), which is a member of that struct or union.

The meaning of `->` and `.` ...

This is a more complex example. It involves a base class 'b' that has a data state, which is accessed through many levels of indirection in a single expression. (This is not a style or design practice I would recommend, having the partials saved as the result of the check to see if it's valid...)

```
 struct b { // base
    int a; // a value
};

struct f { // functional 1
    struct b b;
}

struct g { // functional 2
    struct b b;
}

struct h { // merge
    struct f *f;
    struct g g;
};

struct s {
    struct h h;
    struct h *ph;
}



void f( void ) {
    struct s S;
    sturct s *pS;

    S.ph = &h;
    S.ph->f = NULL;
    S.ph->g.a = 3;

    s.ph->f.b.a = 3; // with new syntax seg fault, but stylistitically -> was used only once
    s.ph->f->b.a = 3; // you would know you only had to check two pointers.
    s.ph.f.b.a = 3; // total adoption, JS style, every '.' access is considered illegal,
                  //   and there's 3 places you need to check if it's a valid pointer; when in reality it's only 2.
}
```

Continued

(more on above style)

```
     struct h *ph_;
    if( ph_ = s.ph ) {
        struct f *pf;
        if( pf = ph_.f ){
            // and really this would be  b_set_a( &pf.b, 3 );
            pf.b .a = 3;
        }
    }
```

That does remind me of the other extension, of declared variables with the scope of an expression and its following statement block... like `for( int .. )` or `while( int ... );` why can't I just arbitraily `( int a, ( a = otherfactors * otherscalars ) ) {` ... well I guess if(...) but that's a different proposal.

C++ Compiler modification notes

The implementation of the correction, is a matter of finding where the error is generated when using '.' to indirect a pointer, and work backwards; an exception handler can be added for 'if left hand value is a pointer to a class/struct use that instead'. It's a matter of decision of how the stanard is worded whether this should also be applied recursively for 'pointer to pointer to...'.

It's a very minor differentiation In the GCC Compiler between the handling of `CPP_DOT` and `CPP_DEREF`, and the routine `build_indirect_ref` is only referenced in that one place (definition of `build_indirect_ref`)... (definition of convert_lvalue_to_rvalue)) which inspects the left hand value to see if it is a pointer, else it returns an error; with a flag check it could instead just return the expression itself, and then that code and the CPP_DOT handling code would be identitical... resulting with just a slightly different resulting expression tree instead of emitting errors.

The other difference, at the start of the handling, `convert_lvalue_to_rvalue (expr_loc, expr, true, false);` and `default_function_array_conversion (expr_loc, expr);` the `convert_lvalue...` routine calls `default_function_array_conversion` as part of its evaluation, and then additionally checks something about an atomic lvalue.

C++ Modifications

https://lists.isocpp.org/std-proposals/2020/02/1021.php is a conversation I had badly phrased a sumamry of this gist to start... Please see there for more criticisms.

Regarding C++, it is important to distinguish between raw pointers and other pointer types `*_ptr`; In the case of the following code, the use of the '.' operator results in an error.

```
 struct { int a; } s, *ps;

/* in code... */
ps.a  // illegal expression, generates a compiler error.
```

This does overlap the funcationality of reference types...

```
 struct s{ int a; } s, *ps;

/* in code... */
void f( struct s &s ) {
    s.a  // legal.
}


void g( struct s *s ) {
    s.a  // proposed legal.
}

void main( void ) {
    struct s s;
    f( &s );
    g( &s ) ;
}
```

The C++ parser.c file is 1.2MB; and Github refuses to show such a large file; cannot directly link to a line of code for reference...

```
 gcc/cp/parser.c : line 7618 (Release Version 9.2.0)



  /* If this is a `->' operator, dereference the pointer.  */
---  if (token_type == CPP_DEREF)
+++  if (token_type == CPP_DEREF || ((token_type == CPP_DOT)&&(cxx_dialect >= cxx2a)&&( (TYPE_PTR_P (TREE_TYPE (postfix_expression))
    postfix_expression = build_x_arrow (location, postfix_expression,
                    tf_warning_or_error);
```

This would be a simpler modification than even the C change.

Offsetof (in an error state for C) but this wouldn't work for C++ anyway? Since I have to #include <stddef.h> for both C and C++, why don't they behave the same? I can manaully define 'myoffsetof'... (later)

```
#include <stdio.h>
#include <stddef.h>

struct inner{
        int c, d;
};

struct s {
        int a, b;
        struct inner Inner;
        struct inner *pInner;
};

void f( void ){
        struct s S;
        struct s *P = &S;
        S.pInner = &S.Inner;
        P.a = 5;
        P.b = 13;
// This is an error; only '.' behaves like '->' not vice versa.
//      printf( "Output: %d %d\n", S->a, S->b );
        printf( "Output: %d %d\n", S.a, S.b );
        printf( "Output: %d %d\n", offsetof( struct s, Inner.c ), offsetof( struct s, pInner.c ) );
}

int main( void ) {
        f();
        return 0;
}
```

```
d3x0r@battlement c2x-test]$ ./tst
In file included from a.c:2:
a.c: In function 'f':
a.c:22:60: error: '0->pInner' is a pointer; did you mean to use '->'?
   22 |  printf( "Output: %d %d\n", offsetof( struct s, Inner.c ), offsetof( struct s, pInner.c ) );
      |                                                             ^~~~~~~~
Output: 5 13
Output: 5 13
Output: 0 4
In file included from b.cc:2:
b.cc: In function 'void f()':
b.cc:23:60: error: cannot apply 'offsetof' to a non constant address
   23 |  printf( "Output: %d %d\n", offsetof( struct s, Inner.c ), offsetof( struct s, pInner.c ) );
      |                                                             ^~~~~~~~
Output: 5 13
Output: 5 13
Output: 0 4
```

C++ Standards Proposal (more)

In section 8.5.1.5 2)

```
add: if the first expression of (dot) is a pointer [to an object], then E1.E2 is converted to (*(E1)).E2 .
```

Although there potential recursion there, without the 'to an object'...

```
class X ****x;   `x.property` could be `(*(*(*(*(x)))).property`
and unique_ptr<X> ***upx;    `upx.reset();` could be (*(*(*(upx)))).reset() and still work...
```

Maybe reword the above specifying that 'pointer to pointer to...' is still an invalid type to '.' and only one level of pointer indirection may apply. (' The unary * operator performs indirection:')

Although, having reflected on the above shorthand ability, I think maybe it could stay. There aren't a lot of use of `****` that aren actually arrays, which would be indexed with a non-zero index potentially; that is the '.' operator only dereferences the 0'th element.

Sections from C++ 17 draft which may be relavent

6.4.5 defines 'class member access' about how the id on the right of -> and . is looked up...

8.5.1.5 2) For the first option (dot) the first expression shall be a glvalue having class type. For the second option (arrow) the first expression shall be a prvalue having pointer to class type
`E1->E2 is converted to the equivalent form (*(E1)).E2;` `the remainder of 8.5.1.5 will address only the first option (dot).` ...

23.11 Smart Pointers .1 1) `A unique pointer is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object u that stores a pointer to a second object p`...

So, a Smart pointer is an object, not a pointer to an object; but clearly 'is an object that owns [a pointer to] another object]. where did 'object' get defined to relate to 'class type' or 'pointer to class type'....

6.6.1 Memory Model 3) "A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width "

6.7.2 Compound Types 3) ...' A pointer to objects of type T is referred to as a “pointer to T '...

6.4.5 Class member access [basic.lookup.classref] "1 In a class member access expression (8.5.1.5), if the . or -> token is immediately followed by an identifier followed by a <, the identifier must be looked up to determine whether the < is the beginning of a template argument list (17.2) or a less-than operator. The identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire postfix-expression and shall name a class template. "