Accredited Standards Committee X3
Information Processing Systems
Operating under the procedures of
American National Standards Institute

Doc No: X3J16/96-0124 WG21/N0942
Date: May 28th, 1996
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Memory Model Issues and Proposed Resolutions

641 - Which allocation/deallocation functions are predefined and which ones may be overridden in a program?

======

3.7.3 [basic.stc.dynamic] para 1 & 2 says:

"A C++ implementation provides access to, and management of, dynamic storage via the global allocation functions operator new and operator new[] and the global deallocation functions operator delete and operator delete[].

The global allocation and deallocation functions are always implicitly declared. The library provides default definitions for them (_lib.new.delete_). A C++ program shall provide at most one definition of any of the functions ::operator new(size_t), ::operator new(size_t, void*), ::operator new(size_t, const std::nothrow&), ::operator new[](size_t), ::operator new[](size_t, void*), ::operator new[](size_t, const std::nothrow&), ::operator delete(void*), ::operator delete(void*, void*), ::operator delete(void*, const std::nothrow&), ::operator delete[](void*), operator delete[](void*, void*), and/or ::operator delete[](void*, const nothrow&). Any such function definitions replace the default versions. This replacement is global and takes effect upon program startup (_basic.start_). Allocation and/or deallocation functions can also be declared and defined for any class (_class.free_)."

1) Para 2 seems to indicate that all allocation/deallocation functions can be overridden by the user in a program. This contradicts what 18.4 [lib.support.dynamic] says.

2) The text above does not make it very clear which allocation/deallocation functions are predefined by the implementation, that is, which allocation functions can be called as the result of a new expression without the need to include <new>.

The above paragraphs seem to imply that the answers to questions 1) and 2) are the same. I don't believe this is right.

Proposed Resolution:

---

1) 18.4 [lib.support.dynamic] indicates that the following allocation/deallocation functions are replaceable:

```
::operator new(size_t)
::operator new(size_t, const std::nothrow&)
::operator new[](size_t)
::operator new[](size_t, const std::nothrow&)
::operator delete(void*)
::operator delete(void*, const std::nothrow&)
::operator delete[](void*)
::operator delete[](void*, const nothrow&)
```

and that the following are not replaceable:

```
::operator new(size_t, void*)
::operator new[](size_t, void*)
```

```
    ::operator delete(void*, void*)
    ::operator delete[](void*, void*)
```

The text in 3.7.3 should match what 18.4 [lib.support.dynamic] says.

2) I believe any allocation or deallocation function can be called as the result of a new expression without the need to include <new> (even if the allocation and deallocation function is not replaceable). In particular, the following is well-formed, even if <new> is not included in the source file to declare placement ::operator new(size_t, void*).

```
struct C {
    int i;
    void f();
    const C& operator=( const C& );
};

const C& C::operator=( const C& other)
{
    if ( this != &other )
    {
            this->~C();
            new (this) C(other); // well-formed
            f();
    }
    return *this;
}
```

The text in 3.7.3 should make this clear.

. . . . . .

577 - Are there any requirement on the alignment of the pointer used with new with placement?

======

For example, 12.4 para 10 gives examples of placement new used with a buffer created as follows:

```
    class X { };
    static char buf[sizeof(X)];
```

Is the alignment of a static array of char guaranteed to satisfy the alignment requirements of an arbitrary class X?

Proposed Resolution:

Either 3.7.3.1[basic.stc.dynamic.allocation] or 18.4.1.3[lib.new.delete.placement] should indicate that the second argument of

```
    ::operator new(size_t, void*)
    ::operator new[](size_t, void*)
```

should be a pointer to a storage location that is suitably aligned to hold an object of the type being newed.

The example in 12.4 para 10 should be rewritten as follows

```
    class X { };

    static union {
        static char buf[sizeof(X)];
        X dummy;
    };
```

so that the constraint above is respected.

. . . . . .

453 - Can operator new be called to allocate storage for temporaries, RTTI or exception handling data structures?

======

Is it permitted for an implementation to create temporaries by calling operator new()? If so, does that require that operator new() be accessible in the context in which such a temporary is created?

Is an implementation allowed to call a replaced operator new() whenever it likes (storage for RTTI, exception handling, initializing objects with static storage duration in a library)?

Proposed Resolution:

The Core 1 WG discussed this issue in Monterey and this is the resolution it seemed to converge towards:

> The storage for variables with static storage duration, for data structures used for RTTI and exception handling shall not be acquired using operator new.

The following words should be added to 3.7.3 to make this clear:

> "A global allocation function is only called as the result of a new expression (5.3.4[expr.new]) or indirectly through calls to the functions in the C++ standard library. [Note: in particular, a global allocation function is never called to allocate storage for variables with static storage duration, for the data structures used for RTTI or exception handling.]"

. . . . . .

NEW - Can a pointer to an incomplete class type be the operand of a delete expression?

======

```
class X;
void f(X* px)
{
    ...
    delete px;
    // undefined if X has a destructor or operator delete.
}
```

At the Santa Cruz meeting, the WG decided that the ability to delete a pointer to an incomplete POD class type was important enough to keep the undefined behavior in this case. C allows a pointer to an incomplete class type to be the operand of free and people participating in the discussions felt that users should be allowed to use delete where free was used in C. The WG believed that it should be a quality-of-implementation issue to generate a warning if a pointer to a incomplete POD class type was the operand of a delete expression.

Mike Miller core-6607:

> > I don't understand the C compatibility argument in this context.
> > It's clearly not source compatibility, since C doesn't support the
> > "delete" syntax. The only compatibility I can see is the ability
> > to deallocate an incompletely-typed pointer in the same contexts
> > in which "free()" could be called in C; however, that capability
> > would not be lost if the rule were changed to require a complete
> > type, since you could just cast to char* if you really needed to

> delete an incomplete type.

Nathan Myers core-6774:
> [Allowing the operand of delete to be a pointer to an incomplete
> class type for reasons of C compatibility] In my view, that's
> pretty thin reasoning -- not least because it encourages people to
> say "delete p" when they really need to check whether "delete[] p"
> is required, and a compiler can't detect that to warn about it.
>
> To delete, deliberately, an object *p of incomplete type, I would
> expect to be obliged to replace "free(p)" with
> "delete static_cast<void*>(p)", which makes visible that something
> unsavory is going on. Then "delete p" could be diagnosed as the
> error it quite likely is.

Erwin Unruh in core-6603
> This problem is related with a more general problem:
>
>   When is the use of a type a reason to instantiate a template?
>
> The mental model answering this question was: "whenever the
> completeness of a class template specialization changes the
> semantics of the program".
>
> A delete expression is one of these situations.
> We then have situations where a class may be incomplete, but a
> class template specialization may not be.
>
> Requiring the class to be complete when a pointer to this class
> type is used as the operand of a delete expression would remove
> one problematic situation for defining when template instantiation
> must take place.

Proposed Resolution:

I don't care greatly but I somewhat favor leaving things as is.
I still believe it is a quality-of-implementation issue to generate
a warning if a pointer to a incomplete POD class type is the operand
of a delete expression.

. . . . . .

645 - Should &*(array+upperbound) be allowed?

5.3.1 [expr.unary.op] para 1 says:
"The unary * operator performs indirection: the expression to which
it is applied shall be a pointer to an object type or a pointer to
function type and the result is an lvalue referring to the object
or function to which the expression points."

```
int a[4];
... *(a+4) ... // well-formed?
```

The problem is that a+4 does not point to an object.
Is it ill-formed to apply the * operator to such an expression?
If it is then the common idiom &a[n] where n is one past the end of
the array is ill-formed since one must be able to rewrite a[n]
as &*(a+n).

Possible Solutions:
---

1) Leave things the way they are now.

Mike Miller core-6570:

> It hasn't seemed to hurt the C standard all that much, and there
> is a simple idiom that is conformant (a+upper).

against 1):
- - - - - -
Bjarne in core-6590:
> It was a surprise to me to find that the
>
>   &v[size]
>
> that I have relied on for the last 20 years or so and widely
> recommended isn't allowed by the C standard. I suspect that
> I'm not the only one caught with a surprise and a lot of
> nonconforming code. I think finding a way of allowing this
> (without rewriting major tracts of standardese) would be an
> excellent idea.

2) allow &a[upper] and nothing else.

Tom Wilcox core-6586
> Perhaps we could avoid the whole issue of l-values and
> interaction with * by recognizing this as an idiom and DEFINING
> (in the definition of unary &) the special case
>
> & (pointer-expression) [index-expression]
>
> as being equivalent to
>
> pointer-expression + index-expression

against 2):
- - - - - -
Mike Miller core-6591:
> The problem I have with this approach is that it breaks the
> fundamental identity between E1[E2] and *((E1)+(E2)).
> Currently, the two are _always_ equivalent. Period. Nothing
> further to say. That's why we can get away with less than 5
> lines total (1.5 of which are non-normative notes) to describe
> subscripting. All the semantics are described under * and +.
>
> If we take this approach, we would have to change 5.2.1 to
> say, "The expression E1[E2] is identical (by definition) to
> *((E1)+(E2)) UNLESS E1 HAS A POINTER TYPE, E2 HAS AN ARITHMETIC
> TYPE, AND THE EXPRESSION IS THE OPERAND OF THE (BUILT-IN) UNARY
> & OPERATOR, IN WHICH CASE SPECIAL SEMANTICS APPLY." This
> strikes me, at least, as unesthetic.
>
> That's why approach 3) dealt with the descriptions of * and &.
> I really don't think we ought to get into the business of
> creating and describing a set of circumstances in which E1[E2]
> and *((E1)+(E2)) are different from each other. (I'm assuming
> from Tom's comment that &array[max] would be legal but
> &*(array+max) would not.)

3) allow &a[upper] and &*(a+upper), but nothing else.

Mike Miller core-6570:
> Perhaps something like the following could be done in 5.3.1p1:
>
> "If the value of the pointer is the address of an object or
> function, the result of the expression is an lvalue
> designating that object or function; otherwise, any use of the
> result of the expression other than as the operand of the
> (built-in) unary & operator or the sizeof operator produces
> undefined behavior."
>

> 5.3.1p2 would also have to be changed to describe what happens
> when the operand of & is a non-lvalue, perhaps something like:
>
> "If the operand is the result of applying (built-in) unary * to
> a pointer whose value is not the address of an object or
> function, the result of unary & is the same value as the operand
> of the unary *. Otherwise, the operand shall be an lvalue...
> [remainder of original description]."

4) allow a[upper] to participate in the language as fully as possible.

Andrew Koenig core-6592:
> I think the way to correct the definition of pointer arithmetic
> is to permit an off-the-end pointer to be dereferenced but to
> prohibit any operation on the resulting reference except taking
> its address or binding another reference to it. Thus, I see
> nothing wrong with the following example:
>
>   int a[100];
>
>   int& f() { return a[100]; }
>
> and I would even allow
>
>   int& x = f();
>   int& y = x;
>
> but not
>
>   int z = x;

John Skaller core-6592:
> More may be necessary than just that.
>   * reinterpret cast SHOULD be allowed, since it doesn't change
>   the address referred to.
>   * typeid SHOULD be allowed if the type is not polymorphic
>   * constant member selection of the FIRST member of a POD
>   struct or union should be allowed (because this is just a
>   reinterpret_cast).
>   Note this includes both constants as in "x.member" and
>   variable as in "x.*ptm" provided the ptm denotes the first
>   member.

Bill Gibbons core-6614:
> I suggest that we base the rule on the existing limitations on
> the use of objects with incomplete types:
>
>   If a pointer refers to the memory location just after the end
>   of an array, any operation on that pointer which, if applied to
>   an ordinary pointer of that type, would have different
>   semantics (including become ill-formed) if the underlying type
>   were incomplete, has undefined behavior, except:
>
>   * Adding a zero or negative value.
>
>   * Subtracting a zero or positive value.
>
>   * Computing the difference between that pointer and another
>   pointer.
>
> This handles a large range of problem cases, such as:
>
> - x[n].member   // undefined
>
> - (BaseClass*) &x[n]   // undefined
>

```
>  - &x[n] - 1 + 1           // OK
>
>  - &x[n] + 1 - 1           // undefined
>
>  - typeid(x[n])            // OK only if refers to a
>                            // non-polymorphic type
>
>  - struct A { }
>    struct B { B* operator&(); };
>    A a[5];
>    &a[5];                  // OK
>    B b[5];
>
> A similar restriction might clarify the use of null pointers,
> e.g.
>
>  int *pi = 0;
>  typeid(*pi);     // OK, since not used where a complete type is
>                   // needed
>  int *pj = &*pi;  // also OK

 Mike Miller core-6617:
> If we do take this approach, however, I think it's mandatory to
> define a[upper] as an lvalue; otherwise, some tinkering will be
> required in many places to define how those permitted operations
> work on a non-lvalue operand.
>
>   "An lvalue refers to a function, to the storage associated
>    with an object (in or outside its lifetime), or to the address
>    just past the last element of an array (5.7)"

   cons against 4):
   - - - - - - - -
   Such a change affects a very basic concept of the language, i.e.
   lvalues.  Changing the meaning of lvalues has rippling effects
   through the WP and has the potential of introducing some
   inconsistencies in the WP.

Proposed Resolution:
--------------------
I prefer option 3).
It is more prudent.

. . . . . .

513 - Clarifications for the rules on pointer conversions
======

1- What is the status quo?
--------------------------

  In particular, 5.9[expr.rel]p2 last '--' says:
    "Other pointer comparisons are unspecified."

  Andrew Koenig notes the following:
    > Saying it is unspecified is a tremendous difference from C.
    > The point is that in C on, say, the Intel 386 in 16-bit mode,
    > when doing an ordering comparison it is sufficient for the
    > compiler to generate code to compare only the low-order 16 bits
    > of the pointers because the comparison is defined only for two
    > elements of the same array.  If C++ is required to compare the
    > whole address, that puts it at a significant performance
    > disadvantage with respect to C.

Proposed Resolution:
--------------------
```

No action.
The WP already reflects the following status quo
(Summary based on Mike Miller's message core-6626):

a) pointers to the same object or function, or both pointers point one past the end of the same array, or both pointers are null: pointers compare equal

b) pointers to different objects or functions, or only one pointer is null: pointers compare unequal; the exact result is unspecified

c) pointers to class members of the same class object if members are not separated by an access-specifier: pointer to the later declared member compares higher

d) pointers to class members of the same class object if members are separated by an access-specifier: the result of the pointer comparison is unspecified

e) pointers to members of the same union object compare equal

f) pointers to elements of the same array or one beyond the end of the array: pointer to the higher subscript compares higher

g) all other pointer comparisons are unspecified

[Note]:
Unspecified means that the pointer comparison is well-formed. The result of the comparison can otherwise be whatever the implementation wishes.

In particular, Andy's comment is already taken care of. An implementation is allowed to only compare the low-order 16 bits if it wishes since the result of pointer comparison [i.e. g)] is unspecified.

. . . . . .

2- Should the standard indicate that:

- the result of a pointer comparison must be either true or false?
- the result of a pointer comparison must be consistent throughout an entire program ?

2.1- Should the standard indicate that the result of a pointer comparison must be either true or false?

I think this is already the case, though it wouldn't hurt to make this clearer.

Unspecified means that the pointer comparisons are well-formed.

Mike Miller in core-6636:
> The "unspecified" result of "other pointer comparisons" is not
> completely unconstrained. I neglected to take into account the
> sentence in 5.9p1 that says, "The type of the result is bool." I
> think it is defensible, if not, perhaps, definitive, to argue
> that this sentence implies that a _valid_ bool value is to be
> returned by all the operations described in the section, even if
> otherwise unspecified.

Proposed Resolution:

Editorial work to make the words in para 1 stronger:
Even though some pointer comparison are unspecified, all pointer

comparison shall yield true or false.

2.2- Should the standard indicate that the result of a pointer comparison must be consistent throughout an entire program, even though the result of such a comparison is unspecified?

Should the standard impose the following additional restrictions?

-- given that p and q are pointers of the same type and that, throughout the duration of the program, p always points to the object x and q always points to the object y, the comparison
p < q
must yield the same result throughout the duration of the program.

-- given that p and q are pointers to data members of the same object of class type T (separated by an access-specifier), the comparison
p < q
must be consistent within the execution of a single program.

-- given that p and q are pointers to data members of an object of class T (separated by an access-specifier), if p and q are modified to point to the same data members of a different object of class T, the comparison
p < q
must yield the same result as it would have yielded with the original values of p and q within the execution of a single program. That is, the comparisons of pointers to data members must be consistent for all objects of the same class type within the execution of a single program.

This seems to be the behavior most people expect from their various implementations. It is not clear that requiring implementations to support these additional constraints actually solves important problems faced by C++ programmers. So we may decide that imposing such constraints is not really necessary.

Requiring implementations to support the "consistency" constraints above imposes some restrictions on implementations on how pointers are manipulated:

Mike Miller in core-6636:
> (This is not completely theoretical -- Tom’s comments in
> core-6635 about unnormalized pointers reminded me of some
> machine architectures I’ve worked on where the format of
> pointers in memory and data registers was different from the
> format in address registers. Depending on which values were in
> which registers for a given comparison, if an implementation is
> free to pick the least expensive instruction sequence that gives
> the right answer for related objects, it might indeed be
> possible to get different results for separate comparisons of
> the same two pointers to unrelated objects in different parts of
> the same program execution. If we want to disallow this, the
> restriction needs to be explicit.)

Erwin Unruh in core-6653:
> I think mandating the same result when comparing two pointers
> disallows some optimisations made possible through data flow
> analysis. See the following program:
>
> int a;
> int ar[5];
> int &b=ar[0];
>
> void f(int* pa, int* pb)

```
    >   {
    >           pa < pb;        // #1
    >           pa < &b;        // #2
    >   }
    >
    >   f(&a, &b);
    >
    > When analysing #1 we don't know whether these pointers point to
    > element of the same array and what relation they have.  The
    > implementation has to generate a comparison.  The result is
    > unspecified and, at runtime yields true in this case.  For #2 the
    > implementation knows that b points to the first element of an
    > array.  So the result is either false or unspecified.  To
    > optimize, the implementation may assume that the result is false
    > and optimise the comparison away.  So comparing the same pointers
    > with correct type yields two different results.
    >
    > Other optimisations may occur more often where the implementation
    > replaces a "<" by a "==", as in
    >
    >   for( ..., p<(ar+5), ... )
    >
    > Here the only valid situation for "false" would be "p==(ar+5)",
    > so an optimizing compiler may replace the comparison (especially
    > when equality is cheaper than less).
    >
    > So mandating consistency will disallow a certain set of
    > optimisations.

  Proposed Resolution:
  --------------------
    I don't have a strong opinion regarding the outcome of this issue.

    We will have to decide whether we care enough about the
    consistency of pointer comparisons and about ensuring that the
    users expectations be supported by all implementations to
    disallow implementation tricks and optimizations such as the ones
    presented by Mike and Erwin.

. . . . . .

3- Should every implementation be required to provide a total
   ordering on pointers?
------------------------

  Solution 1): No. C doesn't so why should we?
  ------------

  Solution 2): Yes.
  ------------

  o Why?

    Andrew Koenig in core-6639:
    > The principal argument is that having a canonical total ordering
    > on pointers makes it much easier to use pointers as indices of
    > associative containers.  In other words, it makes set<T*> and
    > list<T*,V> possible to implement.

 o How?

   a) keep the semantics of < > <= >= unchanged.
      In which case, this becomes a library issue.

      Matt Austern in core-6642:
      > The declaration of set is
```

> template<class Key, class Compare = less<Key>,
> class Allocator = allocator>
> class set { /* ... */ };.
>
> It doesn't matter, then, whether or not you use operator< to
> compare two pointers; all that matters is that you have a
> less<T*>. The trouble is simply that there is no portable
> way to write less<T*>.
>
> Perhaps this is best regarded as a library issue rather than
> a core language issue: perhaps template<class T> class
> less<T*> should simply be added to the library, and standard
> library implementors could use whatever platform-specific
> magic is necessary.

Jerry Schwarz in core-6655:
> I've seen several proposals in this thread that the way to
> answer this question would be a template in the library
> bool less<T*>(T*,T*);
> I think this goes overboard on templates because of the
> potential for all kinds of problems with determining the type
> to instantiate the template. I think
> bool less(void*,void*);
> would be the simplest way to proceed.

Tom Plum in core-6670:
> IMO, both [Matt] and Jerry are right, but at different levels
> of the problem. [...] for the portable library
> implementation to implement the template for less<T*> it must
> ultimately call some function (in general). At least if the
> library standardizes upon a function interface, the
> non-portability is confined to the implementation of that
> (possibly builtin) function.

b) operator< (and company) reflects the total ordering on void*
pointers (only). The result of operator< (and company) on
other pointer types remains unspecified.

Tom Plum in core-6635:
> There was an off-line discussion of this earlier this year,
> and IMO, I think it's a great idea. Once a pointer has been
> cast to void* it's been made clear that the pointer is no
> longer relative to any specific underlying object. And you
> can't fetch or store with it unless you have some other
> information that's kept outside the pointer.
>
> But remember, an unnormalized pointer compare can be
> significantly faster than a normalized compare, so don't force
> char* comparisons to be normalized. They're inside many of
> the hottest inner loops. But void* compares are much less
> frequent.

c) operator< (and company) reflects the total ordering on
all pointer types.

Erwin Unruh in core-6695:

> There have been arguments that requiring a total order on a
> pointer does require suboptimal code on some architectures.
> Now Francis entered a new view of architecture, which is
> currently not implemented but which will be available within
> the lifetime of C++.
>
> > Francis W Glassborow in core-6692:
> >
> > When we move to more complicated distributed systems (e.g.

> > URL's on the Internet) it is possible that there is no
> > normalisation algorithm and that two 'lexically' distinct
> > pointers actually point to the same 'object'
> So I believe that adding a new requirement on the operator<
> for pointers is not the right way. A programmer assumes that
> basic functionality on basic types is somewhat 'fast'. But
> maintaining a complete order on a distributed network (with
> nodes entering and leaving) may require the presence of a
> complete database.
> [...]
> I could live with a library function which provides a
> complete ordering on all pointers. If an implementation of
> such a function becomes too expensive I can just remove that
> function and rewrite the library. Than only users working
> with that library function will have problems (at link time).
> [...]
> So I strongly argue against extending the semantics of
> operator<. I could live with a library function for a
> complete ordering of pointers. Such a library function should
> have a note that a constant time comparison is not guaranteed.

Proposed Resolution:

Given the schedule constraints, I favor 1).
I could live with 2a).

. . . . . .