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

Doc No: X3J16/95-0055 WG21/N0655
Date: March 3, 1995 Page 1 of 12
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Object Lifetimes: Memory Tricks and Objects with Disrupted Lifetimes

1. Pointer manipulations

1.1. Conversion from T* to void*

It is not guaranteed that a T* will point to the beginning of the storage allocated for an object of type T.

```
class B {
   virtual void f();
   virtual ~B();
};
class D1 : public B { void f(); };
D1* pd = new D1;  // pd may not point to the beginning of the storage
                  // allocated for an object of type D1
```

Proposal

When converting a T* to a void*, the pointer might change value and the void* that results is guaranteed to point at the start of the storage holding the object of type T.

Example

```
class B {
  virtual void f();
  virtual ~B();
};
class D1 : public B { void f(); };
void* p = malloc(sizeof(D1+D1)); // gets enough space.
D1* pd = new (p) D1;
```

p will point at the start of the storage allocated to hold the object of type D1. It is not guaranteed that pd will point at the start of the storage location containing the object D1, that is:

```
p == pd           // may yield false
p == (void *) pd  // always true
```

1.2. placement operator new

The WP description of operator new with placement says:

18.4.1.5.1 [_lib.placement.op.new_]
  void* operator new(size_t size, void* ptr);
Returns ptr."

This implies (I believe):

```
T t;
T* tp = new(&t)T;
if (tp==&t)  // yields true since the same type is newed
```

The 2nd operand to placement new '&t' is converted from T* to void*. The new expression creates an object of type T at the memory location ((void*)&t), which, for any type T, is the start of the memory location where the object t resided. The new expression returns a pointer to T, i.e. ((T*)((void*)&t)). tp == &t.

1.3 What can be done with a pointer to an object that has been destroyed?

```
T* pt = new T;
pt->~T();
```

Proposal:

The following text should be added to sub-clause 12.4 [class.dtor]:

8 A pointer to an object of type T that has been destroyed (p->~T()) can only be used in limited ways. Using the pointer as an T* is no longer valid. However, the pointer still points at valid memory and using the pointer as a pointer to the memory where the object was located '(void *)p' is well-defined. In particular, such a pointer cannot be used to refer to any non-static (data or (virtual or non-virtual) function) members of the destroyed object (doing so results in undefined behavior). However, such a pointer can be used to access other objects. For example, the pointer can be used to access static data members or call static member functions of the class type T.

9.5 [class.static]

2 It [the static member] can also be referred to using . and -> member access operators even after the object referred to by the object expression has been destroyed.

Example:

```
class B {
    virtual void f();
    virtual ~B();
    static g();
};
class D : public B { void f(); };
```

```
    void* p = malloc(2*sizeof(D)); // gets enough space.
    B* pb = new (p) D;
    pb->~B();          // calls virtual destructor.
    void *q = pb;      // Ok, pb is a pointer to valid memory
    pb->f();           // undefined: f is a non-static member function
    pb->g();           // undefined: equivalent to *pb (undefined), B::g()
```

Another example:

```
    class C {
        void f();
        void destroy();
        static g();
    };
    void C::destroy ()
    {
        this->~C();
        f();           // undefined: f is a non-static member function
        g();           // well-formed: equivalent to C::g()
    }
```

1.4 Base class subobjects

1.4.1. Conversion from pointer to derived to pointer to base

```
    class B {
       virtual void f();
       virtual ~B();
    } ;
    class D1 : public B { void f(); };
    void* p = malloc(2*sizeof(D1)); // gets enough space.
    D1* pd = new (p) D1;
    B* pb1 = pd;
```

The WP, subclause 4.10 [conv.ptr] already indicates that

```
  pb1 == pd // may yield false
```

For the same reason:

```
  pb1 == p  // may yield false
```

That is, the language does not guarantee that any pointer to an object's base class will point to the start of the storage location where the object resides.

1.4.2. complete object of type T vs base class subobject of type T

Proposal:

An object of type T that is a base subobject is not guaranteed to have the same size and the same layout as a complete object of type T. An object of type T that is a base subobject of another class is not guaranteed to have the same polymorphic behavior as a complete object of type T.

```
class A { };
class B : public virtual A { };
class C : public virtual A { };
class D : public B, public C { };

C c;
D d;
C* pc = &d;
```

The size and layout for the d's base subobject C might be different from the size and layout of c.

Proposal:

```
void* p = (void *)pc
```

When converting a T* to a void*, the pointer might change value and the value of the void* pointer points at the start of the storage holding a complete object of type T.

That is, if T* happens to point at a base class subobject, then it is not guaranteed that the conversion T* --> void* will cause p to point at the beginning of the storage location where the subobject of type T resides. The address calculation from T* to void* is performed as if the T* pointer pointed at a complete object of type T and the resulting void* pointer is set to point at what would be the beginning of the storage location for the complete object of type T.

1.5 Summary

```
class B {
   virtual void f();
   virtual ~B();
} ;
class D1 : public B { void f(); };
class D2 : public B { void f(); };

void* p = malloc(sizeof(D1)+sizeof(D2)); // gets enough space.
D1* pd = new (p) D1;
B* pb1 = pd;
pb1->~B();             // calls virtual destructor.
B* pb2 = new (p) D2;   // reuses space that has been destroyed.
pb2->f();              // calls D2::f().
```

As mentioned in 1.1,

```
p == pd
```

might yield false. However,

```
p == (void*) pd
```

is always true. 'p' points at the start of the storage holding the object of type D1. The following expressions are therefore equivalent:

```
    new (p) D2;
    new ((void *)pd) D2;
    new (pd) D2; // since placement operator new takes a parameter of
                 // type void*, the argument is implicitly converted
                 // from the D1* to void*
```

As mentioned in 1.4,

```
  (void*)pb1
```

results in a pointer to what would be the beginning of the storage location for a complete object of type B. Therefore,

```
  (void*)pb1 == pd  // might yield false
  (void*)pb1 == p   // might yield false
```

And the expression:

```
  new (p) D2;
```

is therefore not equivalent to

```
  new ((void*)pb1) D2;
```

or

```
  new (pb1) D2;
```

2. The effects of placement new on existing objects

2.1 New object allocated of unrelated type to the existing object

Example:

```
  class B {
    virtual void f();
    virtual ~B();
  };
  class D1 : public B { void f(); };
  class D2 : public B { void f(); };

  void mutate(B** pb2, void *p) {
    (*pb2)->~B();
    new (p) D2;   //1: well-defined ??
  }

  void* p = malloc(sizeof(D1) + sizeof(D2));
  B* pb1 = new (p) D1;
  mutate( &pb1, p );
  pb1->f();              //2: well-defined ??
```

The code on line //1 has well-defined behavior if p points at a storage area that is large enough to hold an object of type D2. The code on line //2 has undefined behavior.

The function mutate has changed the type of the complete object at which pb1 points (from D1 to D2). The location of the base subobject B in D1 might be different from the location of the base subobject B in D2. The virtual function table of a base subobject B in D1 might be different from the virtual function table of a base subobject B in D2. Therefore

```
  pb1->f();              // has undefined behavior
```

Does the calling function know that the value of pb1 may have changed in mutate? Yes, this is a strait C aliasing question. pb1 has its address taken, its value may be changed by function calls. However, since the type (dynamic) of the object referred to by pb1 has changed, the result of the expression has undefined behavior.

Proposal:

If the pointer in a new-placement expression has type T1* and the new expression creates an object of type T2, referring to the original object has the same effect has referring to an object that has been destroyed.

In particular, the pointer of type T1* can only be used in limited ways. Using the pointer as a T1* is no longer valid. However, the pointer still points at valid memory and using the pointer as a pointer to the memory where the object was located '(void *)p' is well-defined. That is, such a pointer cannot be used to refer to any non-static (data or (virtual or non-virtual) function) members of the object of type T1 (i.e. doing so results in undefined behavior). However, such a pointer can be used to access other objects. For example, the pointer can be used to access static data members or call static member functions for the type T1.

2.2 new(this) in a member function (to a different type)

Similarly:

```
  class B {
    virtual void f();
    void mutate();
    virtual ~B();
  };
  class D1 : public B { void f(); };
  class D2 : public B { void f(); };

  void B::mutate() {
     this->~B();
     new (this) D2;
  }

  void* p = malloc(sizeof(D1) + sizeof(D2));
  B* pb = new (p) D1;
  pb->mutate();
  pb->f();              // has undefined behavior
```

Is it legal to use placement new in a member function to allocate an object at the memory location where the current object resides if the new object allocated does not have the same type as the 'this' pointer?

Proposal:

Same resolution as 2.1

There are obvious optimizations that must be given up if a member function is allowed to allocate an object of a different type at the 'this' address location. Implementation could no longer assume that the object size and layout, (i.e. pointers to virtual base classes) and polymorphic behavior (virtual function table) are unchangeable within a member function. The cost of loosing these optimizations is too great.

2.3 'new(this) T' if T is a base class type

Example:

```
class A { };
class B : public virtual A { };
class C : public virtual A { };
class D : public B, public C { };

void foo() {
  D d;
  C* cp = &d;
  new(cp) C;  //1: well-defined ??
}
```

As explained in 1.4, during the conversion C* --> void*, the value of cp might change. It is adjusted so that, after the conversion, cp points at the start of the memory location where the object of type C resides as if cp pointed at a complete object of type C. That is, the new object created by the new expression might be allocated at a memory location different from the start of the subobject of type C.

As explained in 1.4, an object of type C that is a base subobject is not guaranteed to have the same size and the same layout as a complete object of type C. The new expression creates an object of type C and this new object has the size and layout of a complete object of type C.

That is, the new expression on line //1 may have clobbered d's subobject C and other subobjects of d as well.

However, line //1 itself has well-defined behavior if cp points at enough storage to hold a complete object of type C. It is the action of referring to d or any of its subobjects after line //1 that results in undefined behavior.

Proposal:

If the pointer in a new-placement expression has type T1* and the new expression creates an object of type T2, (even if T1 is a base class type of T2), referring to the original object or any of its subobjects after the object of type T2 has been created has the same effect has referring to an object that has been destroyed.

That is:
```
new(cp) C; // has undefined behavior
```

2.4 'new(this) T' if T is the complete object type

Example:

```
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); //1: well-defined ??
    }
    return *this;
}

C c1;
C c2;
c1 = c2; //2: well-defined ??
c1.f();  //3: well-defined ??
```

The code on line //1 has well-defined behavior since '(void*)this' points at a storage area that is large enough to hold an object of type C.

The code on line //2 has well-defined behavior.
According to 1.2, the code works since:
- the complete object referred to by 'this' is of type C and the conversion 'this' --> void* yields a pointer that points to the start of the storage location where the complete object resides.
- The object created by placement new is of the same type (dynamic type) as the complete object pointed at by 'this'.

Proposal:

If the pointer in a new-placement expression has type T* and the pointer points to a complete object of type T (dynamic type) and the new expression creates an object of type T, referring to the object T (or any of its members or base classes) after the new expression has completed is well-formed.

That is:
```
c1.f();  //3: well-defined
```

Another similar example:
```
struct T { T* p; };
T::T() : p(new(this)T) { } // well-defined behavior
```

2.5 placement new on class members and array elements

Example:

```
struct X {
  Y y;
  Z z;
};

X x;
(&x.y)->~y();
new (&x.y) Y;
// is referring to x or x.y well-defined ??
```

Proposal:

For the purpose of the rule described in 2.4, class members and array elements are considered "complete objects".

That is:

```
new (&x.y) Y;
// referring to x or x.y is well-defined
```

Another example:

```
T t[5];
T* p = &t[3];
t[3].~T();    // t[3] and p become pointers to memory
p = new(p) T; // p and t[3] refer to T objects
//  referring to t[3] is well-defined
```

3. Destruction for objects allocated with placement new

3.1 When does destruction yield undefined behavior?

Example 1:

```
{
  T t;
  new (&t) T ;
  // according to 2.4, using t as a T is OK.
} //1
```

Rule 2.4 indicates that referring to t as a T after the new expression ’new (&t) T’ has completed is well-formed. In particular, calling T’s member functions (including T’s destructor at block exit as in this example), is well-formed.

Example 2:

```
{
  T t;
  new (&t) X ; // ok if t has enough storage to hold an X

  &t;          // ok: the memory holding t is still valid

  t.f();       // referring to a member of a T: undefined behavior
               // See rule 2.1
} // t is not a T anymore: undefined behavior
```

Rule 2.1 indicates that if a new expression creates an object of type T2 at the storage location where an object of type T1 resides, referring to the original object has the same effect as referring to an object that has been destroyed. In particular, if the original object is used to call T1's member functions (including T1's destructor at block exit as in this example), the program results in undefined behavior.

Example 3:

However, if placement new throws an exception then T (or parts of T) will be destroyed twice when the block //1 exists.

12.4 p13 already says that destroying an object more than once results in undefined behavior. This rule implies that example 1 has undefined behavior.

Example 4:

```
{
  T t;
  new (&t) X ; //1: OK if t has enough storage to hold an X
  new (&t) T ;
} // OK
```

Same as example 2.

3.2 When can destruction be skipped?

```
void h() {
   T* pt = new T;
   /* ... */
   new(pt) X;
}
```

What kind of type must T be for this program to have well-defined behavior?

Proposal:

If type T has a non-trivial destructor, an object of type T must be destroyed before the memory in which the object resides can be reused or released.

This ensures that if a type has a destructor with side-effects (i.e. a destructor that updates global objects, that releases certain program resources, etc) and if the call to the destructor is omitted, the program results in undefined behavior.

4. const objects

The const qualifier can influence the properties of the memory in which an object resides. Implementations are allowed to "color" memory in which const objects reside and "uncolor" the memory when it is deallocated. The implementation can put some const objects with constructors and destructors in read-only memory if it can figure out that the constructor and destructor do not modify the objects.

The model I propose for this behavior is to say that making the memory write protected is the implementation's responsibility and not that of the constructor. Similarly releasing the write protection on the memory is the implementation's responsibility and not that of the destructor.

The model I have in mind is the following:
a. once the constructor has completed, the memory in which a const object resides becomes write protected.
b. before a destructor starts, the write protection on the memory in which a const object resides is released.

I assume that action a. only takes place after the construction of the complete object (i.e. one that is not a member and not a base) has completed and that action b. only takes place before the destruction of the complete object (one that is not a member and not a base) has started.

The WP does not currently support this model:
5.3.4 p20 says:
"Whether the allocation function is called before evaluating the constructor arguments, after evaluating the constructor argument but before entering the constructor or by the constructor itself is unspecified."

Proposal

If the model above is correct, the allocation function cannot be called by the constructor itself. 5.3.4 p20 needs to be rewritten as follows:

"Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor argument but before entering the constructor is unspecified."

Since the memory in which const objects reside may be write protected, users should not be allowed to create new objects using placement new at the memory location where a const object resides.

```
{
  const T t;
  (&t)->~T();  // OK, memory still write protected
  new (&t) T ; // undefined behavior
}
```

Proposal
---

placement new cannot be used to create an object at a memory location where a const object resides.

5. Malloc and free

5.1 What kind of objects can be malloced?

```
void h() {
   T* pt = (T*)malloc(sizeof(T));
   /* ... */
}
```

What kind of type must T be for the expression calling malloc to be well-formed?

Proposal:

If type T has a non-trivial constructor, T's constructor must be called to create an object of type T; otherwise, the behavior is undefined. Objects of type T with dynamic storage duration can only be created by calling operator new.

5.2 using free instead of delete

If T is a type with a non-trivial destructor, does the following have well-defined behavior?

```
void h() {
   T* pt;
   pt->~T(); // calls T's destructor
   free(pt); // Free the memory
}
```

Proposal:

Given an object with dynamic storage duration of a type with a non-trivial destructor, when destroying the object, the destructor must be called for the object before the memory in which the object resides is freed (that is, the program must behave as if a delete expression was used to destroy the object).