```
Accredited Standards Committee X3       Doc No: X3J16/96-0125   WG21/N0943
Information Processing Systems          Date:   May 28th, 1996
Operating under the procedures of       Project: Programming Language C++
American National Standards Institute   Ref Doc:
                                        Reply to: Josee Lajoie
                                                  (josee@vnet.ibm.com)
```

```
+-----------------------------------------------+
| Object Model Issues and Proposed Resolutions |
+-----------------------------------------------+
```

613 - What is the order of destruction of objects statically initialized?

```
613 - What is the order of destruction of objects statically
      initialized?
======
```

3.6.3 [basic.start.term], para 1:
"These objects [initialized objects of static storage duration declared at block scope or namespace scope] are destroyed in the reverse order of the completion of their constructors."

Given:

```
    struct A { int i; ~A(); };
    A a = { 1 };
```

If an implementation decides to initialize a.i "statically", when must the implementation destroy a.i? i.e. what does it mean in such cases to destroy a.i "in reverse order of the completion of their constructors"?

Possible solutions:

Solution 1):
It is unspecified in which order these objects are destroyed.

Solution 2):
These objects are destroyed after the objects declared in the same translation unit and initialized dynamically are destroyed; they are destroyed in the reverse order in which their definition appears in the translation unit.

Solution 3):
These objects are destroyed as if they had a constructors: the destruction for such an object follows the destruction of objects later defined in the same translation unit but precedes the destruction of objects defined earlier in the same translation unit.
i.e.

```
     struct C {
       C();
       ~C();
     };
     struct D {
       D();
       ~D();
     };
     struct X {
       ~X();
     };
     C c;
     X x;
     D d;

     d is destroyed before x, x is destroyed before c.
```

Proposed resolution:

Adopt 3).
I believe solution 3) is the most intuitive for users.

```
    I do not believe the costs of implementing 3) are large enough to
    justify implementing one of the other solutions.

    3.6.3 para 1 needs to be modified to say:
    "These objects [initialized objects of static storage duration
     declared at block scope or namespace scope] are destroyed in the
     reverse order of the completion of their constructor or, if such
     objects do not have constructors, in the reverse order of the
     appearance of their definition."

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

639 - What is the lifetime of declarations in conditions?
======

  Jerry Schwarz asks the following:
      >
      > struct T { T(int); ~T(); operator bool() const; /*...*/ };
      >
      > void f(int i)
      >    {
      >    while (T t = i) { /* do something with 't' */ }
      >    }
      >
      > How often is t constructed/destroyed?

  Another example:
    for ( T *p = first;
          T *next = p->next();
          p = next )
        { p->val = 1; }

    How often is next constructed/destroyed?

  Possible solutions:
  -------------------
    Solution 1):
      Constructed and destructed at each iteration of the loop.

    Solution 2):
      Only once, when the loop is entered/exited, making the loop
      equivalent to:
      {
          T t = i;
          while (t) { /* do something with 't' */ }
      }

  Proposed resolution:
  --------------------
    I have a slight preference for solution 1).
    Variables in conditions are "internal" to the loop and it seems
    to make the most sense if they are initialized/destroyed at every
    iteration of the loop.

    Add at the end of 6.4[stmt.select] paragraph 3:
      "An object defined by a declaration in a condition is initialized
       each time the condition is evaluated.  If this object is of a
       class type with a non-trivial destructor, the object is destroyed
       at the end of the loop, at every iteration of the loop."

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

635 - local static variable initialization and recursive function calls
======

  6.7[stmt.dcl] para 4:
    "A local object with static storage duration not initialized with
     an integral constant-expression is initialized the first time
     control passes completely through its initialization."

  Neal Gafter asks:
  > Given:
  >     int foo(int i) {
  >             if (i == 0) return i;
  >             static int x ( foo (i-1) );
  >             return x;
  >     }
  >     ... foo (10) ...
  >
  > What is the value of x after it has been initialized?

  Proposed Resolution:
  --------------------
    I believe the "completely" in the sentence above already indicates
    what the answer should be. But it probably should be made clearer:

      "A local object with static storage duration not initialized with
       an integral constant-expression is considered initialized upon
       the completion of its constructor or once its initialization has
       completed."

   So, in the example above, x should be initialized with the value 0.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

655 - When is storing into another union member ill-formed?
======

  Bill Gibbons indicates the following:
  > Here is a program which is ill-formed in ISO C, but I cannot find
  > any wording in the C++ working paper which would make it ill-formed
  > in C++:
  >
  > union {
  >     struct A {
  >         double w;
  >         long double x;
  >     } a;
  >     struct B {
  >         long double y;
  >         double z;
  >     } b;
  > } u;
  >
  > int main() {
  >     u.b.y = 0.0;
  >     u.a.x = u.b.y;
  > }
  >
  > ISO C disallows this because of the overlap.  Since the
  > lvalue => rvalue conversion of u.b.y occurs before u.a.x is
  > modified, this code would appear to be valid C++.
  >
  > If the members were aggregate instead of scalar types, this would be
  > implicitly ill-formed.
  >
  > For example:
  >
  > struct tag { int x[1000]; int y[1000] };
  >
  > union {
  >     struct A {
```

```
  >         struct tag w;
  >         long double x;
  >     } a;
  >     struct B {
  >         long double y;
  >         struct tag z;
  >     } b;
  > } u;
  >
  > Once the first array element is copied, the entire union member
  > from which it came becomes invalid - because something has been
  > stored into another union member.  So the usage is already
  > ill-formed for aggregates.

  i.e., 9.5[class.union] para 1 says:
    "In a union, at most one of the data members can be active at any
     time, that is, the value of at most one of the data members can be
     stored in a union at any time."

  > But what about scalars?  In the original example the source and
  > destination overlap, but does the execution model say that an entire
  > scalar is fetched from memory before the store begins?
  > Or should C++ have the same restriction on overlap as ISO C?

  Proposed Resolution:
  --------------------
    I believe C++ should say what C says.

    I couldn't find the rule in the C standard.
    1.8[intro.execution] should probably say something like this:
      "If two objects overlap, and the value of the first object is
       accessed to determined the value to store in the second object,
       the value stored in the second object is unspecified."

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

624 - class with direct and indirect class of the same type:
      how can the base class members be referred to?
======

  10.1 [class.mi] para 3 says:
    "[Note: a class can be an indirect base class more than once and can
     be a direct and indirect base class.]"

  The WP should either:
  1) describe how the base class members can be referred to, how
     conversion to the base class type is performed, how initialization
     of these base class subobjects takes place, or
  2) say that declaring a class as a direct and indirect base class is
     ill-formed.

  Proposed Resolution:
  --------------------
    I prefer solution 2).
    This is of little use and not worth the effort it would require to
    describe the semantics of initialization, access of the base class
    members, and the semantics of the derived to base conversion when an
    object has both a direct and indirect base class of the same type.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

598 - Should a diagnostic be required if an rvalue is used in a
      ctor-initializer or in a return stmt to initialize a reference?
======

  12.2[class.temporary] p5:
    "A temporary bound to a reference in a constructor’s ctor-initializer
     (12.6.2) persists until the constructor exits. ...
     A temporary bound in a function return statement (6.6.3) persists
     until the function exits."

  Tom Plum indicates:
    > This actually means that there is no reliable way to initialize a
    > reference member or a return value of reference type with an
    > rvalue expression.  Given that, a diagnostic should be required.

  Proposed Resolution:
  --------------------
    Do as Tom says.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

536 - When can objects be eliminated (optimized away)?
======

  12.8[class.copy]p15 says:
    "Whenever a class object is copied and the original and the copy
     have the same type, if the implementation can prove that either
     the original object or the copy will never again be used except
     as the result of an implicit destructor call, an implementation
     is permitted to treat the original and the copy as two different
     ways of referring to the same object and not perform the copy at
     all."

  ISSUE 1:
  --------
  However, this is in clear contradiction with other WP text:
  3.7.1[basic.stc.static] says:
    "If an object of static storage duration has initialization or a
     destructor with side effects; it shall not be eliminated even if
     it appears to be unused."

  3.7.2[basic.stc.automatic] says:
    "If a named automatic object has initialization or a destructor
     with side effects; it shall not be destroyed before the end of its
     block, nor shall it be eliminated as an optimization even if
     appears to be unused."

  Subclause 12.8 says that objects may be optimized away while
  subclause 3.7 says that static and automatic objects with
  initialization or a destructor with side effects cannot be optimized
  away.  Which one is right?

  Many have suggested different ways to resolve this difference:

  Andrew Koenig [core-5975]:
    > The correct way to resolve the contradiction is to say that copy
    > optimization applies only to local objects.

  Patrick Smith [core-6083]:
    > 1) Just weaken 3.7.1 and 3.7.2 so they can be overridden by the
    >    copy constructor optimization.
    >
    > 2) Restrict the copy constructor optimization to only eliminate
    >    temporaries representing function return values.
    >
    > 3) Require the programmer to explicitly mark the classes for
    >    which the copy constructor optimization is permitted even
    >    though it would violate 3.7.1 or 3.7.2.
    >
    > 4) Require the programmer to explicitly mark the classes for
    >    which the copy constructor optimization is not permitted when
  >    it would violate 3.7.1 or 3.7.2.
```

ISSUE 2:

Jerry Schwarz in core-5993:

```
  > What may be of concern is not side effects in general, but resource
  > allocation.  E.g. if Thing is intended to obtain a lock that is
  > held until it is destroyed, then you do indeed have to be careful
  > about the semantics you give to the copy constructor.
  >
  >   {
  >        Thing outer ; // get the lock
  >        {
  >            Thing inner = outer ; // copy constructor increments
  >                                  // count on lock.
  >
  >            // do stuff that requires the lock
  >            inner.release() ;  // decrement count
  >            // do stuff that doesn't require the lock
  >        }
  >        // do stuff that still requires the lock.
  >   }
  >
  > The optimization allows outer and inner to be aliased, and the
  > explicit release in inner may cause the lock to be released too
  > early.
```

Is Jerry's concern worth worrying about?

Two possible resolutions were proposed:

Jerry suggested the following:

```
    > When we introduced the "explicit" keyword I remember considering
    > what it would mean on copy constructors and thinking about the
    > possibility that it would suppress this optimization.
```

Jason Merrill proposed in c++std-core-5978:

```
    > Perhaps the language in class.copy should be modified so that it
    > only applies when the end of one object's lifetime coincide with
    > the beginning of its copy's lifetime.
```

Proposed Resolution:

The core WG discussed this in Tokyo, and though we didn't agree on a resolution yet, the WG wanted very much the optimization in the return value case to be allowed by the standard. The questions that remain:

1) should the optimization be allowed for any object with automatic storage duration?
   i.e.

```
     Thing outer ;
     {
         Thing inner(outer);  // can inner be eliminated?
     }
```

   -> Less folks were convinced that this was needed.

2) should the optimization be allowed for any object with static storage duration?
   -> Most folks believed that this was a bad idea.

I believe allowing the optimization for more than the return value optimization causes more trouble than it is worth. 12.8 should be rework to only allow this optimization in the return value case.
i.e.

```
    class A { };
    main() {
      A x = f();
      A y = g();
    }

    // example 1
    A f() {
      A a;
      return a;    // a may never be created,
                   // return value created into x directly
    }

    // example 2
    A g() {
      return A();  // constructor call may be omitted
                   // return value created into y directly
    }
```

Proposed new words:
12.8[class.copy]p15 should say:

"Whenever a function has a return type that has class type, if an object, either with automatic storage duration or a temporary, that has the same type as the return type is created just for the purpose of providing a value to be copied and returned by the function, and if this object is never otherwise used in the function except as the result of an implicit destructor call (12.4, _class.dtor_), an implementation is permitted to omit the creation of this object and the associated copy performed by the return statement and is permitted to create the return value using direct initialization (8.5, _dcl.init_)."

The example in para 15 also needs to be modified to take into account this change.

3.7.2[basic.stc.automatic] should say:

"If a named automatic object has initialization or a destructor with side effects ... it shall not be eliminated as an optimization even if appears to be unused, except in the case of the return statement copy optimization (12.8, _class.copy_)."

. . . .