**Accredited Standards Committee***
**X3, INFORMATION PROCESSING SYSTEMS**

**Doc No:** J16 / 99–0014
WG21 / N1191
**Date:** 23 February 1999
**Project:** Programming Language C++
**Reply to:** Andrew Koenig
AT&T Research
180 Park Avenue, room E265
PO Box 971
Florham Park, NJ 07932–0971 USA
`ark@research.att.com`

# Defect report: definition of *default-initialization* (revised)

*Andrew Koenig*

**Revision history**

This document is a revision of, and supersedes, 98–0018 / N1161, dated 25 August 1998.

**Summary**

The standard defines the concept of *default-initialization* for an object of type `T` in subclause 8.5 [dcl.init], paragraph 5, as follows:

- if `T` is a non-POD class type (clause 9 [class]), the default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor);
- if `T` is an array type, each element is default-initialized;
- otherwise, the storage for the object is zero-initialized.

This definition is appropriate for local variables, but not for objects that are initialized as a result of executing expressions of the form `T()`, because the objects yielded by such expressions will be copied immediately, and should therefore have values that are assured of being copyable. To this end, I propose adding the following new text to 8.5, paragraph 5:

> To *value-initialize* an object of type `T` means:
>
> - if `T` is a class type (clause 9 [class]) with a user-declared constructor (12.1), then the default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor);
> - if `T` is a class type without a user-declared constructor, then every non-static data member and base-class component of `T` is value-initialized;
> - if `T` is an array type, then each element is value-initialized;
> - otherwise, the storage for the object is zero-initialized.

In addition, I propose to change ‘‘default-initialization’’ to ‘‘value-initialization’’ in 5.2.3 paragraph 2.

The balance of this note explains the effects of this change and why I think it is important.

* *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington DC 20005

**Ancient history**

Once upon a time, an AT&T compiler developer named Laura Eaves asked me: ‘‘What should be the value of `int()`?’’ My first thought was that it should be the same value as `x` has after saying

```
int x;
```

but I soon realized that that definition would not do. The reason is that `x` has an indeterminate value (assuming that it is a local variable), but we don’t mind that `x` is indeterminate, because we are presumably going to assign a value to `x` before we use it. In contrast, `int()` had better not have an indeterminate value, because copying such a value has an undefined effect. It would be silly to forbid a compiler from flagging `int()` during compilation, only to allow it to flag it during execution!

So our compiler defined `int()` as zero, and similarly for other built-in types.

About the same time, I was working on an associative-array class library, and I had the following problem:

```
Map<int, int> m;                // my class was called Map, not map

int x = m[42];
```

What should the value of `x` be at this point? There is ample experience to argue that it much more useful for map elements to acquire a known initial value by default, and that that value should be zero for numeric types. For example, such a policy makes it possible to count how many times each distinct word occurs in the input by writing

```
Map<string, int> word_count;
string word;

while (cin >> word)
        ++word_count[word];
```

Unless the `Map` class initializes elements to zero automatically, this program becomes much harder, because it becomes necessary to test whether each element already exists before we try to increment it, and to initialize the element if it does not yet exist.

Even though the compiler guaranteed that `int()` was zero, the language definition at the time did not, so I defined my map class to use a static variable to initialize map elements, along the following lines:

```
template<class Key, class Value> class Map {
        // ...
        static Value default_val;
        // ...
};

// and, in a separate file:
template<class Key, class Value>
Value Map<Key, Value>::default_val;     // static, hence automatically initialized
```

This strategy gave me the behavior that I wanted, but with the unfortunate side effect of relying on the order of static initializers in the case where the value type was a class with a constructor.

**More recent history**

Let us now move forward a few years. In one of the early standards meetings, I brought up the issue of `T()`. There was a great deal of discussion, much of which centered around the question of what it would cost to initialize a copy of `T()` if `T` were defined as, for example,

```
struct T {
      int x[100000];
};
```

Everyone agreed that

```
T t;
```

should not cause any initialization, lest C++ be put at a performance disadvantage compared with C. But Laura Eaves' argument was telling in the case of `T()`: Either all the elements of `T().x` must be defined, or else copying the value of `T()` (which is the only thing you can do with it!) has an undefined effect.

The outcome of this discussion was a change in the meaning of `T()`. The November, 1993 working paper says (in expr.type.conv) that ''the result is an unspecified value of the specified type.'' The corresponding phrase in the March, 1994 working paper says that ''the result is the default value given to a static object of the specified type.''

This definition persisted until the July, 1995 working paper, when it was changed to its current form. I do not recall any discussion about the change, so I believe that the change was intended as a clarification, rather than as a normative change. However, it does change the behavior, and in a highly dangerous way.

**The hazard**

The change in behavior affects classes that are not PODs, but which look like PODs from their authors' perspective. For example:

```
struct Foo {
      int x;
};
```

is a POD, but

```
struct Bar {
      int x;
      string s;
};
```

is not. In order to see that `Bar` is not a POD, we must look at the definition of class `string`. Whether or not a class is a POD is a low-level concept, the primary purpose of which is to determine whether library functions, such as `memcpy`, that deal in raw memory, will work with objects of that class.

Under the 1994 definition, `Foo().x` and `Bar().x` are both guaranteed to be zero. Under the July, 1995 definition, `Foo().x` is still guaranteed to be zero, but `Bar().x` is undefined. Indeed, merely copying the value of `Bar()` is undefined, which effectively renders `Bar()` useless.

Now let us turn to the definition of `operator[]` for library class `map`. Subclause 22.3.1.2 [lib.map.access] defines `map` member

```
T& operator[](const key_type& x);
```

as having the same effect as

```
(*((insert(make_pair(x, T()))).first)).second
```

Note the use of `T()`. It is there to provide a default value in case the map element does not already exist, but it is evaluated regardless of whether the element exists.

The implication of this definition is as follows:

```
Foo foo;                              // foo.x is implicitly zero
Bar bar;                              // bar.x is implicitly zero

void f()
{
      map<int, Foo> m1;
      map<int, Bar> m2;

      m1[0] = foo;                 // OK
      m2[0] = bar;                 // Undefined!
}
```

The reason that the assignment to `m2` is undefined is that evaluating `m2[0]` is equivalent to evaluating

```
(*((insert(make_pair(0, Bar()))).first)).second
```

and the effect of copying `Bar()` as an argument to `make_pair` is undefined.

In fact, we do not even need maps to evoke this problem: We find the same difficulty with

```
vector<Foo> v1;                       // OK
vector<Foo> v2(100);                  // OK
vector<Bar> v3;                       // OK
vector<Bar> v4(100);                  // Undefined!
```

The problem here is essentially the same: Giving a size to `v4` is a request to initialize its elements to copies of `Bar()`, and copying `Bar()` has undefined effect.

If you think that class `Bar` is contrived, consider this alternative:

```
struct Customer {
      int account_number;
      string name;
};
```

This is an example of the kind of data structure that surely comes up all the time in business applications. Can it really be right to prohibit C++ programmers from using vectors of `Customer` objects—especially when built-in arrays of `Customer` objects are allowed?

**Efficiency**

When I first noticed this problem, I thought that the right solution was for the value of `T()` to be defined by first zero-initializing and then default-initializing the object. However, that strategy pays a penalty that I now think is unnecessary:

```
class T {
public:
      T() {
            for (int i = 0; i < 100000; ++i)
                  a[i] = -1;
      }

private:
      int a[100000];
};
```

It is reasonable to expect that evaluating `T()` does not involve zeroing all the elements of `T::a` and then setting them to `-1`. After some thought, I concluded that the problem is relevant only to data members that are out of reach of any constructor, which does not include `T::a` in this example. That is, once a user writes a constructor for a class, it is entirely reasonable to expect that constructor to initialize all the data members. The problem case is only when the author of a class did *not* write any constructors, but the class acquired a constructor because a member or a base class happens to have one. In other words:

```
struct Baz {
      Baz() { }
      int x;
      string s;
};
```

Here, the author of `Baz` had the opportunity to initialize `x` and did not do so. I consider that omission to be an ordinary programming error, and it does not bother me that copying `Baz()` is undefined. If the author of `Baz` wanted to define copying, the author should either written a copy constructor that did not copy `x`, or ensured that the other constructors all initialized `x`. It been ever thus. My proposed change, therefore, does not define `Baz().x`.

Nor does it define `bar.x` in

```
void h()
{
      Bar bar;

      // bar.xf2 is undefined here
}
```

People expect local variables to be uninitialized unless they explicitly initialize them, and I am not proposing to change that.

**Conclusion**

Programmers who are using C++ for ordinary applications are likely to expect to be able to write

```
struct Customer {
      int account_number;
      string name;
};
```

and then define objects such as

```
int main()
{
      vector<Customer> vc(100);
      // ...
}
```

It is likely to come as a shock that the standard does not sanction this seemingly obvious usage. At present, this definition has undefined effect during execution—even though changing `Customer::name` from a `string` to a `char*`, for example, would make the definition of `vc` well defined.

Indeed, a protracted debate resulted in making this usage legal in November, 1993. When it was made illegal again in July, 1995, I believe that the change was inadvertent. The notion that `Customer().account_number` is undefined, even though `int()` is guaranteed to be zero, is too much of a violation of the principle of least surprise to be allowed to stand.