X3J16/93-0157, WG21/N0364

Defined Value for `T()`

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Instead of returning an unspecified value, `int()` should return `0`. In general, the value of `T()` should be defined for every type `T`. This is a pure extension. It is trivial to implement. In addition is provides a solution to an outstanding core issue.

1 The Proposal

The value of `T()` is undefined unless `T` has an explicitly defined constructor taking no arguments.

I'd like to see `T()` have the value that a static `T` has by default. For example, the value of `int()` would be `0` because that's the default value of `i` defined as

```
        static int i;
```

This will allow me to give class members a definite value

```
template<class T> class C {
        T a;
        int i;
        C() : i(0), a(T()) { }
};
```

Without having `T()` defined, I must to do something like this:

```
template<class T> class C {
        T a;
        int i;
        C() : i(0), T(dT()) { }
        static T dt();
};

template<class T> T C::dt() { static T t; return t; }
```

or leave the member `a` uninitialized. Defining a static function returning the value of a static object is a neat technique and very useful, but giving `T()` a defined value seems cleaner. As ever, `T()` would be illegal if `T` has other constructors, but no default constructor.

Working Paper Change

In §5.2.3 ‘‘Explicit type conversion (functional notation)’’ change

If the type is a class, the class must have a default constructor (§12.1) (otherwise the expression is erroneous) and that constructor will be called; otherwise (the type is not a class) the result is an unspecified value of the specified type.’’

to

If the type is a class with a default constructor (§12.1) that constructor will be called; otherwise, the result is the default value given to a static object of the specified type.

2 Discussion

Consider:

```
class X {
        int i, j, k;
public:
        X() :i(int()), j() { }
};
```

The constructor initializes `i` to `0` and leaves `j` and `k` uninitialized. That is, the value of `int()` is a property of the expression `int()` and not some implicitly defined special constructor.

This proposal would also clear up a tricky core-issue. Consider:

```
void f()
{
        T i = T();
        T j = i;
        // ...
}
```

May an implementation terminate the program at the point where an attempt is made to copy `i`? Given the current rule, `i` is known to contain an unspecified value, and we have no guarantee that an unspecified value can be copied. For example, the value copied might be an IEEE signaling NaN.

In an ideal world we might give every object a default value and be done with this problem of uninitialized variables and undefined values. However, implicit default initialization can get very expensive. Consider

```
struct Thing {
        int a[100000];
        int high_water_mark;
};
```

If by default `int()` was called for every member of `Thing::a` to give it a default value, then

```
        new Thing;
```

and

```
        Thing x;
```

would be very expensive. In particular, it would put C++ at a systematic performance disadvantage compared to C in the common C/C++ subset.

Compatibility
Every currently legal program remains legal.

Implementation
For built-in types the simplest implementation of the current rule is to return some specific value. The obvious specific value is `0`. That is what some (all?) current implementations do and that's what the proposal says they should do.

For a class `T` without a default constructor `T()` is currently illegal. To implement it as suggested is easy. The trick of using a function

```
T dt() { static T t; return t; }
```

demonstrates a general implementation technique. Most implementations will be able to apply significant optimizations.