#X3J16/96-0031 WG21/N0849

Relaxing the Rules for Void

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Writing function-like templates often requires a duplication of code to handle value returning and non-value returning behavior. To eliminate this duplication, I propose to allow a `void` expression to be used to represent ''no value'' in a return statement and in a function call where no value is allowed.

1 Introduction

We write

```
int f();

int g() { ... return f(); }
```

and

```
void f1();

void g1() { ... f1(); }
```

but this is an error:

```
void f2();

void g2() { ... return f2(); }
```

I propose to allow this last example. That is, where no return value is allowed, I suggest that an expression of type `void` should be allowed. Similarly, I propose to allow calling a function that takes no arguments with an argument of type `void`. This involves lifting a semantic restriction, but not adding new syntax or new semantics. For that reason, I consider this a truly minimal extension; the smallest extension ever proposed.

The reason to consider this relaxation is to ease the writing of templates. For example:

```
template<class T> T g(T (*f)()) { ... return f(); }

int ff();

void h()
{
        g(ff);  // ok
        g(h);   // currently error, proposed ok
}
```

I have run into this problem several times. See my proposal for `mem_fct()` (#X3J16/96-0030,WG21/N0848) for a real example, and for the amount of work needed to circumvent this problem. The proposed relaxation halves the number of templates you need to write when a pointer to function is involved by allowing the style above to cover both the `void` and value returning case.

2 Details

I propose to allow a ‘‘`void` parameter’’ or a ‘‘`void` return value’’ to be initialized by a a ‘‘`void` value ’’. For example:

```
void f();

void g()
{
        f();            // ok as always
        return;         // ok as always

        f(g());         // proposed ok
        return f();     // proposed ok
}
```

Note that there is (as ever) no difference between

```
void f();
```

and

```
void f(void);
```

It would be possible to implement a slightly different extension that would also allow

```
void x = f();
```

I am *not* proposing that. I see no compelling need for ‘‘`void` objects’’ or ‘‘`void` values.’’

3 Working Paper Changes

In §6.6.3 [stmt.return] paragraph 2, replace the second sentence up to the semicolon with:

> An expression in an return statement in a function that does not return a value must be of type `void`; the expression is evaluated, but no value is returned. In a function that does return a value, an expression is required in a return statement

I believe that this formulation also clarifies that `return;` is ill-formed in a value returning function.

In §5.2.2 [expr.call] paragraph [3], add after the first sentence:

> An expression of type `void` can be given as an argument for a function that takes no arguments (_dcl.fct_); the expression is evaluated, but no value is passed.

4 Implications

The proposed relaxation can be implemented by a small localized change to the semantic checking in a compiler front end. It does not involve syntax changes or changes to code generators.

No other part of the language definition is affected.

The current library specification would not be affected by this relaxation. My proposal for `mem_fct` (#X3J16/96-0030,WG21/N0848) can be implemented without used of specialization given this relaxation.

5 Acknowledgements

Andrew Koenig made constructive comments on this proposal.