The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.1.2.2 Hosted environment

hosted environment

A hosted environment need not be provided, but shall conform to the following specifications if present. 158

Commentary

This is a requirement on the implementation. The standard defines a specification, not an implementation. Issues such as typing the name of the program on a command line or clicking on an icon are possible implementations, which the standard says nothing about.

C++

1.4p7 *For a hosted implementation, this International Standard defines the set of available libraries.*

17.4.1.3p1 *For a hosted implementation, this International Standard describes the set of available headers.*

Common Implementations

Most hosted environments provide the full set of functionality specified here. The POSIX (ISO/IEC 9945) standard defines some of the functions in the C library. On the whole the specification of this functionality is a pure extension of the C specification.

Coding Guidelines

This specification is the minimal set of requirements that a program can assume will be provided by a conforming hosted implementation. The application domain may influence developer expectations about the minimal functionality available. This issue is discussed elsewhere.

coding guidelines applications

footnote 8

8) The intent is that an implementation should identify the nature of, and where possible localize, each violation. 159

Commentary

The Committee is indicating their intent here; it is not a requirement of the standard that this localization be implemented.

The US government has also had something to say on this issue: FIPS PUB 160 (Federal Information Processing Standards PUBlication), issued March 13, 1991, said:

FIPS PUB 160, 10pc *The message provided will identify:*

— The statement or declaration that directly contains the nonconforming or obsolete syntax.

Paragraph c also included a fuller definition and requirements. A change notice, issued on August 24 1992, changed these requirements to:

Page 3, paragraph 10.c, Specifications: remove paragraph 10.c.

FIPS PUB 160 no longer contains any requirements on the localization of violations in C source by implementations.

Common Implementations

macro replacement

Violations detected during the expansion of a macro invocation can be difficult to localize beyond the original name of the macro being expanded and perhaps the line causing the original invocations.

Many implementations handle C syntax by using a LALR(1) grammar to control an associated parser. Grammars having this property are guaranteed to be able to localize a syntax violation on the first unexpected token[1] (i.e., no possible sequence of tokens following it could produce a valid parse).

Violations arising out of semantic constraints are usually localized within a few tokens.

Translators tend to generate a diagnostic based on a localized cause for the constraint violations. Experience has shown that assuming a localized cause for a violation is a good strategy (it is often also the simplest to implement). For instance, in:

```
static char g;

void f(void)
{
char *v;

if (g)
   {
   int v;
     /*
      * Other code.
      */
   v = &g;
   }
}
```

the assignment to v is likely to be diagnosed as an incompatible assignment, rather than a misplaced **}** token. Many violations do have a localized cause. Looking for a nonlocalized cause requires a lot more effort on the part of vendors' implementations and is only likely to improve the relevance of diagnostics in a few cases. Most implementations generate diagnostic messages that specify what the violation is, not what needs to be done to correct it. In:

```
struct {
        int mem;
       } s;
int glob;

void f(void)
{
int loc;

loc = s + mem;
}
```

the implementation diagnostic is likely to point out that the binary plus operator cannot be applied to an operand having a structure type. One of the skills learned by developers is interpreting what different translators' diagnostic messages mean.

Example

```
#define COMPARE(x, y) ((y) < (x))

struct {
        int mem;
       } s;
int glob;

void f(void)
{
int loc;

/*
 * Where should an implementation point, on the following line,
 * as an indication of where the violations occurred?
 */
loc = COMPARE(glob, s);
}
```

Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. 160

Commentary

A valid program, as in a strictly conforming program.

C++

The C++ Standard does not explicitly give this permission. However, producing diagnostic messages that the C++ Standard does not require to be generated might be regarded as an extension, and these are explicitly permitted (1.4p8).

Other Languages

Most languages are silent on the topic of extensions. Languages that claim to have Java as a subset have been designed and compilers written for them.

Common Implementations

One syntax, or constraint violation, may result in diagnostic messages being generated for tokens close to the original violation. These may be caused by a translators' incorrect attempts to recover from the first violation, or they may have the same underlying cause as the first violation. In many cases these are localized and translation usually continues (on the basis that there may be other violations and developers would like to have a complete list of them).

Some translators support options whose purpose is to increase the number of messages generated. It is intended that these messages be of use to developers. For instance, the gcc option *-Wall* issues diagnostics for constructs which it considers to be potential faults. Some translators support an option that causes any usage of an extension, provided by the implementation, to be diagnosed.

implementation document

There is a market for tools that act like translators but do not produce a program image. Rather, they analyze the source code looking for likely coding and portability problems— so-called *static analysis tools*. The term *lint-like tools* is often heard, after the first such tool of their ilk, called lint (lint being the fluff-like material that clings to clothes and finds its way into cracks and crevices).

Implementation vendors receive a lot of customer requests for improvements in the performance of generated code and support for additional library functionality. The quality of diagnostic messages produced by translators is rarely given a high priority by purchasers. Experience shows that developers can adapt themselves to the kinds of diagnostics produced by a translator.

Example

Many translators now attempt to diagnose likely programmer errors. The classic example is:

```
if (x = y)
```

where:

```
if (x == y)
```

had been intended.

Experience suggests that developers will ignore such diagnostics, or disable them, if they are incorrect more than 40% of the time.

161 It may also successfully translate an invalid program. translated invalid program

Commentary

The term *invalid program* is not defined by the C Standard. Common usage suggests that a program containing violations of syntax or constraints was intended.

Before the spread of desktop computers, it was common for programs to be translated via a batch process on a central computer. The edit/compile/run cycle could take an hour or more. Having the translator fail to translate a program, even if it contained errors, was often considered unacceptable. Useful work could often be done with an invalid (only a part of it was likely to be invalid) but executable program while waiting for the next cycle to complete. Within multiperson projects a change made by one person can have widespread consequences. Having a translator continue to operate, while a particular issue is resolved, often increases its perceived quality.

Common Implementations

There have been several parsers that perform very sophisticated error recovery.[3] In the case of the C language, inserting a semicolon token is a remarkably effective, and simple, recovery strategy from a syntax violation (another is deleting tokens up to and including the next semicolon).

Prior to the invention of desktop computers and even before the spread of minicomputers, when translation requests had to be submitted as a batch job, vendors found it worthwhile investing effort in sophisticated syntax error recovery.[2, 3] The turnaround of a batch-processing system being sufficiently long (in the past, several hours was not considered unusual) that anything a translator could do to repair simple developer typing mistakes, in order to obtain a successful program build, was very welcome. Once the turnaround time on submitting source for translation became almost instantaneous, by providing developers with direct access to a computer, customer demand for sophisticated syntax error recovery disappeared.

Commonly recovered semantic errors include the use of undeclared objects (many implementations recover by declaring them with type **`int`**) or missing structure members (adding them to the structure type is a common recovery strategy).

Coding Guidelines

A guideline recommendation of the form “The translated output from an invalid program shall never be executed in a production environment.” is outside the scope of the guidelines.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1985.
2. R. P. Corbett. *Static Semantics and Compiler Error Recovery*. PhD thesis, University of California, Berkeley, June 1985. Report No. UCB/CSD 85/251.
3. P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software–Practice and Experience*, 25(6):657–679, 1995.