Doc No: X3J16/92-0030
WG21/N0107
Date: March 02, 1992
Project: Programming Language C++
Ref Doc: X3J16/90-105
Reply to: Philippe Gautron (gautron@rxf.ibp.fr)

Position Paper

A Proposal for an Assertion Mechanism based on Exceptions

Philippe Gautron
Université Paris VI
4 place Jussieu, 75252 PARIS CEDEX 05, France

March 1992

Abstract

This paper presents a C++ alternative to the **assert** facility provided by the standard C library. The proposed assertion mechanism is based on exceptions and its genericity on templates.

1 Introduction

The C assertion mechanism defines `assert` as a macro indicating that its expression argument is expected evaluating to true at the point of the call. If the assertion is false, a diagnostic comment is written on the standard error output and a call to the abort function is performed.

The `assert` macro is defined in the standard header file `<assert.h>` and refers to another macro, **NDEBUG**, which is *not* defined in this header file. Should the macro be defined as a function, the behavior would be undefined.

[Clamage 90] discusses different degrees in incorporating the ISO-C library into C++, from pure inclusion to complete rewriting. This same paper presents the `assert` C macro as a special case for the following reasons:

- the `NDEBUG` macro cannot be defined as a constant. The value of this macro may be defined at compile-time: compiling with the option `-DNDEBUG` effectively *deletes* the assertions from the program.
- its definition cannot be turned into an inline function. The diagnostic reported when the assertion turns out to be false includes the "stringification" of the assertion, the line number, and the file name at the check point of the assertion. Since macro substitution occurs *before* inlining, this information are expanded with their values at the declaration point within the inline function. In the absence of C++ run-time support for this information, the use of macros is thus unavoidable.
- ISO-C specifies that *no* assertion evaluation occurs when assertions are disabled. In order to achieve this goal, macro support is more definitive than relying on compiler optimization (such as constant expressions in control statement tests).

A standard mechanism owes itself to use the other standard tools: in our proposal, the standard stream I/O library will be used to write the diagnostic message.

Limitations
The names used in this paper are merely indicative.[1] The primary intention is to describe the techniques that can be used to implement the assertion mechanism. On the other hand, requiring (or not) a similar implementation to be in conformance with the standard is not discussed at all.

Note that since macros are used in C, the compatibility with C is not an issue: C assertions, when evaluating to false, will abort the whole program.

2 The Proposal

The overall purpose of the proposed implementation is twofold:

- to provide a default behavior similar to the C `assert` facility.
- to rely on specific C++ facilities, templates and exceptions, to provide a more generic and powerful support than simple macros.

What follows is the proposed implementation:

[1] Except for `assert`. Tony Hansen has suggested to use `eassert` or `Eassert` instead of `Assert` to refer to the exception version.

```
// -- file assert.h --

# ifndef __ASSERT_H
# define __ASSERT_H

# ifndef NDEBUG

# include <iostream.h>
extern "C" void abort();

// -- generic implementation
  template <class E>
    class __assert {

      public:
        __assert (int expr, const char *exp, const char* file, int line){
            if (! expr) throw E(exp, file, line);
        }

        __assert (void *ptr, const char *exp, const char* file, int line){
            if (! ptr) throw E(exp, file, line);
        }
    };

// -- specific C++ macro
# define Assert(expr, excep)\
         (__assert<excep> (expr, #expr, __FILE__, __LINE__))

// -- standard exception
  class Bad_assertion {
    public:
      Bad_assertion (const char *exp, const char* file, int line){
          cerr << "Assertion failed: " << exp << ", file " << file
               << ", line " << line << '\n';
          abort();
      }
  };

// -- C-like macro
# define assert(expr)    (Assert(expr, Bad_assertion))

# else /* !NDEBUG */

# define Assert(expr, excep)  (0)
# define assert(expr)         (0)

# endif /* NDEBUG */
# endif /* __ASSERT_H */
```

3 More about the Macros

3.1 The assert Macro

The C assert macro is intended to be used as the irrevocable detection of a program failure. A trivial example is null pointer testing, as in:

```
class String {
    // ...
  public:
    String (const char* p){
        assert (p != 0);    // C macro
        ...
    }
};
```

Validity of the expression is checked and a rudimentary message is printed in case of failure. The macro is intended to be used with traditional debuggers, that is debuggers supporting post-mortem debugging. §5.2 presents a C++ version of **assert**, quite similar to its C definition.

The sole difference between a C-like macro and the macro introduced in the previous section is the creation of an additional object to handle the Bad_assertion exception. Where this object is created (local, free store, pre-allocated memory) is not yet clearly specified (if not implementation-dependent). The **Bad_assertion** class is here used because it (or a similar standard class) is expected to be used in other contexts. Should this difference turn out to be a real problem, a C-like macro could be re-introduced with the Assert macro.

3.2 The Assert Macro

The primary goal of the **Assert** macro is to delegate to the caller responsibility for handling the failure (appropriate error messages, call to **exit** instead of **abort**, ...). A typical example is range checking of a subscript operator, as in:

```
class Vector_RC : public Vector {
    // ...
    class OutOfRange{
        int l;
      public:
        OutOfRange (const char*, const char*, int line) : l(line) {}
        int line (){ return l; }
    };
};

int& Vector_RC::operator [] (int index){
    Assert (index>=0 && index<size, Vector_RC::OutOfRange);
    // ...
}

int f (Vector_RC& v, int index){
    int elem;

    try {
        elem = v[index];
    }
    catch (Vector_RC::OutOfRange& e) {
        cerr << "Vector_RC: range checking failed: index=" << index
             << ", size= " << v.size()
             << ", line= " << e.line()
             << '\n';
        exit (-1);
   }

   return elem;
}
```

Since the exception is thrown before the program failure occur, the environment is not corrupted when the run-time flow returns to the caller.

If an exception is not caught (as could be OutOfRange above), a call to terminate is performed. The default behavior of terminate is to call abort. An uncaught exception resulting from a call to Assert will thus unwind the stack, unlike a call to assert. Calls to local destructors will be performed: this can alter the conditions under which the failure occurred.

4 Assertion Type

Overloading of the _assert constructor is used in our implementation to specify the type of the assertion. Assertion can be of type int, void* or of a type convertible to these types. Should expererience show these definitions to be incomplete, a more generic alternative could be used (see §5.3).

It is worth noticing that our first proposal requires us to relax in one way or another the exact match rule applied to (member) function templates.

5 Alternatives

This section examines three alternatives to the current proposal and one limitation of the current specifications of the pre-processor.

5.1 A Simple Macro

A simple alternative should be to take as such the C definition of `assert`. Of course, this should restrict its use to diagnose of program failure.

In [Koenig 89, p. 82-83], it is demonstrated that an implementation that should be reliable is not as reliable as expected at first sight. An implementation might look like:

```
// -- file assert.h --

# ifndef NDEBUG

# include <iostream.h>

extern "C" void abort();

# define assert(expr)    ((expr) || \
          (cerr << "Assertion failed: " << #expr << ", file " << file \
               << ", line " << line << '\n', \
           abort (), 0))
# else /* !NDEBUG */

# define assert(expr)         (0)

# endif /* NDEBUG */
```

Our macro differs from Koenig's proposal on:

- the type of the assertion. The type proposed by Koenig is `void`, as is the behavior defined by ISO-C when the `NDEBUG` macro is enabled:

  ```
  # define assert(ignore) ((void)0)
  ```

  It seems that we can rely on the C++ compiler to determine the exact type of the expressions. More, specifying a `void` type should explicitly restrict the use of `assert` to single statements.

- A `0` ends the macro definition after the call to `abort`. This trailing expression prevents presenting a void operand to the `||` operator.

Note that our implementation §2 does not present similar drawbacks since each macro we define will be substituted by a single statement.

5.2 A Function Template

[Stroustrup 91, p. 419] shows an example of an inline *function* template `Assert` that mimics the C `assert` macro:

```
// -- definition of Assert
  template <class T, class E>
    inline void Assert (T expr, E excep){
        if (! NDEBUG)
           if (! expr) throw excep;
    }

// -- example of the use of Assert
  class Bad_assertion {
    public:
       Bad_assertion ();
  };

  void f (void *p){
      Assert (p, Bad_assertion());  // <*** see below
  }
```

In this example, a `Bad_assertion` exception will be thrown if `p` is a null pointer.

This approach suffers from the following drawbacks:

- the object thrown by the exception is created *before* the call to `Assert`. This object will be created, whatever the evaluation of the assertion is. This contradicts our requirement (see §1) that no evaluation must occur when assertions are disabled. More, a call to `abort` *inside* the body of the constructor of `Bad_assertion` will cause a memory dump before testing the assertion.

  Should the place of `NDEBUG` be changed, for example:

  ```
  # ifndef NDEBUG

  template <class T, class E>
    inline void Assert (T expr, E excep){
        if (! expr) throw excep;
    }

  # else /* !NDEBUG */

  template <class T, class E>
    inline void Assert (T, E) {}

  # endif /* NDEBUG */
  ```

  the issue would still occur since the argument object will be created *prior* to any function call. And relying on compiler optimization in this particular case is not an appropriate technique.

- stringification of the assertion, line number and file name cannot be directly supported by the **Assert** function (see §1). Introduction of these informations might be possible (with contortions) although this would not solve the conceptual issue explained above.

5.3 Generalizing Assertion Type

A variation on our primitive proposal would be to make the type of the assertion a template parameter. The revised definitions would then be:

```
template <class T, class E>
  class __assert {

    public:
      __assert (T expr, const char *exp, const char* file, int line){
          if (! expr) throw E(exp, file, line);
      }
  };

# define Assert(expr, excep)\
        (__assert<expr, excep> (#expr, __FILE__, __LINE__))

# define assert(expr)    (Assert(expr, Bad_assertion))
```

The difference between the two proposals is that the former is based on constructor overloading whereas the latter is based on template instantiation: a new class will be created for each pair (assertion type, exception class).

5.4 A Limitation to the Pre-processor

Although macro definitions are not first-class functions, they can appear to be equivalent. This is (sometimes and a bit tricky) also true for calls to class constructors, as in:

```
class A {
    // ...
  public:
    A (int line, char *filename);
};

# define A() A(__LINE__, __FILE__);

void f (){
    A* a1 = new A();     // ok: A* a1 = new A(10, "t.c");
    A();                 // ok: A(11, "t.c");
    A a2;                // syntax error
}
```

Our first intention was to keep in the macro definitions a strict separation between the exceptions, used as template parameters, and the assertions, used as expressions. Something like:

```
template <class E> class __assert {
  public:
    __assert (int expr, const char *exp, const char* file, int line){
        if (! expr) throw E(exp, file, line);
    }
    // ...
};

# define Assert<excep>(expr)      __assert<excep> (expr, /* ... */)
# define assert(expr)             Assert<Bad_expression>(expr)
```

(Un)fortunately, the pre-processor knows nothing specific about templates. The definition of `Assert` above is fairly detected as an error. Making it legal would require the support of macro definitions with two lists of arguments, which is not reasonable.

6 Acknowledgements

This paper has benefited from discussions on the Library Working Group mailing list. Particular thanks to Tony Hansen for his contributions and his review of the first draft of this paper.

References

[Clamage 90] Stephen D. Clamage. *ANSI C Library Compatibility Issues.* Doc No: ANSI X3J16/90-0105, November 1990.

[Koenig 89] Andrew Koenig. *C Traps and Pitfalls.* Addison-Wesley, ISBN 0-201-17928-8, 1990.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language.* Second Edition. Addison-Wesley, ISBN 0-201-53992-6, 1991.