Object-Oriented Design and Programming

Programming with Assertions and Exceptions

Outline

What Are Assertions?

- Assertions are boolean expressions that serve to express the semantic properties of classes and member functions.
- Assertions are similar to the mathematical notion of a *predicate*.
- Assertions are tools for expressing and validating the correctness of modules, classes, and subprograms.

Four Purposes for Assertions

- Aid in constructing correct programs.
  - *e.g.*, specify input preconditions and output postconditions.
- Documentation aid.
  - *e.g.*, supports "programming by contract"
- Debugging aid.
  - Find out where/when assumptions are wrong...
- Basis for an exception mechanism.
  - *e.g.*, integrate with exceptions by allowing assertion failures to be caught dynamically.

Types of Assertions

- Assertions are used for several purposes:
  - *Preconditions*
    - State the requirements under which subprograms are applicable.
  - *Postconditions*
    - Properties guaranteed upon subprogram exit.
  - *Class Invariants*
    - Properties that characterize class instances over their lifetime
      - Note, subprogram preconditions and postconditions are implicitly assumed to include the class invariant.
  - *Loop Invariants*
    - Loop invariants specify properties that are always true during the execution of a loop.

Assertion Example

- **--** Eiffel array

```
class ARRAY[T] export
      lower, upper, size, get, put
feature
      lower, upper, size : INTEGER;

      Create (minb, maxb : INTEGER) is do ... end;

      get (i : INTEGER): T is
            require -- precondition
                  lower <= i; i <= upper;
            do ... end;

      put (i : INTEGER; value : T) is
            require
                  lower <= i; i <= upper;
            do ...
            ensure -- post condition
                  get (i) = value;
            end;

invariant -- class invariant
      size = upper − lower + 1; size >= 0;
end -- class ARRAY
```


Programming by Contract

- Assertions support *Programming by Contract*.
  - This formally specifies the relationship between a class and its clients, expressing each party's rights and obligations.
- *e.g.*,
  - A *precondition* and a *postcondition* associated with a subprogram describe a contract that binds the subprogram.
    - ∗ But *only* if callers observe the precondition...
- The contract guarantees that if the *precondition* is fulfilled the *postcondition* holds upon subprogram return.

Using Assertions to Specify ADTs

- Conceptually, ADTs consist of four parts:

  (1) *types*
  (2) *functions*
  (3) *preconditions/postconditions*
  (4) *axioms*

- However, most languages only allow specification of the first two parts (*i.e.*, *types* and *functions*)
- Assertions provide a mechanism to express the preconditions and axioms corresponding to ADTs.
  - However, few general purpose languages provide support for complete ADT specifications, Eiffel goes further than most in this regard.

Handling Assertion Violations

- If the client's part of the contract is not fulfilled (*i.e.*, if the caller does not satisfy the preconditions) then the class is *not* bound by the postcondition.
- This can be integrated with an exception handling mechanism, *e.g.*,:
  - Exceptions are generated:

    (1) when an assertion is violated at run-time
    (2) when the hardware or operating system signals an abnormal condition.
  - Note, exceptions should not be used as non-local gotos.
    - ∗ They are a mechanism for dealing with abnormal conditions by either:

      (1) *Termination*: cleaning up the environment and repo the caller,
      (2) *Resumption*: attempting to achieve the aim of the

Assertions in C

- Enabled by including the <assert.h> header.
- It incurs no code size increase and no execution speed decrease in the delivered product.
- Typical definition via a macro definition such as:

```
#ifdef NDEBUG
#define assert(ignore) 0
#else
#define assert(ex) \
      ((ex) ? 1 : \
      (__eprintf("Failed assertion " #ex \
      " at line %d of "%s".\n", \
      __LINE__, __FILE__), abort (), 0))
      /* Note use of ANSI-C "stringize" facility.
#endif // NDEBUG
```


Assertions in C (cont'd)

- If the expression supplied to the `assert` macro is false, an error message will be printed and the program will STOP DEAD AT THAT POINT!
- *e.g.*, provide array bounds checking

```
#include <string.h>
/* ...*/
{
      char *callers_buffer;
      char buffer[100];
      /* ...*/
      assert (sizeof buffer > 1 + strlen (callers_buffer));
      /* Program aborts here if assertion fails. */
      strcpy (buffer, callers_buffer);
      /* ...*/
}
```


Assertions in C (cont'd)

- Another interesting application of `assert` is to extend it to perform other duties as well.
  - *e.g.*, code profiling and error logging:

```
#define assert(x) { \
      static int once_only = 0; \
      if (0 == once_only) { \
            once_only = 1; \
            profile_assert ("__LINE__", "__FILE__"); \
      } \
      /* ...*/ \
      /* standard assert test code goes here */ \
}
```

- However, the main problem C `assert` is that it doesn't integrate with any exception handling scheme.
  - *e.g.*, as contrasted to Eiffel.

Assertions in C++

- The overall purpose of the proposed ANSI-C++ assertion implementation is twofold:
  1. To provide a default behavior similar to the C `assert` facility.
  2. To rely on specific C++ facilities (*e.g.*, templates and exceptions) to provide a more generic and powerful support than simple macros.

Assertions in C++ (cont'd)

- What follows is the proposed implementation:

```
// -- file assert.h --
#ifndef __ASSERT_H
#define __ASSERT_H
#ifndef NDEBUG
#include <iostream.h>
extern "C" void abort (void);
// -- 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);
    }
};
```


Assertions in C++ (cont'd)

- Proposed implementation (cont'd)

```
// -- specific C++ macro (needed for preprocessing!)
#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 */
```


Assertions in C++

- The C++ `assert` Macro
  - As with the C 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
        /* Aborts if p == 0 */
        ...
    }
};
```

  - Validity of the expression is checked and a rudimentary message is printed in case of failure.

Assertions in C++ (cont'd)

- The C++ `Assert` Macro
  - The primary goal of the `Assert` macro is to delegate the responsibility for handling the failure to the caller.
    - *e.g.*, print appropriate error messages, make a call to exit instead of abort...
  - A typical example is range checking of a subscript operator, as in:

```
class Checked_Vector : public Vector {
public:
    class Out_Of_Range {
        int l;
    public:
        Out_Of_Range (const char *,
            const char *, int line)
            : l (line) {}
        int line (void) { return l; }
    };
    int& Checked_Vector::operator[] (int index) {
        Assert (index >= 0 && index < size,
            Checked_Vector::Out_Of_Range);
        // ...
};
```

Assertions in C++ (cont'd)

- The Assert Macro (cont'd)
  - *e.g.*,

```
int f (Checked_Vector &v, int index) {
      int elem;
      try {
            elem = v[index];
      }
      catch (Checked_Vector::Out_Of_Range &e) {
            cerr << "Checked_Vector:"
                  << " range checking failed:"
                  << " index="
                  << index
                  << ", size= " << v.size ()
                  << ", line= " << e.line ()
                  << 'n';
                  exit (−1);
      }
      return elem;
}
```


Assertions in C++ (cont'd)

- The Assert Macro (cont'd)
  - Since the exception is thrown before the program failure occurs (*e.g.*, Out_Of_Range), the environment is not corrupted when the run-time flow returns to the caller.
  - If an exception is not caught (as is the case for the Checked_Vector::Out_Of_Range 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.
    * Note, this can alter the conditions under which the failure occurred.