- Automate and randomize the generation of test cases so that millions or billions of tests are executed, of course depending on how long time each test takes.
- Perform all tests on both little and big endian machines.
- Perform all tests using different compilers and libraries.
- Perform all tests using different compiler optimization levels.
- Perform all tests using both 32-bit and 64-bit machines, and different models such as 32/64-bit pointers and integers.

1.8 List of errors

In this section we list common errors made especially by new C programmers. All are not necessarily invalid C code, but probably mean something different than what the programmer had in mind.

1. Write `const int SIZE = 100;` and expect it to be a compile-time constant-expression suitable for a `case`-label or global array size (in C++ it is a real constant, though). Use either of the following instead:

   ```
   #define SIZE    (100)

   enum e { SIZE = 100 };
   ```

   where the object macro is preferable if the constant might exceed the size of an `int` since an enumeration constant has that type, although many compilers including `gcc` support larger enumeration constants.
2. Mix lower and upper case letters in identifiers. Use instead underscore to separate words in a identifier. Write `print_table` instead of `printTable`.
3. Calling a function such as `strlen(s)` each loop iteration even if `s` has not changed. The Java attribute `length` does not exist in C for strings or other forms of arrays.
4. Using `sizeof(char)`. It is defined to always be 1. While not wrong, of course, writing it might be interpreted to indicate that the programmer didn't know this.
5. Forgetting to write a new-line character at the end of the last output the program performs.
6. Assuming the C implementation uses ASCII and that e.g. space has value 32 when it is portable to write `' '`.

7. Risk leaking memory at a `realloc`. Assume you have a table which contains pointers to objects with memory allocated from the heap. If you then want to increase the size of the table using:

   ```
   table = realloc(table, new_size);
   ```

   then, if `realloc` returns `NULL` you will create a memory leak since all pointers in the table are lost. Save a copy of the old value of `table` in case you need it:

   ```
   old_table = table;
   table = realloc(table, new_size);
   if (table == NULL) {
           /* disaster which requires the deallocation of
            * all objects in the table, which can be done
            * using the old value of the pointer.
            */

            /* ... */
   }
   ```

8. Using a variable of type `char` to store a character returned from an I/O function. The type of the return value from the I/O functions in the Standard Library is `int`, which is the correct type to use. The reason is that end of file is indicated by returning the constant `EOF` which usually is -1. If `EOF` is defined so, and the type `char` is unsigned, then:

   ```
   char c;

   c = fgetc(fp);
   if (c == EOF) {
           /* ... */
   }
   ```

   will not work when `fgetc` returns `EOF`, since, assuming 8 bits in a `char`, and the `char` being an unsigned type, when the value -1 is returned from `fgetc` it is converted to an `unsigned char` and `c` is assigned the value 255. Whether `char` is a signed or unsigned type is implementation defined, and with implementations where `char` is a signed type, the code will work, but is thus not portable.

9. Using an uninitialized local variable. This is especially dangerous if it is a pointer.

10. Writing your own versions of what is already in the Standard C library and declared e.g. in `<ctype.h>`.

11. Using an integer instead of a symbolic constant in declarations and statements as in:

    ```
    int     count[100];
    ```

    while instead:

    ```
    int     count[SIZE];
    ```

    is better. Exceptions to this rule include e.g. the 2 in $\pi/2$.

12. Not an error in a strict sense, but it is recommended to write a comment about the purpose of each variable after its declaration.

13. Using nested functions. This is a GNU extension and not ISO C.

14. Defining a variable-length array (VLA) when the array size is known at compile-time:

    ```
    int main(void)
    {
            int size = 100;
            int a[size];

            /* ... */
    }
    ```

    It is then better to `#define SIZE (100)` so that the array will be allocated its memory as a fixed part of the function's call frame. Additionally, and more serious, the code indicates that the programmer thinks the compiler will automatically propagate the constant 100 and therefore must allocate it as a normal array on the stack. For instance, if the program also contained a struct as follows:

    ```
    struct {
            int     b;
            int     c[size];
    }       d;
    ```

    there would be a compilation error, since VLA's are forbidden in structs.

15. The indentation used makes the program too inconvenient to read when printed on paper. A function which exceeds 80 characters per line (with a tabstop of eight) should be divided into smaller pieces.

16. The program is insecure and can suffer from undefined behavior due to an array index out of bound if the input is constructed properly.