**EXP20-C. Perform explicit tests to determine success, true and false, and equality**

Perform explicit tests to determine success, true/false, and equality to improve the readability and maintainability of code and for compatibility with common conventions.

In particular, do not default the test for nonzero. For instance, suppose a `foo()` function returns 0 to indicate failure or a nonzero value to indicate success. Testing for inequality with 0,

```c
if (foo() != 0) ...
```

is preferable to

```c
if (foo()) ...
```

despite the convention that 0 indicates failure. Explicitly testing for inequality with 0 benefits maintainability if `foo()` is later modified to return 1 rather than 0 on failure.

This recommendation is derived from and considers the implications of the following common conventions:

1. Functions return 0 if false and nonzero if true [StackOvflw 2009].
2. Function failures can typically be indicated by 1 or any nonzero number.
3. Comparison functions (such as the standard library function `strcmp()`, which has a trinary return value) return 0 if the arguments are equal and nonzero otherwise (see `strcmp function`).

**Noncompliant Code Example**

In this noncompliant code example, `is_banned()` returns 0 if false and nonzero if true:

```c
LinkedList bannedUsers;

int is_banned(User usr) {
    int x = 0;
    Node cur_node = (bannedUsers->head);
    while (cur_node != NULL) {
        if(!strcmp((char *)cur_node->data, usr->name)) {
            x++;
        }
        cur_node = cur_node->next;
    }
    return x;
}

void processRequest(User usr) {
    if(is_banned(usr) == 1) {
        return;
    }
    serveResults();
}
```

If a banned user is listed twice, the user is granted access. Although `is_banned()` follows the common convention of returning nonzero for true, `processRequest` checks for equality only with 1.

**Compliant Solution**

Because most functions guarantee a return value of nonzero only for true, the preceding code is better written by checking for inequality with 0 (false), as follows:
LinkedList bannedUsers;

int is_banned(User usr) {
    int x = 0;
    Node cur_node = (bannedUsers->head);
    while(cur_node != NULL) {
        if (strcmp((char *)cur_node->data, usr->name)==0) {
            x++;
        }
        cur_node = cur_node->next;
    }
    return x;
}

void processRequest(User usr) {
    if (is_banned(usr) != 0) {
        return;
    }
    serveResults();
}

Noncompliant Code Example

In noncompliant code, function status can typically be indicated by returning 1 on failure or any nonnegative number on success. This is a common convention in the standard C library, but it is discouraged in ERR02-C. Avoid in-band error indicators.

Although failures are frequently indicated by a return value of 0, some common conventions may conflict in the future with code in which the test for nonzero is not explicit. In this case, defaulting the test for nonzero welcomes bugs if and when a developer modifies validateUser() to return an error code or 1 rather than 0 to indicate a failure (all of which are also common conventions).

int validateUser(User usr) {
    if(listContains(validUsers, usr)) {
        return 1;
    }
    return 0;
}

void processRequest(User usr, Request request) {
    if(!validateUser(usr)) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}

Although the code will work as intended, it is possible that a future modification will result in the following:
errno_t validateUser(User usr) {
    if(list_contains(allUsers, usr) == 0) {
        return 303; /* User not found error code */
    }
    if(list_contains(validUsers, usr) == 0) {
        return 304; /* Invalid user error code */
    }
    return 0;
}

void processRequest(User usr, Request request) {
    if(!validateUser(usr)) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}

In this code, the programmer intended to add error code functionality to indicate the cause of a validation failure. The new code, however, validates any invalid or nonexisting user. Because there is no explicit test in processRequest(), the logical error is not obvious and seems correct by certain conventions.

Compliant Solution

This compliant code is preferable for improved maintenance. By defining what constitutes a failure and explicitly testing for it, the behavior is clearly implied, and future modifications are more likely to preserve it. If a future modification is made, such as in the previous example, it is immediately obvious that the if statement in processRequest() does not correctly utilize the specification of validateUser().

int validateUser(User usr) {
    if(list_contains(validUsers, usr)) {
        return 1;
    }
    return 0;
}

void processRequest(User usr, Request request) {
    if(validateUser(usr) == 0) {
        return "invalid user";
    }
    else {
        serveResults();
    }
}

Noncompliant Code Example

Comparison functions (such as the standard library strcmp() function) return 0 if the arguments are equal and nonzero otherwise.

Because many comparison functions return 0 for equality and nonzero for inequality, they can cause confusion when used to test for equality. If someone were to switch the following strcmp() call with a function testing for equality, but the programmer did not follow the same convention as strcmp(), the programmer might instinctively just replace the function name. Also, when quickly reviewed, the code could easily appear to test for inequality.

void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (!strcmp((user->password), pw)) {
        grantAccess();
    } else {
        denyAccess("Incorrect Password");
    }
}
The preceding code works correctly. However, to simplify the login code or to facilitate checking a user's password more than once, a programmer can separate the password-checking code from the login function in the following way:

```c
int check_password(User *user, char *pw_given) {
    if (!strcmp((user->password),pw_given)) {
        return 1;
    }
    return 0;
}

void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (!check_password(user, pw)) {
        grantAccess();
    } else {
        denyAccess("Incorrect Password");
    }
}
```

In an attempt to leave the previous logic intact, the developer just replaces `strcmp()` with a call to the new function. However, doing so produces incorrect behavior. In this case, any user who inputs an incorrect password is granted access. Again, two conventions conflict and produce code that is easily corrupted when modified. To make code maintainable and to avoid these conflicts, such a result should never be defaulted.

Compliant Solution

This compliant solution, using a comparison function for this purpose, is the preferred approach. By performing an explicit test, any programmer who wishes to modify the equality test can clearly see the implied behavior and convention that is being followed.

```c
void login(char *usr, char *pw) {
    User user = find_user(usr);
    if (strcmp((user->password),pw) == 0) {
        grantAccess();
    } else {
        denyAccess("Incorrect Password");
    }
}
```

Risk Assessment

Code that does not conform to the common practices presented is difficult to maintain. Bugs can easily arise when modifying helper functions that evaluate true/false or success/failure. Bugs can also easily arise when modifying code that tests for equality using a comparison function that obeys the same conventions as standard library functions such as `strcmp`.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP20-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Low</td>
<td>P12</td>
<td>L1</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-EXP20</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>114 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-EXP20-a</td>
<td>Avoid comparing values with TRUE macro/enum constant using equality operators (&quot;==&quot;, &quot;!&quot;)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>3344, 4116</td>
<td></td>
</tr>
</tbody>
</table>

Bibliography
Should I Return TRUE/FALSE Values from a C Function?