Process Summary Static Code Analysis

For this assignment I systematically analyzed the Cppcheck static code analysis against the Visual Studio analysis in an attempt at pinpointing the differences between the two, determination of risks, description of issues, and a description on how to mitigate the varying issues.

Beginning with the warning Visual Studio displayed on line 42 pertaining to an uninitialized variable, this error was also mirrored on the Cppcheck in the XML file but with more extensive information. Mitigation of this issue would be to modify the copy constructor to copy ‘x’ from the source object to the new object. Risks could range from unexpected behavior such as bugs in the code, data corruption if the value of x is changed it will not be reflected on other objects that were copied from original object resulting in data corruption or inconsistent values. Potential for memory leaks exist if dynamically allocated resources are stored in ‘x’ they will not be deallocated properly when the object is destroyed.

Line 66 is a buffer overrun error which is reasonably correctable due to it accessing an invalid index. It is corrected by ensuring ‘count’ is within bounds of array ‘buf’ before accessing such as placing count >= 0 and count < 10 instead of == 1000. This can also result in memory corruption or program crashes. Comparatively, the XML output displays that the code on line 6is assigned a value that is never used which is generated by the static tool and apparently has to do with redundancy and is safe to remove due to it not being used elsewhere in the code.

Moving down the error list in Visual Studio, line 64 is minimal as a local variable is not initialized which is the same method as the buffer overrun issue. This can be corrected if ‘buf’ is declared inside the ‘if’ statement, so the scope is limited to that specific block. The benefit of this is it reduces memory used and is more readable.

The next warning is on line 52 where the function assumed not to throw an exception but does. It is marked as ‘noexcept’ resulting in undefined behavior or errors. There is resolution in removing ‘noexcept’ or catch the exception within the method. The risk is in its unexpected behavior or potential crash because the caller is not prepared to handle the exception. The XML output declares the same error displaying the specific error message indicating the function declared ‘noexcept’ has thrown an exception. The additional warning message for line 52 indicated a violation of the promise and how an exception is thrown regardless. However, this message does not appear to be directly reflected in the XML output as the error already addresses the issue.

The final issue displayed on the error list of Visual Studio is for an unsafe operation on line 132 with a ‘try’ statement. The issue with this is it may produce unexpected results as there is a comparison between a Boolean type which is true or false and an integer type which cannot equal a constant. The compiler generates the warning message due to the ‘==’ operation. There is risk in this mainly due to ‘assert’ expects a Boolean argument and non-zero integer is considered true while zero is false. If ‘my\_function () ==3’ is false, then the assert will fail and throw an assert but if it is true then the assert will do nothing. When I reference the XML output for further information, this warning is not referenced.

While there are many other messages in the XML output, they seem to all correlate loosely with the warnings and messages of Visual Studio. Lines 127 and 129 in the XML display correspond to the same block of code for warning about the unsafe ‘==’ operation. Line 109 error suggests that the dereference pointer might have been left out allowing the assignment to modify the object being pointed to. The error on line 59 in the XML output has to do with the function ‘foo’ setting a pointer and the local variable ‘b’ being destroyed when the function returns. This means the pointer will be pointing to a random memory location leading to undefined behavior and could lead to major unexpected issues. The error for line 98 in the XML output is referring to the function declaring the Boolean type which has a non-Boolean value. This can lead to unintended errors as well. The errors on line 65 and 66 correlate directly to the warnings in Visual Studio about the bufferoverrun. The same with errors on lines 42 and 50 in the XML output that have been addressed previously. The XML error for line 103 suggests that ‘Token::next’ can be made a static function potentially improving performance. Apparently, the function may be more appropriately moved to an unnamed space. The XML error on line 129 corresponds to the comparison error in Visual Studio which has already been addressed previously. Moving down the XML output to the next unaddressed error, lines 117 and 133 have shadow variable errors which is basically a variable name shadows an outer variable which could lead to unexpected behavior. It is easily resolved by changing the name or the inner variable or to explicitly reference the outer variable using its full scope. Lines 135 and 119 of the XML file both have shadow variable errors as well which are remedied in the same manner. The next issue defined in the XML output is from line 82 to 89 with an invalid container error. This can lead to undefined behavior and crashes as well because an iterator is being used to access elements that may have been invalidated due to the ‘erase’ call. Ideally the elements should be stored in a separate data structure or use a safe iterator to modify the container. The remaining errors on the XML output have been covered previously but are displaying unreadVariable errors. The variable is being assigned a value, but it is never used in the rest of the code. They could indicate potential issues such as a logic problem. Therefore, it is important to remove unused or redundant variables and keep the code clear and concise.

Text

Description automatically generated

