Issues Detected by Both Tools

1) Uninitialized Member Variable (A::x)

* Risk Level: High
* Description: The member variable A::x is not initialized in the copy constructor. Uninitialized variables can lead to undefined behavior and potential security vulnerabilities.
* Location: Line 42
* Mitigation: Initialize A::x in the copy constructor to ensure it holds a valid value.

A(const A& other) : x(other.x) { /\* ... \*/ }

2) Buffer Overrun Writing to buf

* Risk Level: High
* Description: There is a potential buffer overrun when writing to buf[count] if count equals 1000. The array buf has a size of 10, so accessing index 1000 is out of bounds.
* Location: Line 66
* Mitigation: Ensure that count is within the bounds of the array buf. Adjust the condition or resize the array as needed.

if (count == 10) {

buf[count] = /\* ... \*/;

}

3) Exception Thrown in noexcept Function

* Risk Level: High
* Description: The function MySpecialType::DontThrow is declared with noexcept, but it throws an exception. This can lead to a call to std::terminate, abruptly ending the program.
* Location: Line 52
* Mitigation: Either remove the noexcept specifier from the function or ensure that no exceptions are thrown within the function.

void DontThrow() noexcept {

// Remove code that may throw exceptions

}

4) Comparison of Boolean with Integer

* Risk Level: Medium
* Description: A boolean expression is compared with an integer other than 0 or 1, which may lead to unexpected behavior.
* Location: Line 129
* Mitigation: Modify the comparison to ensure that a boolean is compared with true/false or cast the integer appropriately.

if (my\_function() == 2 && 3)

Issues Detected Only by Cppcheck

5) Assignment in assert Statement Modifies Variable

* Risk Level: Medium
* Description: The assert statement modifies the variable z. Since assert statements are removed in release builds, this modification would not occur, potentially leading to inconsistent behavior between debug and release versions.
* Location: Line 127
* Mitigation: Avoid modifying variables within assert statements. Move the assignment outside the assert.

z = /\* new value \*/;

assert(z == expected\_value);

6) Function with Side Effects Called in assert Statement

* Risk Level: Medium
* Description: The function my\_function, which may have side effects, is called within an assert statement. As assert statements are omitted in release builds, the side effects will not occur, leading to inconsistent behavior.
* Location: Line 129
* Mitigation: Call the function outside the assert statement if its side effects are needed.

int result = my\_function();

assert(result == expected\_value);

7) Assignment of Function Parameter Has No Effect

* Risk Level: Low
* Description: A function parameter is assigned a value within the function, but this has no effect outside the function. This might indicate a logic error or missing pointer dereference.
* Location: Line 109
* Mitigation: If the intention is to modify the argument's value outside the function, pass it by reference or pointer and dereference appropriately.

void some\_function(int\* param) {

\*param = new\_value;

}

8) Address of Local Variable Assigned to Function Parameter

* Risk Level: High
* Description: The address of a local variable is assigned to a function parameter, potentially leading to dangling pointers if used outside the function's scope.
* Location: Line 59
* Mitigation: Ensure that pointers passed to functions are valid after the function returns. Avoid returning addresses of local variables.

void function(int\*& param) {

int\* local = new int;

param = local;

}

9) Non-Boolean Value Returned from Function Returning bool

* Risk Level: Low
* Description: A function declared to return bool is returning a non-boolean value, which may cause implicit conversions and unexpected results.
* Location: Line 98
* Mitigation: Ensure that the function returns a boolean value (true or false).

bool is\_valid() {

return (value == expected\_value);

}

10) Possible Null Pointer Dereference (tok)

* Risk Level: High
* Description: There's a possible null pointer dereference of tok. The condition checking if tok is null may be redundant or insufficient.
* Location: Line 109
* Mitigation: Validate that tok is not null before dereferencing, and ensure that any null checks are meaningful.

if (tok != nullptr) { }

11) Variable Scope Can Be Reduced (buf)

* Risk Level: Low
* Description: The scope of the variable buf can be reduced to limit its visibility and lifetime, improving code maintainability.
* Location: Line 64
* Mitigation: Move the declaration of buf to the innermost scope where it is used.

for (...) {

int buf[10];

}

12) Shadowed Variables (x, y, z)

* Risk Level: Low
* Description: Local variables x, y, and z are shadowing outer variables of the same names. This can cause confusion and potential errors.
* Location: Lines 133-135
* Mitigation: Rename inner variables or avoid redeclaring variables with the same names in nested scopes.

int x\_outer = /\* ... \*/;

{

int x\_inner = /\* ... \*/;

}

13) Using Iterator to Potentially Invalid Container (items)

* Risk Level: High
* Description: An iterator to a local container items may become invalid, especially after modifications like erase, leading to undefined behavior.
* Location: Line 87
* Mitigation: Ensure that iterators are valid after container modifications or avoid using iterators that may be invalidated.

for (auto it = items.begin(); it != items.end(); /\* no increment \*/) {

if (condition) {

it = items.erase(it);

} else {

++it;

}

}

14) Unused Struct Member (A::x)

* Risk Level: Low
* Description: The class member A::x is never used, indicating possible dead code or an incomplete implementation.
* Location: Line 41
* Mitigation: Remove the unused member if it's not needed, or implement its usage if it was intended to be used.

15) Unread Variables (buf[count], tok, x, y)

* Risk Level: Low
* Description: Variables are assigned values that are never read, indicating redundant code or possible logical errors.
* Locations:
  + buf[count] at Line 66
  + tok at Line 109
  + x at Line 117
  + y at Line 118
* Mitigation: Remove the assignments if they are unnecessary, or utilize the variables if they are meant to be used.