Checked C Internship Projects - Summer 2020

# Code conversion – Convert the string subdirectory in the musl C library to Checked C

**Background:** musl is an implementation of the C standard library for Linux-based systems. The goal of Checked C is to improve systems programming in C by making memory accesses secure. To take advantage of the security guarantees provided by Checked C an existing C codebase needs to be re-written in Checked C and compiled using the Checked C compiler.

**Project:** We would like musl to take advantage of the security features provided by Checked C. To achieve this, we would have to convert the musl codebase to Checked C. Converting a C codebase to Checked C would mainly involve the following three main conversions:

1. Convert singleton pointers to checked pointers.
2. Convert array pointers to checked array pointers.
3. Convert null terminated arrays to checked nt array pointers.

Due to the size of the musl codebase, the goal of the internship is to convert all .c files in the **string** subdirectory of musl to Checked C. There are a total of 1492 .c files in the string subdirectory.

An important aspect of the conversion would be to maintain compatibility with gcc. To ensure that the converted code can compile with gcc we would need to guard the conversion by #ifdef’s.

To check the correctness of conversions the intern would also be expected to run the musl unit tests. The intern may also benchmark musl to determine the overall effect of Checked C on code size, compile time, memory consumption, etc.

**Useful links:**

* [https://www.musl-libc.org](https://www.musl-libc.org/)
* <https://github.com/microsoft/checkedc/wiki/Legacy-Conversion-Tips>
* <https://github.com/microsoft/checkedc/wiki/Conversion-operations>

# Compiler implementation: Bounds widening for multiple null-terminated pointer dereferences

**Background:** A null-terminated array is a sequence of elements in memory that ends with a null terminator. These arrays can be divided into two parts: a prefix with bounds and a sequence of additional elements that ends with a null terminator. An important property of a null-terminated array is that the initial elements of the sequence can be read, provided that preceding elements are not the null terminator. This gives rise to the following observation:

“The bounds of a null-terminated array can be widened based on the number of elements read.”

**Project:** Checked C can automatically widen bounds of a null-terminated pointer if it is dereferenced at its upper bound. For example:

\_Nt\_array\_ptr<T> p : count(0) = “”;

if (\*p) // widen bounds by 1.

Now, we would like to support bounds widening when the same null-terminated pointer is dereferenced multiple times in the same condition. For example:

if (\*p && \*(p + 1) && \*(p + 2)) // widen bounds by 3.

Bounds widening is implemented using a dataflow analysis over the CFG. To support multiple null-terminated pointer dereferences the goal of the internship is to implement another local dataflow analysis per conditional statement. The intern would be expected to write unit tests to check the correctness of the approach. The intern may also benchmark the compiler to determine the overall effect on run time, code size, compile time, memory consumption, etc.

**Useful links:**

* <https://github.com/microsoft/checkedc-clang/blob/master/clang/docs/checkedc/Bounds-Widening-for-Null-Terminated-Arrays.md>
* <https://en.wikipedia.org/wiki/Data-flow_analysis>
* <https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf>

# Convert the network subdirectory in the musl C library to Checked C

**Background:** musl is an implementation of the C standard library for Linux-based systems. The goal of Checked C is to improve systems programming in C by making memory accesses secure. To take advantage of the security guarantees provided by Checked C an existing C codebase needs to be re-written in Checked C and compiled using the Checked C compiler.

**Project:** To take advantage of the memory safety features of Checked C, a converted .c file from the musl codebase should make the following conversions:

1. Convert singleton pointers that are not involved in pointer arithmetic to checked singleton pointers.
2. Convert array pointers to checked array pointers.
3. Convert null terminated arrays to checked nt array pointers.

Due to the size of the musl codebase, the goal of this project is to convert all .c files in the **network** subdirectory of musl to Checked C. There are 77 .c files with approximately 3450 lines of code in the **network** subdirectory.

An important aspect of the conversion would be to maintain compatibility with gcc. To ensure that the converted code can compile with gcc we would need to guard the conversion by #ifdef’s.

To check the correctness of conversions the intern would also be expected to run the musl unit tests. The intern may also benchmark musl to determine the overall effect of Checked C on code size, compile time, memory consumption, etc.

**Useful links:**

* [https://www.musl-libc.org](https://www.musl-libc.org/)
* <https://github.com/microsoft/checkedc/wiki/Legacy-Conversion-Tips>
* <https://github.com/microsoft/checkedc/wiki/Conversion-operations>

# Compiler implementation: Improving error messages

**Background:** As the framework for checking declared bounds matures, it can detect bounds errors that result from a wider variety of causes. Some of these causes can be subtle and difficult to debug without error messages that a) provide the user with sufficient context and b) explain the reasons for the error.

**Project:** This project will focus on two ways to improve the error or warning messages that are emitted when the compiler cannot prove that the inferred bounds of a variable imply the variable’s declared bounds.

1. For an expression embedded within a top-level statement that causes a bounds checking error, the error message should highlight the embedded expression rather than the top-level statement. For example: in the statement len++, other = 0; where len++ causes a bounds checking error, the error message should highlight len++ rather than len++, other = 0;.
2. When a bounds error is caused by a variable v used in inferred bounds where no information is known about v, the error message should state that no information is known about v.

**Details:** This work can be broken into two parts. In the first part, the objective is to target messages to a single assignment. In the second part, the objective is to provide reasons for bounds-related proof failures.

*Target messages to a single assignment:* As part of the work on bounds soundness for variables, bounds checking is done after each top-level CFG statement, rather than at each individual assignment. A statement can contain multiple assignments, any one of which may result in a bounds error. Any errors that occur during bounds checking highlight the entire top-level statement, rather than the specific assignment that caused the error. To aid in debugging, each error message should highlight only the assignment that caused the error. For example:

void multiple\_assignments(array\_ptr<int> p : count(len), int len, int i) {

len = 0, i = 0;

}

The single assignment len = 0 causes the inferred bounds of p to be unknown. The error message should highlight only the assignment len = 0 rather than the top-level statement len = 0, i = 0;. In addition, the error message should say “declared bounds for ‘p’ are invalid after assignment”. Currently, the error message would say “declared bounds for ‘p’ are invalid after statement”.

*Provide reasons for bounds-related proof failures:* There are several reasons why the bounds checker is unable to prove that the inferred bounds of an expression imply the declared bounds. One of these reasons is that the declared bounds use a variable x, the inferred bounds use a variable y, and there is no known information on the relationship between x and y. One way this can occur is when a variable x, used in the declared bounds of a variable p, is modified, where the original value of x is a variable y. For example:

void original\_value(array\_ptr<int> p : count(x), int y) {

// This assignment allows x to have the original value of y in the next assignment.

y = x;

// There is no known relationship between y (the original value of x, used in

// the inferred bounds (p, p + y)) and x (used in the declared bounds (p, p + x).

x = 0;

}

The error message should inform the user that the inferred bounds of p use the variable y, and that no information is known about y.