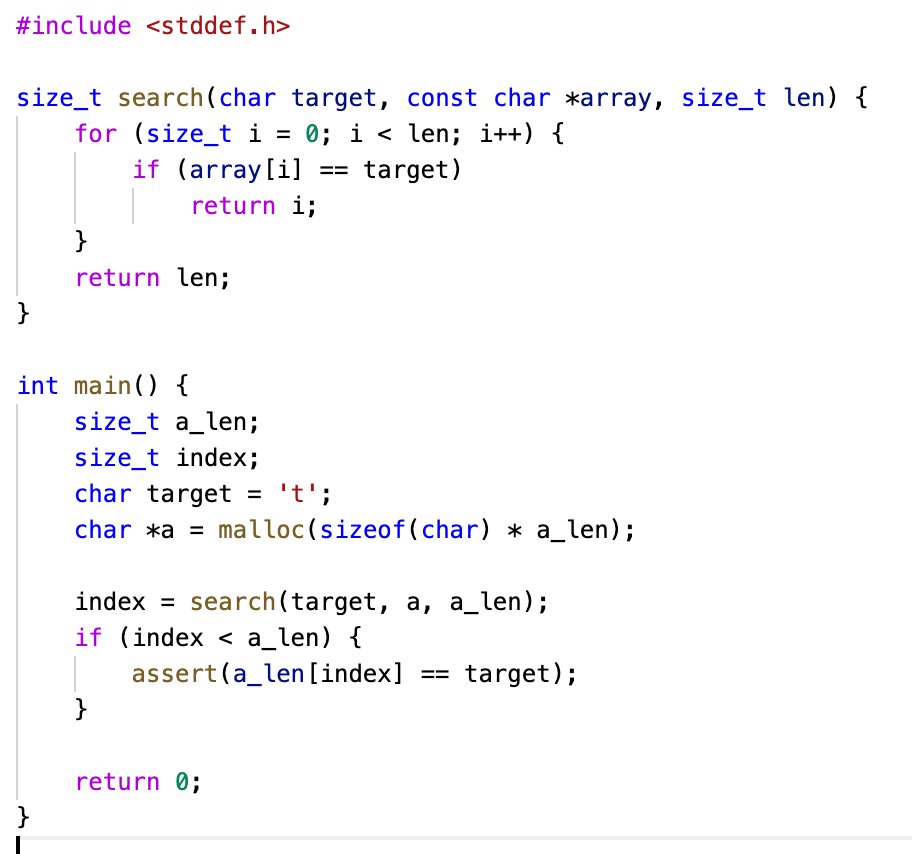
# Name-based abstraction vs address-based abstraction

This document discusses the differences between name-based abstraction (implemented in 2020) and address-based abstraction (to be implemented in 2021) and the reasons we want to move to address-based abstraction.

In the discussion, we will abstract the following example in two ways:



This program iterate through an array to find the index where a target value appears for the first time. If the target value is not found, the search function will return the length of the array.

Consider that we want to abstract the array main:a. The shape of the abstraction is “c\*c\*c\*” which has 3 concrete locations: 0 must be a concrete location and there are another 2 different concrete locations. Note how the shape looks is not important in this document.

## Name-based abstraction (2020)

The name-based abstraction tries to maintain every variable in the abstracted space if it is possible. For example, if the shape is “c\*c\*c\*”, all variables that can be considered “indices” would be kept abstracted with values from 0 to 5. Our tool applies static analysis to the program to to determine which variables should be abstracted. Going from initial abstracted variables we specify in the json file, it gradually enlarge the set of abstracted variables based on some “rules”:

* If a variable is used to access an abstracted array (i.e. array[var]), the variable should be an abstracted index with the same shape as the array.
* If we assign an abstracted index to another variable, (i.e. var = var\_abst), that another variables should be an abstracted index with the same shape.
* If we compare two variables and one of them is an abstracted index (e.g. var1 < var2), the other should be abstracted with the same shape as well.
* If a function is called, parameter variable inside the function should be abstracted in the same way as its argument. For example, if we define a function “func(size\_t a)” and call it with “func(b)” where b is an abstracted index, then a should also be an abstracted index
* .... a lot of more rules

We applied a number of such “rules” to cover situation we met in real programs so that we were able to prove 14 programs in AWS-C-Common. If there are calculation/assignments that is not covered by our rules, we terminate the abstraction process and tell users something is not supported by our tool.

Let’s go back to the example program. Say in the json specification file, “main:a” is an abstracted array and “main:a\_len” is an abstracted index (length). Our initial set of abstracted entities would consist of these two. Then based on static analysis using the “rules”:

* main:index should be an abstracted index because it is used to access “main:a” (a[index])
* search:array should be an abstracted array because we assign “main:a” when calling the function
* search:len should be an abstracted index because we assign “main:a\_len” when calling the function
* search:i should be an abstracted index because we compare i vs search:len.
* ...

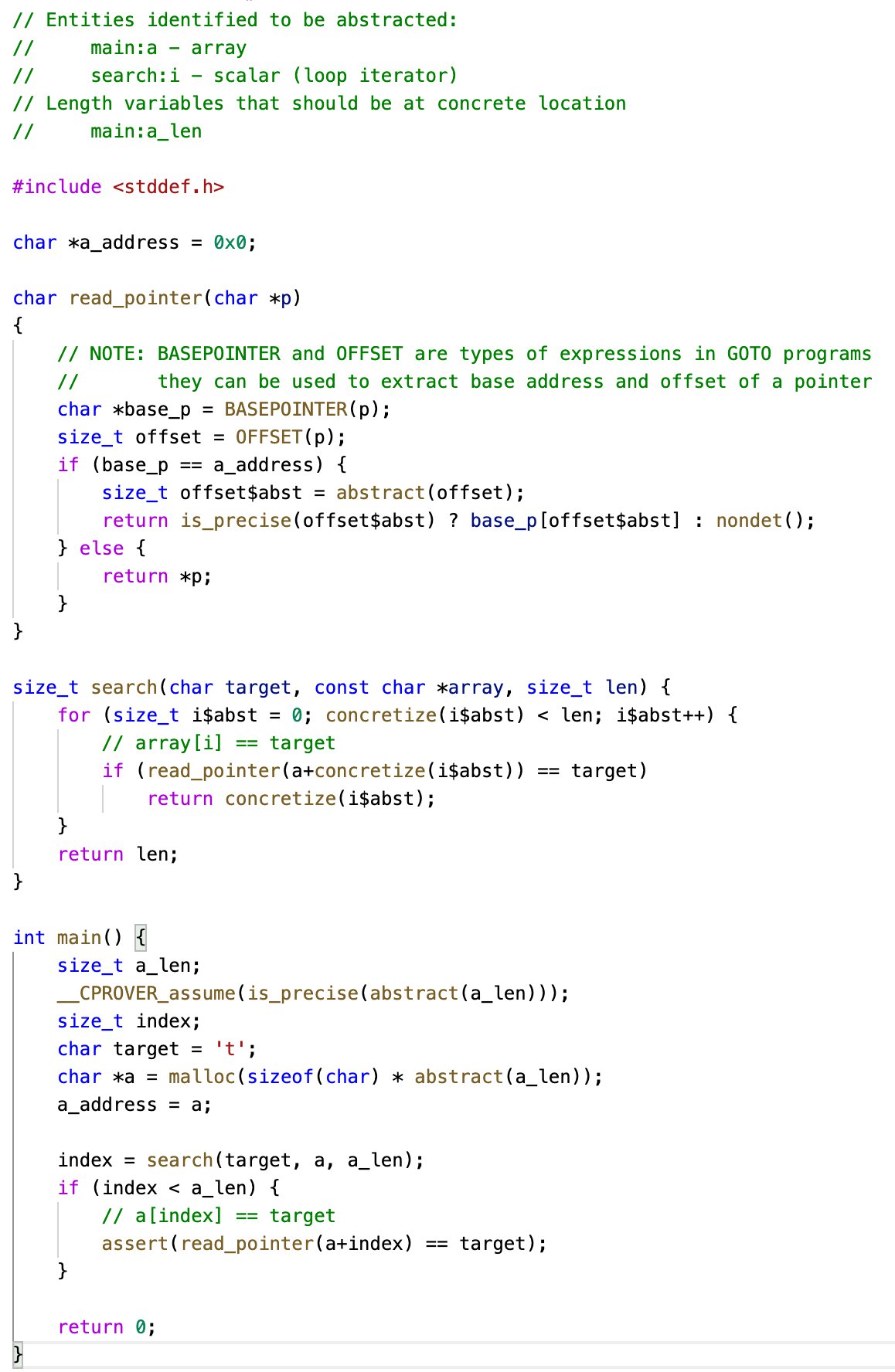
The above entities we found are marked “abstract”, and will be maintained in the abstract world during runtime. For example, search:i’s value could be 0-5 for shape “c\*c\*c\*”. We mark those variables with suffix $abst to flag that they are abstract. After abstraction, the program will look like this (not exactly following C style):



## Address-based abstraction (2021)

Comparing to the previous solution, the address-based abstraction method only abstracts variables when it’s absolutely needed (1. array accesses and 2. loop iterations). For array accesses, it maintains memory addresses where abstract arrays are located and only applies abstraction when there are reads/writes to these addresses. For loop iterations, it marks the index variable used in the loop and lets it run in the abstract space. Values of variables in other scenarios are kept concrete.

After we apply address-based abstraction, the example program would look like this:

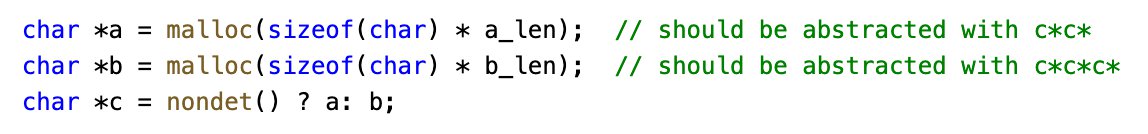


Note that we only need to replace all pointer accesses with our logic for abstraction and concretize loop iterators if needed. Other places of the program is not touched.

## Why do we want to move to address-based approaches?

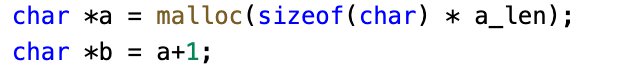
### It is more general.

We can see that the name-based approach determines whether an index should be abstracted and the shape to use before runtime. However, it might not be possible in a lot of cases because sometimes it is only determined at runtime. For example:



The shape of c for abstraction could only be determined at runtime. Such things happens a lot in AWS-C-Common. Normally such expressions are used to initialize things that can fail (“nondet() ? Null: malloc()”).

Another type of behavior the name-based abstraction does not support is complex pointer manipulation. Say we have the following program:



When a is an abstracted array, b should also be an abstracted array. However, they shouldn’t be using the same shape because b has an offset over a. When we access entries using pointer b, the name-based approach would completely mess up at this point. We raised errors in such scenarios. The memory-based approaches can reason about base pointers and offsets when we are accessing an address and solve this issue.

There are a lot of other possible scenarios that can happen in real programs, and memory-based approaches could support them better potentially because it is lower-level.

### It is more simple.

With the address-based approach, we only need to modify low-level read/writes to memory addresses and loop iterators. Other parts of the program are kept intact. With simpler modification, we are more confident that our implementation is inline with our proof.