**Quinsigamond Community College**

Visual Programming with Blockly for Beginners of C++

**Chris Haidas**

**Jonathan Andrew White**

**Christopher Anthony Lindberg**

**CSC212**

**Professor Hao Loi**

**April 4, 2020**

[**Abstract**](#_heading=h.zfq7e6fiwfjx) **3**

[**Introduction**](#_heading=h.1t3h5sf) **4**

[**Research**](#_heading=h.4d34og8) **5**

[**Implementation**](#_heading=h.2s8eyo1) **7**

[**Conclusion**](#_heading=h.b9gq87sgwzhv) **13**

[**References**](#_heading=h.3rdcrjn) **14**

# 

# 

# 

# 

# 

# Abstract

The demand for programmers is growing as our world requires better software, leading to shortages in programmers in all industries. Unfortunately, the abstract nature of many programming languages can be difficult to grasp. This can present a barrier of entry for those interested in programming. Therefore, we plan to implement an application of visual-based programming with graphical representations of the C++ language in order to provide aid and structure to beginner programmers. Visual programming languages, such as Scratch and Alice, use visual representations of code to create games and stories with an emphasis on accessibility and creativity[[1]](#footnote-0). Scratch and Alice however, deal predominantly with game and story creation[[2]](#footnote-1). Our application will utilize graphical representations and type checking capabilities to aid new programmers, as well as those struggling to understand abstract programming structures. We hope our application can help users understand programming and help give them the confidence they need to pursue a career in the field of computer science. Our application will be built upon Blockly, a visual programming application developed by Google developer Neil Fraser and the Blockly team[[3]](#footnote-2). It will be designed for beginner users of C++.

# **Introduction**

The idea of a tool to simply and visualize C++ code was predicated on the developers taking C++ classes in college. C++, being the first language for many in the program, is a difficult language for many student programmers to understand. From this, the desire to aid new students began.

The existence of other software representing some concepts were not unrecognized. Other programs such as Scratch, exists as an introduction to many concepts and ideas in programming, and more specifically: game design. However, while a user of Scratch may understand intuitively the idea and end of a design (or implementation of code, such as a loop), it does not necessarily result in sufficient understanding of how to properly implement code to get to those ends. Thus, a problem unfolds: on one side, an introduction to programming and design for beginners, on the other end of the spectrum exists fully fledged software development that requires knowledge of said intricacies of code. Particularly, a visualization and aid used to help beginner programmers bridge the gap between the start and the end of this spectrum. Blockly C++ is a product of this problem.

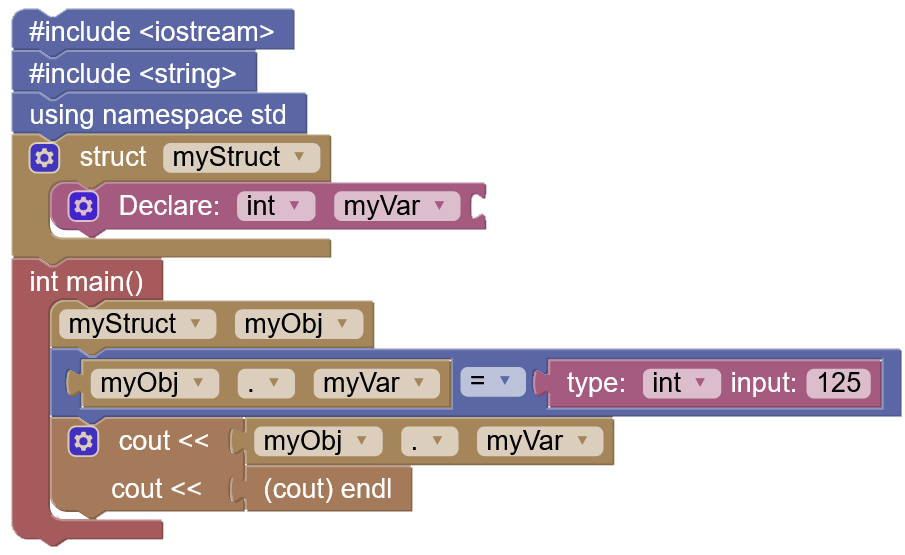
The categorization of blocks into chapters and subchapters may, at first glance, seem out of place. However, the implementation and structure of Blockly C++ is predicated primarily on the student’s required textbook. Further, the various forms of checking present are predicated on a C++ compiler, not for the purpose of replacing its use, but rather to recognize common errors with a visual representation of code structures.

# **Research**

The goal of Blockly C++ is the correct generation of C++ code, supplemented by built-in checking. Because C++ is an intricate language, we needed an API that was large, flexible and easy to work with. Blockly immediately stuck out due to its freely available library, and other projects built upon Blockly such as roboBlockly[[4]](#footnote-3); the latter of which is described as a “C/C++ interpreter”.

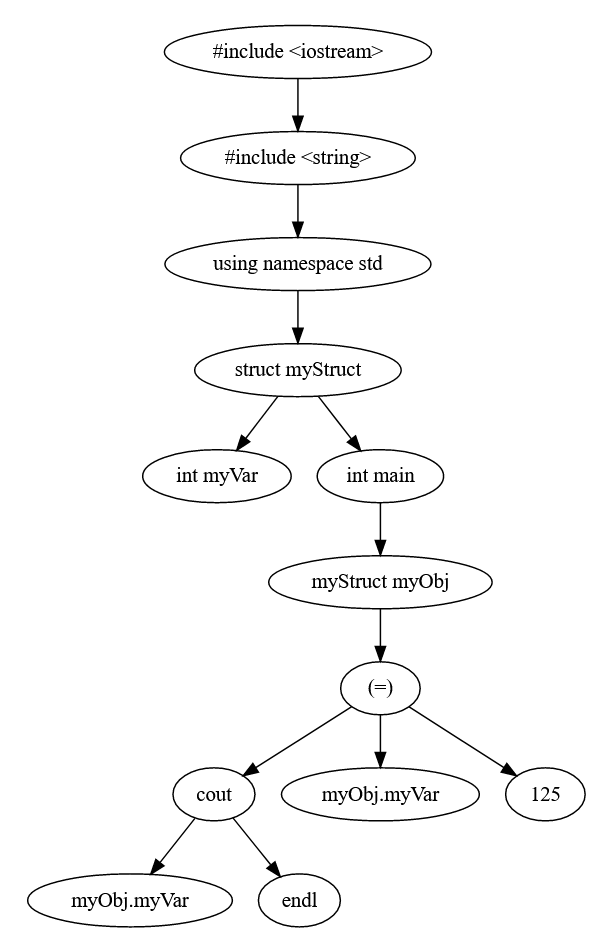
To the left of the workspace is the toolbox. The toolbox exists as the source of all blocks the user is capable of placing. As users place blocks on the workspace, connections formed are similar to nodes connecting in a tree with the blocks being the nodes. Like nodes, each block has certain information that we may need to access. Further, every block may have any number of other blocks attached to them.

Take the following workspace:



Example 1

These connections of blocks can be visually represented as such:



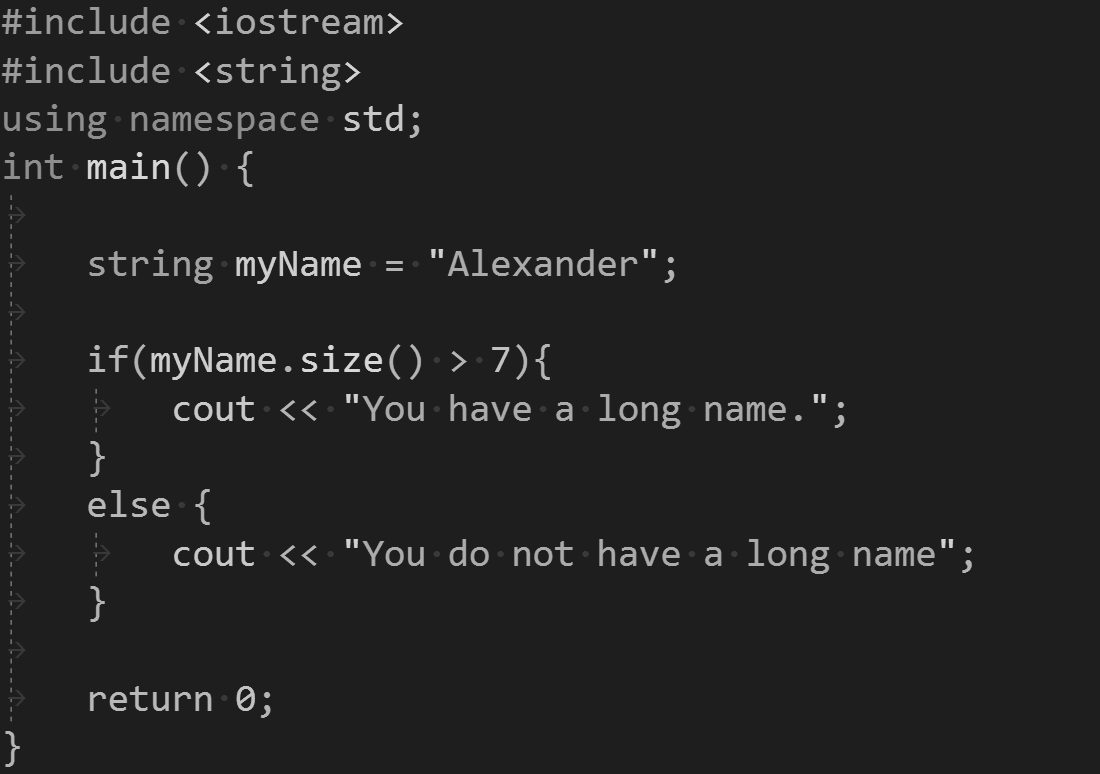
Example 2

Every block in this diagram has information that may be required at any given time. For example, the user may use the *cout* block without the inclusion of *#include <iostream>*. Concurrently, the user may attempt to declare the same variable twice within the same scope. Fortunately, the combination of a prolific and well documented Blockly Library with a flexible data structure allows for easy checking and correction.

# **Implementation**

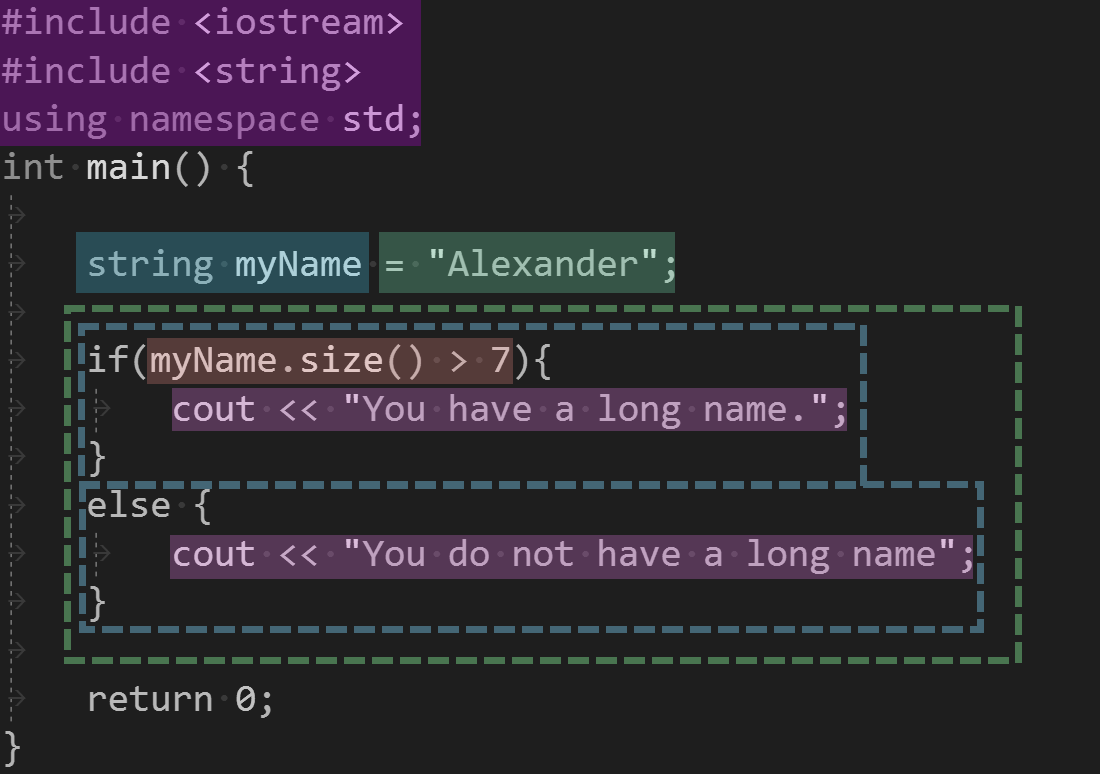
Due to the code generation, logic checking, and type checking all being predicated on visual blocks, general C++ code can also be seen as an aggregation of “blocks”, so to speak.

Take the following C++ code:



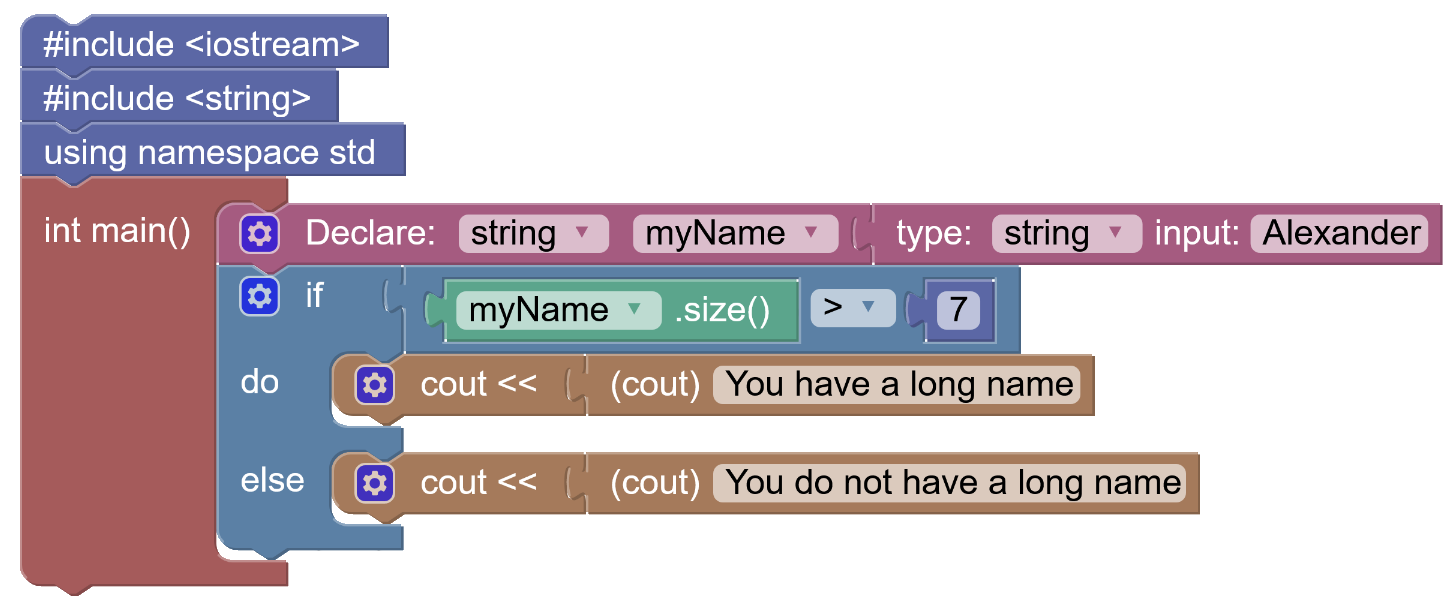
Example 3

Each line of code here has a purpose, and within those lines of code are “blocks” of logic that interact with each other. They can be visually represented as such:



Example 4

Each block of color here represents a different “section” of code that a programmer must keep track of. Each block here can manifest as at least one block in Blockly C++:

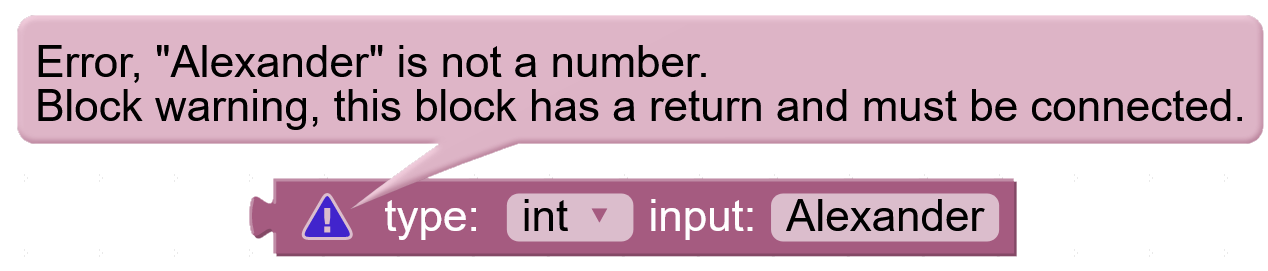


Example 5

These blocks are similar to the previous example of the coded C++. These blocks interact with each other in certain ways, e.g. you cannot declare a variable inside of an “if” condition. Thus, programming with Blockly C++ is intuitively similar to programming C++.

Blockly C++ has three *forms* of type and logic checking. The first form is called a *block warning*. Block warnings do not have to do with code generation, but rather with how the blocks interact with each other. The second form is called a *warning.* Warnings are code related and are analogous to general warnings in C++, such as comparing an *int* to a *size\_t*. Finally, the third form is called an *error.* Errors are code related and are analogous to compiler errors, such as declaring a variable with type *int*, and initializing it with type *string[[5]](#footnote-4)*.

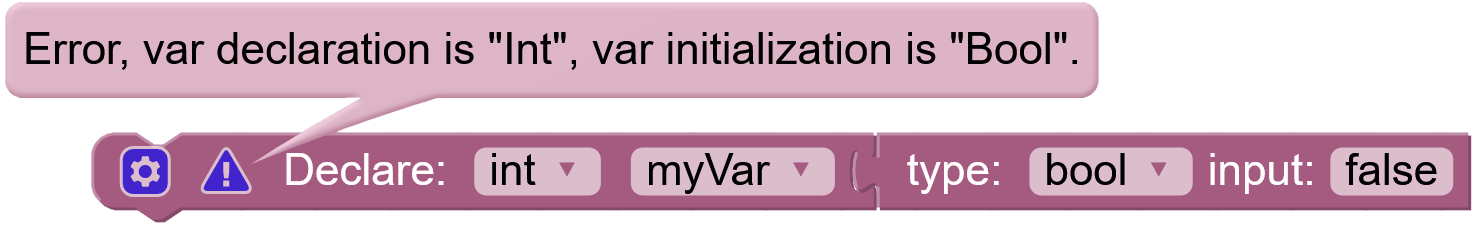
Blockly C++ has four *levels* of type and logic checking. The first type is within the individual blocks themselves, called *block checking*. The user cannot use a Variable Initialization block to declare an integer with a string.



Example 6 - Type checking within a single block.

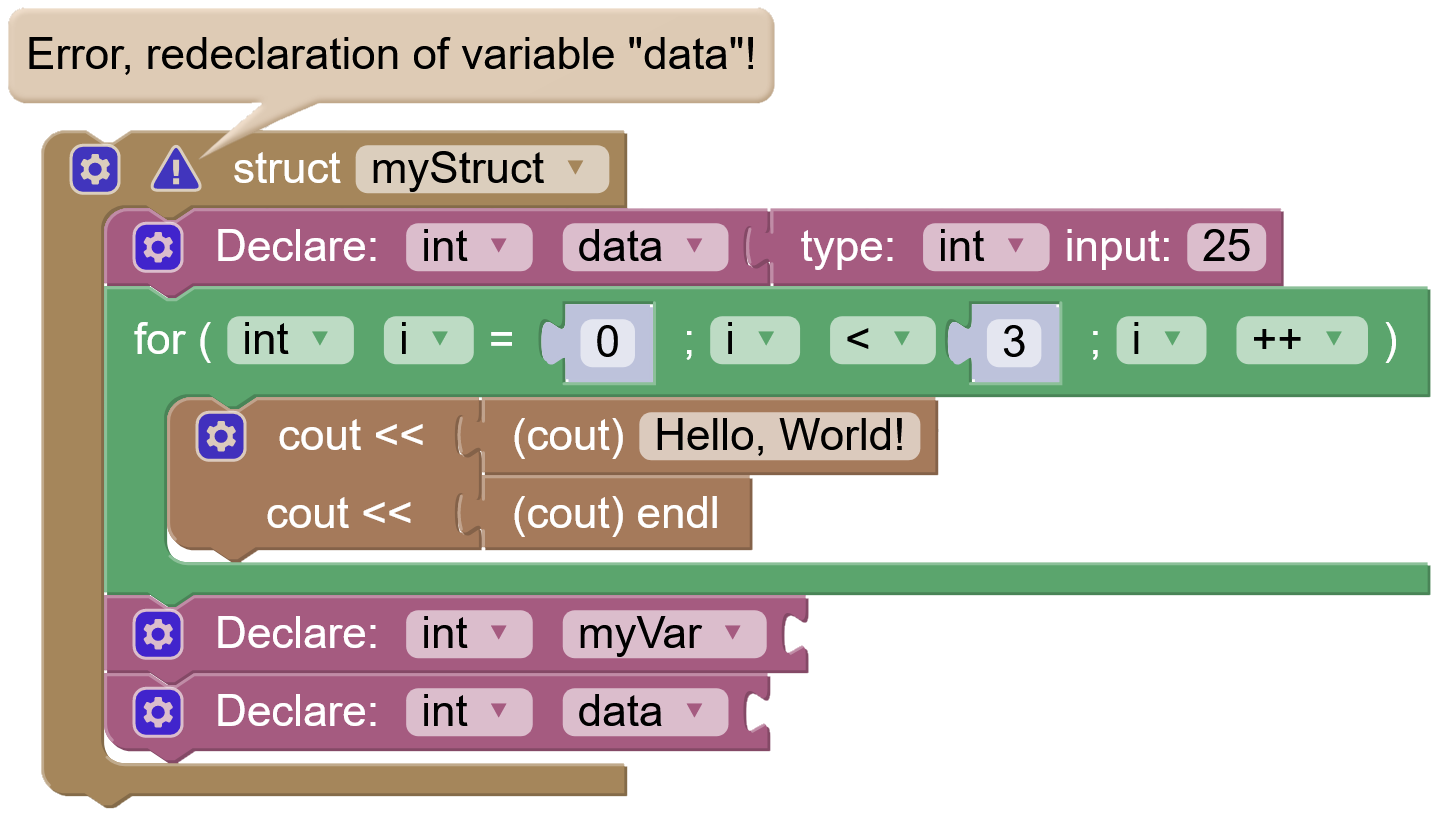
Here there are two types of type or logic checking, an *error*, and a *block warning*. Both of these are located within the one block, and do not require other blocks in order for these alerts to become apparent.

The second level is called *inter-block checking.* This form of checking occurs between two or more directly connected blocks, such as the aforementioned example of declaring a variable with an integer type, but initializing it with a string.



Example 7 - I*nter-block checking.*

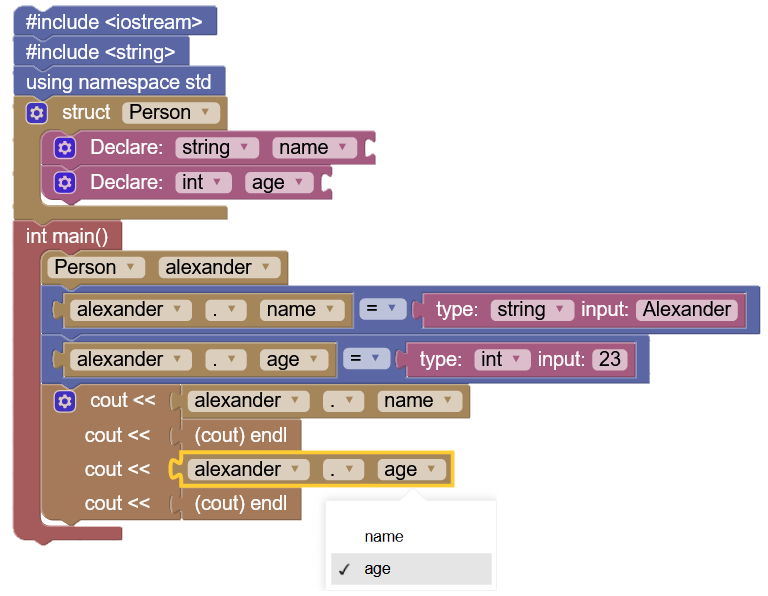
The third level is called *scope checking.* Scope checking occurs between blocks that may not be directly connected, but are scope aware. Assume that you have a *struct*, and within that struct you have multiple variable declarations. Externally, you may be able to declare a variable with the same name, but within the same scope you cannot.



Example 8 - Scope checking within the scope of a struct.

Because variable *data* is redeclared within the same *scope* of struct myStruct*,* the error occurs. However, if a variable was declared outside of myStruct, there would not necessarily be an error that would occur even if there was a variable declared with the same name.[[6]](#footnote-5)

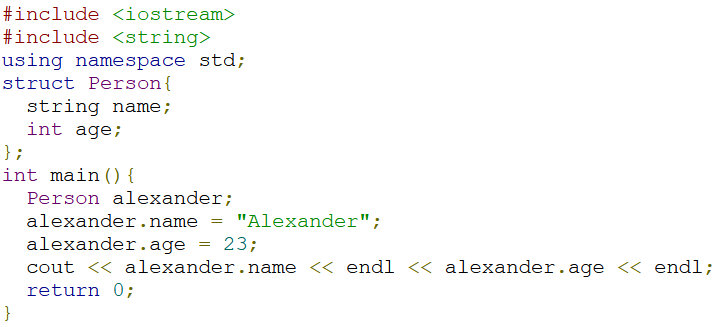
The fourth and final level is called *workspace checking.* Workspace checking occurs between blocks that may not be directly connected, but are logically predicated in the C++ language.



Example 9 - Workspace checking

In the above example, a struct Person has been declared. Within said declaration exists two variables, name and age. Workspace checking manifests in this example within the dropdown menu in “*alexander.age*” and *“alexander.name”*. In other words, Blockly C++ is scope aware of variables, functions, and class declarations that occur within the workspace. Despite the fact that this dropdown menu does not exist within the same scope as the struct, it still recognizes the declarations within the declared struct.

The purpose of Blockly C++ is twofold, on top of the aforementioned checking, Blockly C++ also produces C++ code for every block. This generated code is produced such that it may be spontaneously compiled. However, Blocky C++ does not compile the code. The following code is the generated product of the blocks present in *example 9:*



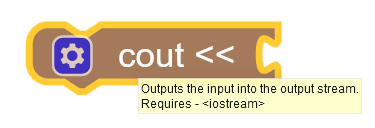
Example 10 - Code Generation

This code is generated simultaneously as the user places blocks, and its console output testing in a compiler is:

*Alexander*

*23*

Other supplementary aid exists within the blocks. If the user hovers his mouse over a block, a tooltip can appear with information relation to the current block with information about the block itself, what library it requires, etc.



Example 11 - Tooltips

Blocks can also have quick links to extra resources, i.e. the for-loop block’s link will go to an online article discussing the logic behind loops, and their implementation.

Blockly C++’s design does not only rely on code generation and checking, but also on the UI of the blocks. The user can easily identify what code he can and is generating (see examples 4 and 5). Because of this, blocks such as structs, classes, and inheritance are represented as several different blocks instead of being a single block with the option to convert.

# Conclusion

The aim of this software is to make C++ code easier to understand for beginners. Blockly C++ is continuously being updated with development plans spanning the next several months with the goal of making Blockly C++ continuously better. We plan to overhaul the variable system entirely in place of ad-hoc variable checking. Nested scopes (such as a loop within a function) will be able to accurately detect variables in their respective scopes, i.e. a variable declared in a loop “*for(int i =0; …)”* will only be accessible from within that loop instead of everywhere on the workspace.

# References

[1]. Seidman, R. H. (2009, July 1). Alice first: 3D interactive game programming. Retrieved from https://dl.acm.org/doi/abs/10.1145/1595496.1562986

[2]. Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010, November 1). The Scratch Programming Language and Environment. Retrieved from https://dl.acm.org/doi/abs/10.1145/1868358.1868363

[3] Fraser, N. (2015). "Ten things we've learned from Blockly," 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) pp 49-50. Retrieved from https://ieeexplore.ieee.org/document/7369000

[4] Qian, Binsen, and Cheng, Harry H. "C-STEM Studio: A Solution for Learning Computing and STEM Topics With Robotics and Embedded Systems." *Proceedings of the ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. *Volume 9: 13th ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications*. Cleveland, Ohio, USA. August 6–9, 2017. V009T07A041. ASME.

<https://doi.org/10.1115/DETC2017-68362>

1. 1Seidman, R. H. (2009, July 1). Alice first: 3D interactive game programming. Retrieved from <https://dl.acm.org/doi/abs/10.1145/1595496.1562986> [↑](#footnote-ref-0)
2. 2Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010, November 1). The Scratch Programming Language and Environment. Retrieved from <https://dl.acm.org/doi/abs/10.1145/1868358.1868363> [↑](#footnote-ref-1)
3. Fraser, N. (2015). "Ten things we've learned from Blockly," 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) pp 49-50. Retrieved from https://ieeexplore.ieee.org/document/7369000 [↑](#footnote-ref-2)
4. Qian, Binsen, and Cheng, Harry H. "C-STEM Studio: A Solution for Learning Computing and STEM Topics With Robotics and Embedded Systems." *Proceedings of the ASME 2017 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. *Volume 9: 13th ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications*. Cleveland, Ohio, USA. August 6–9, 2017. V009T07A041. ASME.<https://doi.org/10.1115/DETC2017-68362> [↑](#footnote-ref-3)
5. Such as *int num = “Hello, World!”;* [↑](#footnote-ref-4)
6. Such as namespaces and class members like variable *data* and a struct variable *myStruct.data*. [↑](#footnote-ref-5)