**Assembler using C++**

1. **Introduction:**

**Brief overview of the project:**

Our project is an assembler written using C++ programming language. The primary objective is to create a tool that translates human-readable assembly code into machine code, enabling computers to execute the specified instructions.

**Objectives and significance of implementing an assembler:**

An assembler eases the way programmers can write and debug codes. They use the human-friendly assembly code instead of the more complex machine code that results in more errors. They can also implement better optimization using low-level languages when working with systems like embedded systems than when using high-level languages.

1. **Technical Background:**

**Basic concepts of assemblers and their role in computer organization:**

Assembler converts assembly code to machine code for computers to understand and execute the provided instructions. It acts as an intermediary between the programmers using assembly code and the complexities of machine code.

**Overview of assembly language and machine code relevant to the project:**

Assembly language is human readable structure of instructions. Machine language is a numerical language that consists of 0s and 1s that tell a computer to directly execute an instruction. An assembler converts the instructions written in assembly code to machine language to be understood and executed by the computer.

1. **Methodology:**

**Detailed description of the programming language used:**

For the implementation of our assembler, we chose the C++ programming language.

C++ is efficie­nt, adaptable, and provides low-leve­l programming tools. We chose C++ as it is one of the languages that we are most skilled at making it much easier and more efficient when producing a better outcome for the project.

**Explanation of the development environment and tools utilized:**

We used Visual Studio 2022 as our main development platform. Visual Studio is a robust tool for developers, allowing all aspects of the development cycle in one place. This IDE equipped us with features like highlighting errors and debugging assistance, all of which boosted our efficiency during the coding process. We put it to work compiling C++ source code which guaranteed language compatibility and helped produce the final executable file.

1. **Implementation Details:**

We start off by including libraries and initializing used variables, strings, and arrays.

A screen shot of a computer code

Description automatically generated

A computer screen with text and numbers

Description automatically generated

We created an array of strings to include the instruction and its hexadecimal code. The instructions were divided as memory-reference, register-reference and input/output reference.

A computer screen shot of a code

Description automatically generated

The text file including input was read and each line was copied into an array of strings. This helped in the iteration and searching for labels with their memory locations.

Also, the pseudo instruction ORG was looked for to know how to create the symbol address table. The counter string takes the number along the ORG (if org was present) and converts it to integer into the ct variable.

The ct was decreased to be able to start the next instruction in the code with the correct memory location. If ORG was not present, we would start our ct from 100.

A computer screen shot of a program code

Description automatically generated

The program starts iterating through the created array.

A black screen with white text

Description automatically generated

If the length of the current line was more than 4 meaning it includes an address of some kind or a label.

If the character right after the instruction code is an integer, this means that it indicates a memory location. The instruction hexadecimal code is searched for in the made strings and output along with the memory location.

Also, the presence of I(indirect) is taken into consideration, and this is done by counting the number of spaces in the line. If there are two spaces, three words present so I is included.

If I is present, the third column in the memory instruction string is used, else the second column is used.

A computer screen shot of a program code

Description automatically generated

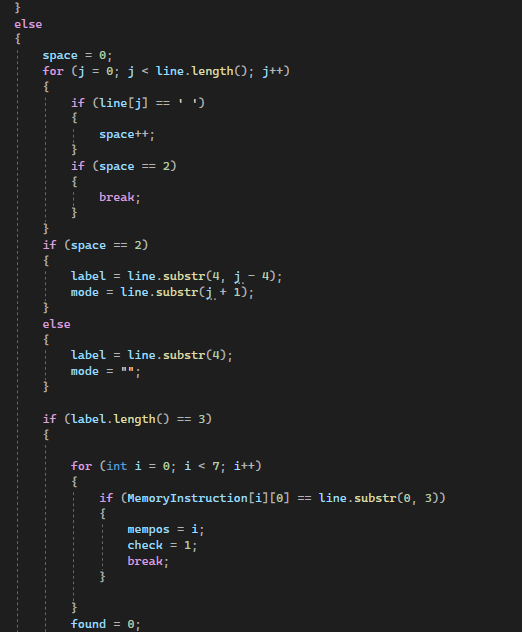
A computer screen with many colorful text

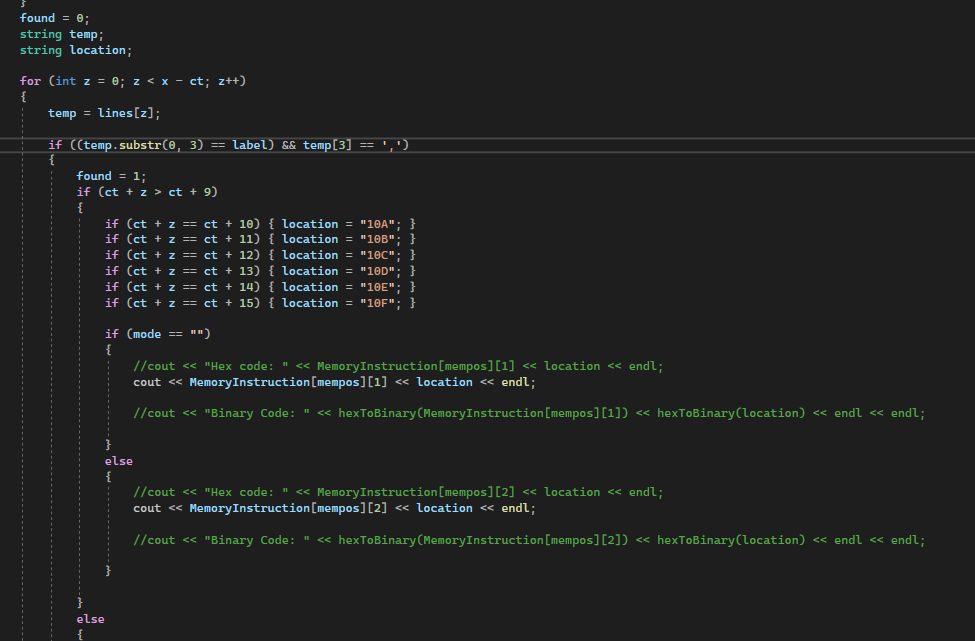
Description automatically generated with medium confidence

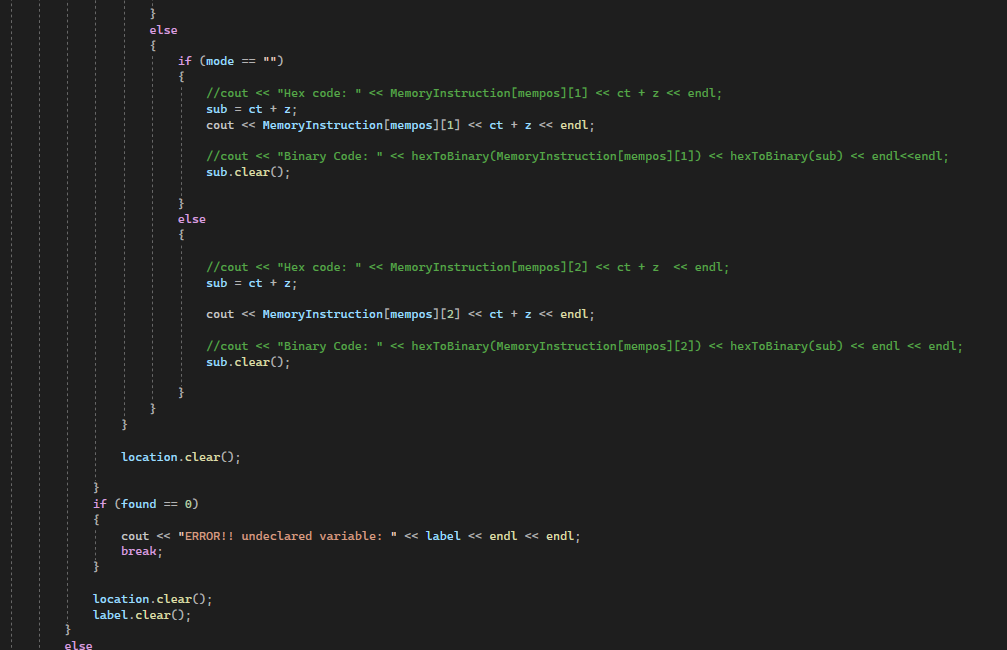
If the character right after the instruction code is NOT an integer, this means that it indicates the presence of a label.

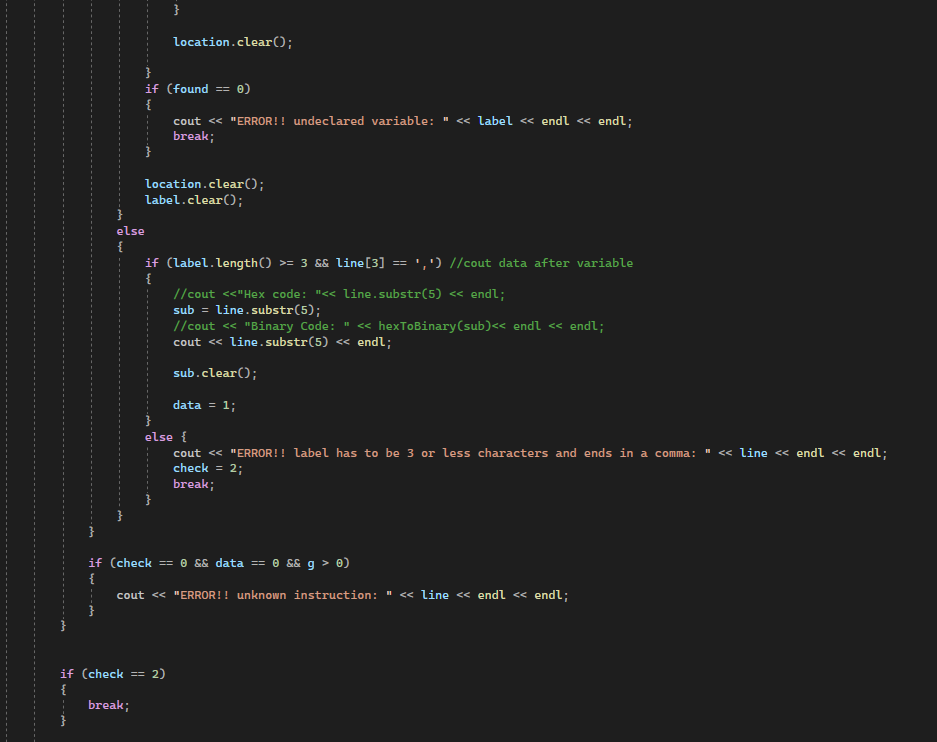
The label is searched for in the rest of the program to identify their location (symbol memory location). If their location exceeds 109, the hexadecimal code for values 10 and above is displayed instead.

The instruction hexadecimal code is searched for in the made strings and output along with the memory location of the label.



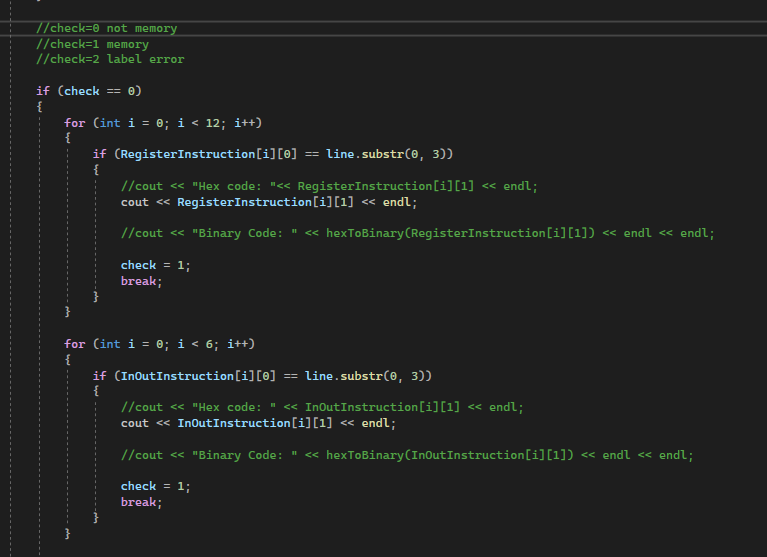


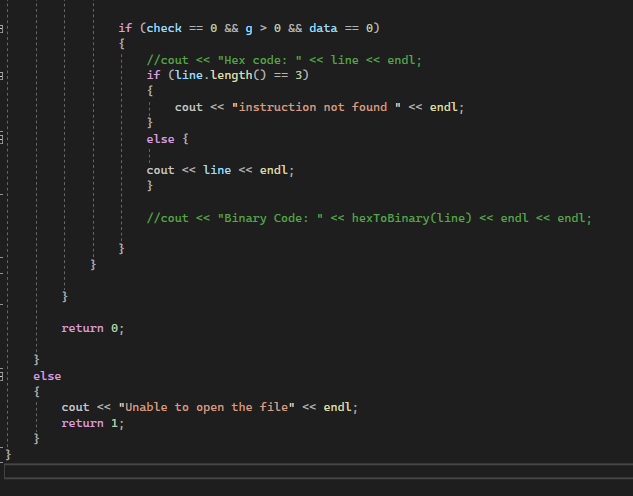




The strings are cleared after usage to avoid over accumulation of data.

If an instruction was not found in memory, the check remains 0 and it is looked for inside the register reference string and the input/output reference string.





**Errors:**

The presence of errors regarding the user entry were taken into consideration.

Labels for variables must be 3 or less characters and must be terminated with a comma. The programs display an error if the labels exceed 3 characters.

If an instruction was not found at all (invalid instruction), an error also gets displayed.

An error gets displayed if a variable was called and its label was not declared in the program.

**Functions**:

is\_integer was used to determine if the character is a digit.

hexToBinary was used to convert the hexadecimal string to binary string.

1. **Challenges and Solutions:**

**Challenge 1:**

- Working with memory addresses and labels was difficult, particularly when figuring out addresses using the ORG value and handling labels in memory commands.

**Solution:**

- Two phases were implemented to overcome this challenge. On phase one, details about labels were collected and found the matching memory addresses. On phase two, this data was used to work out addresses when creating code.

**Challenge 2:**

- Difficulty in spotting and flagging undeclared variables or unfamiliar instructions, especially when dealing with labels and memory commands.

**Solution:**

- A detailed error-check system was created to spot and flag issues, like undeclared variables or unfamiliar instructions. The error messages were clear to help with debugging.

1. **Results and Discussion:**

**Examples:**

**\*Using ORG, no labels**

A screenshot of a computer

Description automatically generated

**\*no ORG, no labels**

A screenshot of a computer

Description automatically generated

**\*ORG, label**

A screenshot of a computer

Description automatically generated

**\*no ORG, label**

A screenshot of a computer

Description automatically generated

**\*indirect, no label**

A screenshot of a computer

Description automatically generated

**\*indirect, label**

A screenshot of a computer

Description automatically generated

**\*adding a comment**

A screenshot of a computer

Description automatically generated

**ERRORS Examples:**

**\*label > 3**

A screenshot of a computer

Description automatically generated

**\*undeclared label**

A screenshot of a computer

Description automatically generated

**\*unknown instruction**

A screenshot of a computer

Description automatically generated

1. **Conclusion:**

The project provided a deep understanding of assembly language through the different types of instruction references. By making strong error-checking systems, we learned about handling undeclared variables and unknown commands.

The project experience was benefitable as it showed us the importance of following the rules for any type of application or program to ensure its successful execution.

The two-pass approach indicates how vital it is to plan and design your program to face its complex requirements.