Allakhverdiyev, Dmytro - dallakhv
Darling,Bailey - bdarling
Li, Wilbur - wiwili
CMPS 109

# Project Phase 1 Report

## Building with G++

Building a program with G++ requires preprocessing, assembling to object files, and linking. G++ operating in a terminal has a shorthand for completing all of these tasks with just a simple g++ (all cpp files) -o (output)  -std=c++(version). There are manual flags for creating object files if there was a need to step by step, which is actually useful for the makefile. A makefile contains directions for building all parts of the program individually, and will create new object files as cpp files get updated, drastically on compile time of large projects. This works with a top-down build order. It will attempt to build the program with the defined name and parameters. If any of the parameters are not found, it will go down and look for it to build until everything is built. It might seem annoying to include new cpp and header files every time something is added, but makefiles have very useful macros that allow for easy integration of new files. For this project, a makefile is provided as required, generating a program of mis in the root.

## Design

The UML design stage was definitely the most difficult stage of this project. There were multiple ideas tossed around at the beginning, but for a while we stuck with an interpreter design that was proposed by Bailey. Following the lecture on STL containers and being introduced to the factory, Wilbur came up with a design that would be an alternative to the interpreter design put forth earlier. This design utilized the process of having expressions inherit both a base class and the number of arguments that Karim demonstrated in class. This did not work, however, and the final design would incorporate elements of both and were iterated upon. Like for example, the earlier iteration did not have a class for logging outputs and errors and was added as the need arose.

The final design can be divided into main parts, the framework and the expressions. The framework consists of the client, parser, storage, logger, and factories working together to make a unit that reads likes and interprets what command will be executed. Expressions are commands that client will execute, and represent the different commands that the Machine Instructor Simulator can do. These will be detailed below in their respective sections, starting with the framework. Although it mentions who primarily implemented it, there are countless minor changes made by each member in each class to get it functioning up to spec and bug free.

## Framework

**Client:**

Implemented by Wilbur. This can be considered the main program and the purpose of the client is to manage the expression factory, storage, parser, initialize the logger's file name. The flow of the client is to accept the file, check for the correct extension, feed the file's code into the parser, run a check of LABELs, then continue onto normal code execution with a generic Expression* calling execute. Execute is a command that is in the abstract class of Expression*,

and through polymorphism, client can just generically call the function and execute logic commands.

The hurdle with this, however, is that executed commands do not have access to client's storage and parser for modification. It was because of this that we came up with two options. The first is to have the pointer to client's storage and parser passed as arguments to execute, allowing the expressions to do whatever they wished to it. The second is to have parser and storage be static, allowing any class that has scope of them access it. We went with the first option because it was simpler to implement since this issue arose after programming began, and it seems slightly safer than having those classes be static.

Another issue that may arise is whether or not to delete the old output files here, so that every time the program is run the old data gets deleted.

**Parser:**

Implemented by Bailey. The job of the parser is to do some token clean-up and save the code. Cleaning the tokens is done by erasing them before storing them into the vector. The token cleanup works on actually inserted tabs and newlines and such, but for some reason does not clean the written "\n", "\t", and "\r" as strings. It was left as is because out needs these for printing to the output, and if the parser completely erases those strings then it would not work properly.

There was a choice between saving the code as a vector/map or , but after deliberation it was decided to be a vector. Saving the code makes it much more convenient to access in the future, and with an index variable it is very easy to have JMP change its location. The issue here that may arise is with multi-threading. Having JMP change the index with a pointer does not seem very thread-safe, but it was not tested and will probably see some change in the future.

The original implementation had the parser getting the next line following, but this was changed to only retrieve the current line indicated by the index of parser. This was changed because the next line functionality was very buggy and ran into problems with segmentation faults or sometimes did not parse lines correctly.

**Storage:**

Initially implemented by Dmytro and further implementations by Wilbur. One of the most difficult classes to devise and implement. There were many iterations of this design, and many of them did not work. These designs ranged from having a map that holds everything, having multiple maps that hold each individual type, having maps of tuples or pairs, making it a singleton, and so forth. The design that was settled with uses a factory and a map of base types pointers, map<string,DataType*>. While not the most effective or generic, it does hold the data needed without crowding the private members with too many maps.

Although, the side effect of this is that there are many public functions, as polymorphism does not allow the base class to access it's children's functions. There are probably a million more ways that this could have been implemented, but this is the one that we have to stick with for the duration of the project.

In overall client flow, it is instantiated by client only once, which made the idea of having it as a singleton somewhat appealing but it that idea was scrapped. The pointer to this instance is then passed around by execute to the expressions that need to access the data. Overall this design

still encapsulates the data as there are only creators, setters, and getters for the data stored, even though nearly every class has access to it, though the thread safety seems somewhat questionable.

**Logger:**

Implemented by Wilbur. This was actually created some ways into the project when we realized that we had to output to two files, and error and output file. Since it was not clearly specified, we assumed that the output and error files had to have the same name as the mis file, like if the file is test.mis, the outputs would be test.out and test.err. When originally conceived, it was to be passed around through execute like storage and parser, but since this class is much simpler in implementation, it was made into a static class. Client would initialize the file name from its argument field, then every class will include the logger in the header either to inheritance or direct #include.

There are only two functions associated with this, and they just append lines to the end of filename.out and filename.err streams, which are a part of the specs. Being static, they are global and can be accessed by any class in the scope. This makes it very convenient to output strings to files. One design decision that is a bit troubling is whether or not to flush the output files every execution so there isn't a whole list of errors on repeated executions.

**Abstract Factory:**

Implemented by Wilbur. An abstract template class for a factory. Originally there was only one factory in our design before it was expanded to two, the DataType and Expression factories. As a result, this abstract template base class was created. It contains a template map<string,T*>, which serves as the base of the factories. It was made into a template because unlike the DataType class, which was meant to be a template class as well, this base class is never used for polymorphism.

**Expression Factory:**

Implemented by Wilbur. The original design had the parser using if statements for expressions. As a result of the lecture on STL containers, this factory was born as the only factory to be used in this project. Its implementation is pretty much the implementation from the slides. There is a map that holds all instantiated empty objects of type Expression*, which are then identified and cloned and initialized, as lines come in from client through getObject(). These objects then get passed back to the client to have execute() run and execute the logic.

Since LABEL was preprocessed

**Data Type Factory:**

Implemented by Wilbur. Following the debacle of trying to figure out storage, the solution that was settled on was the use of a factory to generate data type objects. Similar to the Expression factory, this factory generates objects of DataType. This works mostly, but storage unfortunately cannot access data of specific types due to the way polymorphism works. At best, storage can get the type of data, which is useful in itself.

**Abstract DataType:**

Implemented by Wilbur. Originally, this was supposed to be a template class that each of the data types inherit from and templates their data type as a private member. This did not work as storage needed to call DataType* in its functionality, similar to how client called Expression's

execute(). As a result, the abstract DataType is only an abstract and not a template, unlike the Abstract Factory.

**MISNumeric:**

Initially implemented by Wilbur, further improved upon by Dmytro. First implemented as a test for the abstract DataType template and was found to not work. It was then left as a test data type for the proof of concept for storage. Following this, Dmytro picked up working on numeric and polished it. It was found that in the instructions, the size of a NUMERIC in MIS should be 8 bytes. Originally it was meant to be an int, so it was changed to a long long, which is an 8 byte instruction.

**MISChar**

Implemented by Dmytro and modified by Wilbur. The original iteration of this data type did not have very much checking. It simple accepted chars and inserted them on the assumption that they were valid chars from strings. This lead to problems with things like tokens, which in string representation are two characters. Conditions were added to look for them, like "\n" for example, and input '\n' directly as there is only 1 char to store. In basic operation, CHAR will shed the ' ' only on the first pass creation and relies on ASSIGN to shed the ' ' and provide proper special characters for it in future modifications.

**MISReal:**

Implemented by Dmytro. Very straight forward classes that act similar to MISNumeric. MISReal uses a double as it is 8 bytes, as per the requirement of it being 8 bytes.

**MISString:**

Implemented by Dmytro and modified by Wilbur. This data type was one of the bigger problems in deciding what to do about storage. It was a struggle deciding whether or not to go with an array of chars, vector, or a string. In the instructions, it was listed to be an array of characters with a size from 1-256. Initially implemented with a string, it was unsure of whether or not to keep this or change to a char array. Wilbur attempted to make this into an array of chars, but the functionality was very difficult to implement and led to numerous errors. In the end, string was decided upon because in the end, a string is an array of char*, so it does meet that criteria from the specs. It is important to note that in this current iteration, the class will shed quotation marks only on initialization. Further reassignment requires VAR to shed quotation marks prior to assignment. It was not quite clear whether or not variables of type STRING have to keep the quotation marks, but this implementation can be very easily changed.

## Expressions

**VAR:**

Implemented by Wilbur. One of the most important Expression objects that signals the assignment of variables with their data. It passes the two arguments as strings and leaves storage to parse and insert them as needed. The reason why is because this is one of the earliest Expressions implemented and lacks type safety, yet it is the backbone of variable assignment. As a result, many of the operation features are handled by storage. If there were more time, this would be rewritten so that it never calls storage in the first place if the variable is not of correct type.

**ADD, SUB, DIV, MULT: These are grouped for their similar design and functionality**

Implemented by Bailey. In an earlier design, these classes were meant to be under an abstraction called MATH that contained partial implementation and a template for what type of operation it would be for its functionality. However, that idea was scrapped and all now just inherit from Expression directly. With the current iteration, since execute takes in a pointer to storage, variables can be manipulated directly by the class. So in this case, initialization merely sets the name and its private members to 0.

For normal operation, these operations function very similarly. Execute has access to both the storage and parser provided by client, so they are able to fetch and assign variables as needed. Each member also contains a function called cleanString(), which cleans up any white space or tokens that were not cleaned up by parser for whatever reason. Each class will check if they are working on either a NUMERIC, REAL, or variable of correct type, with DIV having an additional check for dividing by 0.

**ASSIGN:**

Implemented by Wilbur. This operation takes two parameters and puts parameter 2 into parameter 1. There are checks needed to make sure whether or not they are of the right type, so standard flow goes as such. Upon cloning, the object is initialized. The first thing initialize does is look for the first variable, if nothing is found then the operation has no parameters and an error is thrown. Then, the parameters are actually checked to make sure there are exactly two if it is not a string or char. This is done by counting the number of commas present. This addition was added later from testing it in SLEEP, which was coded later.

The way this works is by first checking if a " or ' is present, if they are then it designates it as a STRING or CHAR. They are both required to have terminators at the end, so if they are missing it is an invalid command for invalid parameters or unclosed terminators. If the length is 1, because beginning and end read a terminator, then it is also invalid. If it is a REAL, NUMERIC, or variable, then we can count commas, since all of these values cannot have commas in them. Since ASSIGN takes 2 parameters, there can only be 1 comma. However, since the first comma was taken as the delimiter for parameter 1, parameter 2's string should contain 0 commas. If it does, then it goes to the next phase, if not then it will fail.

During execution, it will search to see if parameter 1 exists and if any of the earlier checks failed by checking the parameters for "undefined," which was chosen as the constructor value because "undefined" is an invalid string anyway as it lacks "", which is needed for the conditional. This then leads to branching situations of reading the type of parameter 1 and if parameter 2 is a variable by checking for the $ that is required in every variable name. If it does not exist, then it is not a variable, and it goes into the constants branch. The variable branch I can assume that it is whatever is stored in variable of proper type so it is just directly assigned. For string, it is only assigned if the length is less than the destination length. The specs did not make it a clear requirement but it was implemented that way and can very easily be changed.

For constants, STRING and CHAR had their checks done earlier, so their checks are pretty simple compared to the others two. NUMERIC and REAL use strtoll and strtod to check if the string they received is actually a number. This works by providing a pointer *p of type char, which strtoll and strtod will dump the first non-numerical character into. If this does not happen and *p remains NULL, which evaluates to false, then it is of matching type and can be assigned. Otherwise it will throw an error over to the logger.

**OUT:**

Initially implemented by Wilbur where token detection did not work and re-implemented by Dmytro. In its initial implementation, all it did was output the commands with no parameter checking and did not output tokens. Dmytro improved upon this design by adding multiple layers of checks for finding tokens with string representation and converting them to character representation. The old code is still present in but commented out and considered obsolete.

**GET_STR_CHAR:**

Implemented by Dmytro. Gets the character at index of a string and sets the value of a char variable to it. During initialization, this uses peek() as the method of checking whether something is valid for initialization. For normal operation, this function will check for type validation with getVariableType() provided by storage and return any time mismatching types are found. Another important check it does is make sure that the index provided is not out of bounds of the string, otherwise there will be access violation issues. After it passes all checks, it will simply set the data to char's value.

**SET_STR_CHAR:**

Implemented by Dmytro. Opposite of GET_STR_CHAR, this function will take a char and insert it into string at index. Many of the operations used for initialization and checking are similar. The primary difference in how insert is done. The string to be modified is retrieved, has the character inserted at index, and data reinserted through setStringData().

**LABEL:**

Implemented by Wilbur. Labels are processed done immediately after the parser is created and before standard execution. Initial implementation lacked checking but did the create labels. The LABEL expression is ignored by the expression factory to simplify having to create a LABEL class that does nothing. On cleanup after learning of new techniques, checks were added to make sure labels are not created with empty strings, commas for parameter issues, and tokens via the counting method.

**JMP:**

Implemented by Wilbur. An unconditional jump to a label. The way this function works is it checks for a label, if found gets the index, and sets parser's index to it. Somewhat worrying for the future assignment of multi-threading, as this means that multiple loops can interfere with each other's jumps, but it works for a sequential MIS. Will probably be changed in the future along with parser for better multi-thread implementation.

**JMP(Z/NZ):**

Implemented by Wilbur and updated by Dmytro. A conditional jump to a label if the variable provided is 0. It uses basic checking of variable types to retrieve the right data and jump to the label. If the retrieved value is 0, then it either jumps or does not jump to it. Originally this was meant to be inheriting a base class of JMPZNZ where initialization was defined, and both JMPZ and JMPNZ would inherit and redefine execution. The compiler used gave several errors about constructors instantiating and assigning in incorrect order, so the idea was scrapped and made into 2 classes that defined their own initializer, despite being identical. At the time, it did

not make sense to apply constants to this logic because it would basically treat it as a non-execute or unconditional JMP, but as Dmytro noted, it is in the specs so he updated the code. Doubles were used for checking because even though there might be loss from the conversion is because 0.1 vs 0 should not result in a jump.

**JMP(GT/LT/GTE/LTE):**

Implemented by Dmytro. Conditional jump where Another class that intended to have a base class that contains logic for initialization and have a minor difference in execution. Initialization is done much the same as the others that use peek() for confirmation. Standard execution flow will check the variables and constants, then make a comparison that is respective to what the class intends. JMPGT means P2 > P3, JMPGTE means P2 >= P3, and soforth. The logic for using doubles here is identical to in the prior JMP(Z/NZ).

**SLEEP:**

Implemented by Wilbur. The first class where the idea of counting with commas to check parameters came from. It initializes first by checking for an empty string, if passed then it counts the commas. This works because SLEEP cannot take a string or char and only has 1 parameter, so it has no business with commas. If commas exist, then there is a problem and if it doesn't, then there can only be one parameter.

Functionality uses <thread> and <chrono>, which provide sleep for a thread and time respectively. Chrono sleep does not take doubles and floating points, so everything is typecast into integers. I believe it takes long long as well for the 8 byte requirement, but my specific compiler gave errors from chrono. It runs standard checks on variables to retrieve their data and cast it as a long. For numerical ints, this is the first instance of using strtol to count lines as an experiment, and back ported to the other expressions. This was previously explained under ASSIGN.