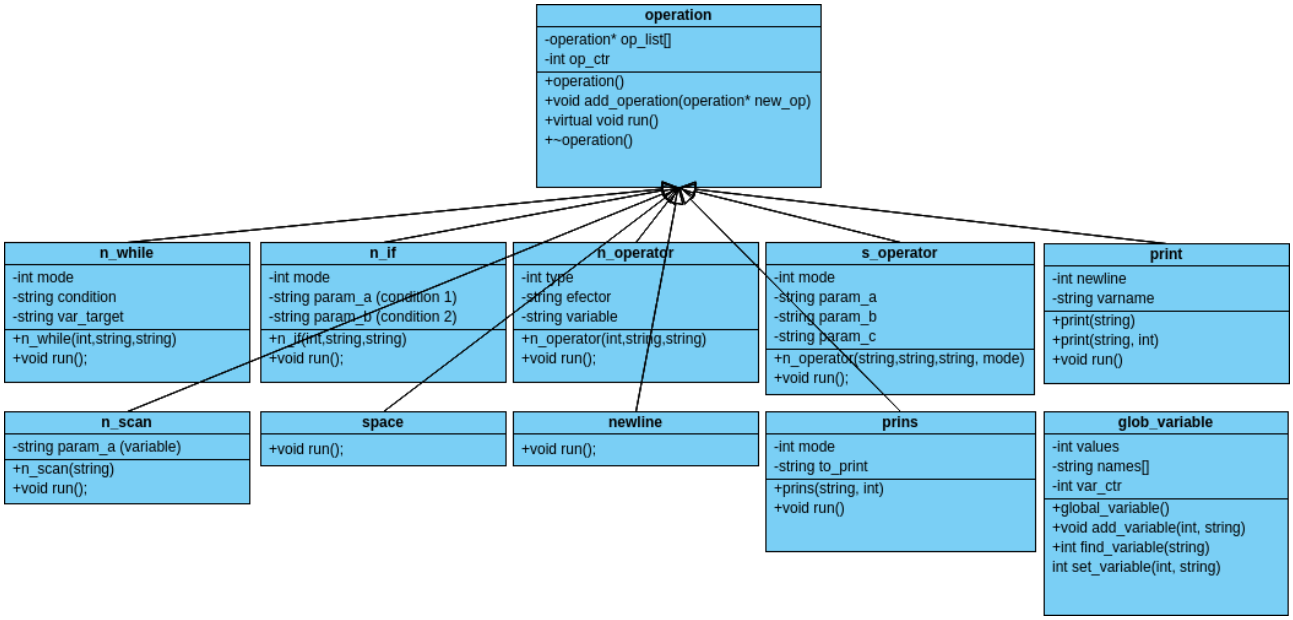
Description :

This interpreter will run user's code (written in specific syntax) and make a new binary. The program will interpret user's code and run it, the program allows you to create a new binary which will contain the code appended to the end of the executable, which will then be extracted and interpreted.



Design

Classes :

1. Operation Class : Main class, which will be the parrent of all the operations.
   private : Operation** op_list;, which will be the list of all operations to do (For the branching statements e.g if, while,for)
   int op_ctr, cunts the number of operations in a branching statements.
   public : void add_operation(operation* new_op) this adds new operation to the op_list pointer. also increments op_ctr
   virtual void run() : this is a generic run, only used in an if statement, just to make the recursive function works. other classes will implement their own specific run() function, depending on their functionality
   this class has a run() code in order to make if statement work. Because if statement needs to have two operation* array; having an entire class would be easier, as it could be plugged in directly to the recursive function without having to modify add_operation() function.
2. While Class : will do while statements
   private : string param_a, string param_b, to check value with the class glob_variable
   int mode. (<,=,>)
   public : void run() : this will run the if statement. while(param_a mode param_b)
   { for(int i=0;i<op_ctr;i++) op_list[i]->run(); }
(runs all the statements inside this branching statement)
3. Operator class : will do ariithmetic operations
4. n_if class : if statement

has the usual op_list pointer, and extra alt_op pointer (public) so that the operation.add_operator could be accesed by determine function, which takes the pointer to operation as argument
when determine() read stop, it will return negative value, which will terminate the reading (while(determine(cur_op/alt_op))>=0)

5. n_while class : does while statement
this class will accept operation* to be inserted into it's operation* op_list array, continues to add until determine find "stop" command, in which it will return negative value, stopping the recursive function
run() will run all the statements inside (for i=0 until i=op_ctr) depending on the argument passed.
for example : while i=0 < 5, op_ctr=3, this will run all content of op_list from index 0 to 3 5 times (if "+ i 1" every loop).

6. print class : will take string as argument(to search it with glob_variable class), this class doesn't accept integer as parameter so it could update itself, incase it's situated inside a loop
run() function on this class will print integer, which is searched from glob_variable class if a name(string) matches.

7. prins class : will take string argument, will print it.
run() function will print the string

8. newline & space class : take no arguments, run() will just do cout<<" " or cout<<endl;

9. s_operator class, this will does the exact same operation as operaator(with addition of modulo), but will store result in param_a

10. glob_variable class : this class will provide the functionality of creating new variables with certain names(string) that has certain integer value, and getting an integer value from a variable name(string). Globally declared, so that it can be accessed from any part of the program (classes and functions)

all classes (except for operation and glob_variable) is inherited from operation class

Global Variables :

ifstream* input_stream : pointer to ifstream, initialized in the main function, so that it will be able to be accesed from any part of the program.
operation* p_operation_main : the pointer to the main code list, done this way to make the recursive function work.
glob_variable global_variable : the object of the class glob_variable, that acts as a storage for variables.

functions :

void determine(operation* cur_op)
{
recursive function, will take pointer to operation , and will do cur_op->add_operation(operation*) when a command is found.
}
void extract(string filename)
{
will extract the code from inside the executable and store it to ".code", will then call determine_helper to start executing the command
}
void generate(string this_executable, string input_file(bob), string output_filename)
{

will copy the current binary file, and paste the input_file into the end of the copied binary. when extract() is run, will extract the content and run it.
}

Testing :

1. Write a .bob program, check if it runs correctly
2. Write a c++ equivalent program, check if it runs correctly
3. Compile both bob program and c++ program
4. diff bob.out cpp.out
5. if there are any difference(given that the logic of both programs are correct, then there is something wrong with the compiler, else it's working fine)

If the interpreter is working correctly, then diff will not show anything. I will write a bash script that will automatically check if diff returns anything. If it doesn't it will print PASS else, it will print ERROR

syntax :

var var_name value
ex : var i 0
will create a variable named i with value 0

print i
will print the value of variable i

prins string
will print the string

+ i 5
will add 5 to i

- i 5
will subtract 5 from i

the same with divide and multiply

++ a b c
will store the result of b+c into a

-- a b c
will store the result of b-c into a
same with multiply and divide

% a b c
will store the result of b modulo c into a

if a = b
if a > b
if a < c

do_this1
do_this2
do_this3
stop

do_that1
do_that2
do_that3
do_that4
do_that5
stop

if the statement matches, do_this else do_that. else=stop

while a < b
while a > b
while a = b
do_this1
do_this2
do_this3
stop
endwhile=stop

scan (var)i
will ask user for input(integer) then store the input into variable i

automatic testing : check and compare if the output produced by the .bob files is the same as .cpp file.