Logical Problem Solving

The goal of this document is to provide information and to educate individuals on the importance of utilizing both critical thinking and all available tools when faced with programmatic problems.

- Logical Problem Solving
  - Breakdown of Logic
  - Step By Step Programming
    - Translating Code Into Language
    - Translating Language Into Code
    - Translating Complex Language Into Simple Language
  - Troubleshooting
  - Documenting Code
  - Note Taking Reference

Written by Michael Warmbier

December 12th, 2022

Breakdown of Logic

Logic is a term we use frequently when describing solutions to problems, refers to the validity proven by reasoning reliant on formal rules, whether defined by nature or the human mind. With computers, all logic resolves to simple *true* or *false* questions. The answers to these questions provide information to the machine that determine its actions. Programmers write *programs* that answer these questions.

This type of reasoning is purely theoretically and overly formal, but we can use it to make a thematically appropriate *logical conclusion* that if programs rely on providing gradual instructions, then we can represent those instructions linguistically as well.

Step By Step Programming

Breaking down a program into digestible tasks is an absolutely necessary skill in problem solving. This could be done in several ways, including:

- translating code into language
- translating language into code
- translating complex language into simple language

Translating Code Into Language

Understanding the terminology used to describe concepts is important, to be as concise as possible. Instructions such as:

```
int myValue = 3;
```

Example in C++

Can be described in a number of ways (in descending order of specificity):

```
- Create a variable with a value of 3
- Define an integer with the value of 3
- Declare an integers and assign it an initial value of 3
```

By being more specific in your phrasing you may prevent more mistakes. For example, the first description in this list does not specify the *data type*, which may be important in a *strongly typed* language such as C++ or TypeScript. Additionally, both the first and second descriptions in this list do *not* specify that the value is declared initially, which may or may not be relevant to the task.

Translating Language Into Code

The process of translating human language into code is often the easiest. Given the task:

```
- Define a function, isEven(), which which takes an integer argument and re
- Within isEven(), store the evaluation of whether the argument is even or
- Return the evaluation
```

This problem is broken down into three steps that may easily be written in *pseudo-code*:

```
function isEven(integer)
boolean = (expression)
return boolean
```

This pseudo-code is useful because it may be applied in the context of nearly any high-level programming language.

Note: just like a typical exam problem, the expression to create the evaluation would be up to you to determine.

```
int isEven(int operand) { // Step 1
bool eval = (operand % 2 == 0); // Step 2
return eval; // Step 3
}
```

Example in C++

```
def isEven(operand): # Step 1
eval = (operand % 2 == 0); # Step 2
return eval # Step 3
```

Example in Python

```
function isEven(operand) { // Step 1
let eval = (operand % 2 == 0); // Step 2
return eval; // Step 3
}
```

Example in JavaScript

Translating Complex Language Into Simple Language

It's common for questions given in college examinations and courses to be difficult to decipher and provided through blurbs of text. You may get a problem such as this:

```
"Functions are useful for completing simple tasks.
For example, returning the parity of a number.
Your task is to create a function that utilizes an
integer and returns a boolean value informing the user
on whether or not that number is even"
```

This is the same problem as above, but prior to being broken down into steps. It's much harder to keep track of key concepts and words. This example is small, but you may imagine how much more confusing this may be when provided with larger questions:

```
"Create a UEmployee class that contains member variables
for the university employee name and salary. The
UEmployee class should contain member methods for
returning the employee name and salary. Create Faculty
and Staff classes that inherit the UEmployee class. The
Faculty class should include members for storing and
returning the department name. The Staff class should
include members for storing and returning the job title.
Write a runner program that creates one instance of each
class and prints all of the data for each object" (Java)
```

Broken down into easier to read steps and notes:

```
/* Predfined Data */
- Create a class, UEmployee
- Member Variable: Uni Employee
- Member Variable: Salary
- Member Method: Uni Employee GETTER
- Member Method: Salary GETTER
- Create a class, Faculty, inherits UEmployee
```

```
- Member Variable: Department Name
- Member Method: Department Name GETTER/SETTER
- Create a class, Staff, inherits Uemployee
- Member Variable: Job Title
- Member Method: Job Title GETTER/SETTER

/* Program */
- Create instance of: UEmployee, Faculty, Staff
- Use all GETTER methods

/* Notes: */
* Setter methods must be defined, but are optional to use
* Variable names not specified
```

Although longer, this version of the problem is much easier to isolate into sections and problems to solve one at a time. It makes use of notes to include information either excluded or implied, as well as white space and indentation similar to something like Python.

Depending on what you use to to write these steps down (either a specific application or through hand written notes) you may even highlight certain keywords, such as **getter** and **setter** so that you are sure to remember them.

Troubleshooting

The environment (IDE) that you use for compilation is very important. This is obvious in many ways, but often overlooked when it comes to troubleshooting. *Errors*, whether thrown by the programmer manually or as a result of syntax errors and such, may be described differently depending on the application.

For example, the error `segmentation fault` is common and easy to look-up. This, combined with your language and possibly the context for your issue make quick troubleshooting an error easy. However, some compilers may return an error such `memory error` or `stack overflow`. Even worse, you may come across a description as vague as the addresses that the error occurred in, with no information as to what error it was.

Given an error:

```
Assets\Scripts\ScriptBlock\myScript.cs(432,17): error CS1955:
Non-invocable member 'Transform.position' cannot be used like a method.
```

In this example it's further emphasized how important it is to understand the relevant terminology.

This error comes from an error returned by Unity. Not every programmer uses Unity, but enough of them understand the terminology provided that they can estimate a probable cause and possible offer a solution. Broken down:

- `Assets\Scripts\ScriptBlock\myScript.cs(432,17)` : The location of the issue in location/file(line, character) format.
- `error CS19455` : The error code, which may be directly searched for.
- `Non-invocable member: 'Transform.position' cannot be used like a method` : A description of the error, which states that a member, which may not be called, is being misused as a method when it isn't one.

In this specific example, all of this information was provided to us. This may not always be the case. Potentially, we could get an error such as `core dumped` or `ERROR: #F3220, #F32201, #FEFC33...`. These errors are, unfortunately, not very specific and don't leave us a lot to work with. If out of options, you may be inclined to share your problem with others who are more experienced. Doing this, however, will likely leave you waiting and halt your progress.

Knowing how to use your IDEs built-in features is a fundemental skill. Features such as *break-points* and *debug mode* allow developers to trace problems and view the state of their program at specific points in runtime in order to narrow down the exact moment their issue occurs.

Documenting Code

Documenting key elements of your code is important for yourself and potential future contributors when creating larger projects. It does not rely on standard naming conventions and relevant identifiers, however it is strongly encouraged to always title your data in an easy-tounderstand manner. By documenting your code, another tool for diagnosing problems is always readily available.

Code documenting, similar to translating complex language into simplified language, is entirely a preference. However, it may be cruicial to create code that is clear to the viewer:

```
function identifier()
```

Description: *description*
Return Type: *type*
Arguments: arg_namearg_type(range of expected values)
Handled Errors: *list of error titles*

Documentation in this manner should also be done for data, such as variables and objects. Official documentation of this type applied to a real project can be seen on the official Game Maker Language documentation.

Note Taking Reference

Short-hand representations of logical concepts designed to resemble pseudo-code and make the translation from concept to implementation easier. This chart is intended to be a loose guide for the sake of understanding how to write good notes, not a catch-all reference associate with a standard practice.

Concept	Linguistic Description
Weak-Type Variables	VAR: identifier
Strong-Type Variables	VAR: identifier TYPE: type
For Loop	FOR: iterator i; condition a; expression b

Concept	Linguistic Description
While Loop	WHILE: condition a
Do-While Loop	DO-WHILE: condition a
Switch Statement	SWITCH: condition a CASE-VALUES: range of case values RESULT: description of cases
If/Else Statements	IF condition a THEN: relevant code ELSE: fallback
Else If	IF: condition a THEN: relevant code ELSE IF: condition b THEN: relevant code
Weak-Type Function	METHOD: identifier() ARGS: arguments1, argument2, ... DESC: description of function
Strong-Type Function	METHOD: identifier() RET: type ARGS: argument1, argument2, ... DESC: description of function
Classes/Objects	OBJ: identifier
Events	EVENT: event type DESC: description of event

Important Note: Attaching descriptors, such as LOCAL, ASYNC or MEMBER are also useful for more unique elements of code.