Logical Problem Solving

This document serves as a short lesson on solving complex problems and the importance of utilizing key information provided to do so.

- 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
  - Conclusion

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

When it comes to turning language into code, implementation is easy. 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 that, just like a typical exam problem, the expression to create the evaluation would be up to you to determine).

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

Example in C++

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

Example in Python

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

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 like this:

```
"Functions are useful for completing simple tasks. For example, returning t
```

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
```

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
```

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 case, we got lucky; all of this information was provided to us. But what about an error where none of it is? What if, we get an error that just plainly states `core dumped` or `ERROR: #F32200, #F32201, #F32202, #C3B1302, #EFEC33`. You *may* find information online, but to do so likely requires your contextual issue. If your issue has not been asked about before, you will likely be required to inquire yourself. This means waiting for a response and halting your progress.

Conclusion

Hopefully this (very non-formal) document may emphasize a lesson many students often ignore early on in their studies, only to need later on in their career. A good programmer does not rely on *only* knowledge. Understanding and breaking down logical problems and troubleshooting properly are necessary skills to prevent being halted significantly in practice.