Type Systems

Type systems are essentially a “set of rules” built to maintain integrity in a programming language. They can be categorised by the way their outputs from a compiler/interpreter are designed to handle: a. try to fix the problem, b. stop and display an error.

Definitions

Weak typing is a term used to describe a system with data type assignment that can work outside typing restrictions. For example, a weakly typed programming language can allow users to perform arithmetic operations on a pointer type. Weakly typed systems would, for example, generalise operations of compatible data types such as, *int*, *float*, *double*, *boolean*, to a “number” arithmetic problem and allow the program to not get interrupted. A notable method of utilising weakly typed systems is through punning, or implicit type conversion. An example would be,

**JavaScript:**

**// concatenating 2 different data types**

**3 + ‘8’; // ‘38’**

**2 \* ‘3’; // 6**

**8 + true; // 9**

Strong typing, a contrast to weak typing, describes a system where data type assignment must strictly follow its rules. It is a system where unchecked run-time type errors must not exist [1]. In this system, type matters when performing operations, that means only type operations that are written in the rules can be performed. For example, a variable or method of *boolean* type cannot perform any operations with one of *int* type. There are explicit rules that determine only the same type can perform operations with one another.

A more well-defined typing system is dynamic typing. Dynamic typing operates on the premise that variable or method data type is determined during runtime, in other words type checking occurs during runtime, commonly by an interpreter. This means that during development, a variable or method can be created and assigned with any values that exist within the language itself, and, during execution, the system will determine the most appropriate data type for the variable or method. For example, in Python3, an *int* variable can be assigned as,

**Python3:**

**# Assigning integer 3 to variable age.**

**age = 3**

and during runtime the system will determine *int* type for variable “age”. Another characteristic of dynamic typing is that due to being bound by rules of weak and strong typing, one variable or method can be multiple data types at different parts of the execution, such as a *printf* statement being both a *string* and *int* value so that 3 + “years old” can be outputted as ‘3 years old’. With this benefit of flexibility, there come restrictions on how such values can be used and, when violated, a type error will be output.

The opposite of dynamic typing is static typing, which requires a variable or method to be explicitly declared a type before assignment. Type checking occurs not during runtime, but before, when the code is being compiled. An example would be:

**C Programming Language**

**‘’’ Assigning integer 3 to variable age ‘’’**

**int age = 3;**

Notice the explicit declaration of *int* type here during variable assignment. A tradeoff for the lack of flexibility comes with an edge in performance speed. This is because the type system does not have to deal with figuring out what data type should be assigned to the variable or method since it has already been declared by the programmer. The compiler can optimise your code knowing you are following the rules of the system.

A third, less discussed, type system is called “duck typing”. It can basically be summed up by a common phrase “If it looks like a duck and quacks like a duck, it's a duck” [2]. Duck typing can be characteristically like dynamic typing in which how you define your object is prioritised over declaring types. Moreover, much like weak typing, there aren’t as strict rules on typing as strong typing is.

In Object-Oriented Programming (OOP) languages, there is a term called polymorphism which describes a method of a class having more than one form, that line of thought is necessary to understand duck typing. Example,

**Python3:**

**class India():**

**def capital(self):**

**print("New Delhi is the capital of India.")**

**class USA():**

**def capital(self):**

**print("Washington, D.C. is the capital of USA.")**

**obj\_ind = India()**

**obj\_usa = USA()**

**for country in (obj\_ind, obj\_usa):** **# notice that only country.capital() is called**

**country.capital()** **# but it outputs 2 unique values**

**Output:**

**New Delhi is the capital of India.**

**Washington, D.C. is the capital of USA.**

Notice that country.capital() just calls for a feature “capital”, disregarding any specific function names. One characteristic of duck typing, let’s say with an array data type, is that any object with array-like operators such as indexing can be used in any code in place of an array [2].

Many developers within the programming community debate about the type system of C, but in truth, it exhibits properties of both (however mostly siding weak typing). Going back to the definition of strongly typed languages, the rules of operation in C is strict by nature, meaning that if a variable or function that was to be added is a whole number, then the others in the operation must be specifically of the same time (*int*, *float*, etc.),

**C Programming Language**

**// if operation is a + b, notice that both a and b are *int* data type**

**int a = 3, b = 4;**

**printf(a+b);**

**// Output -> 7**

**// if a and b are different number types**

**int a = 3;**

**float b = 4.0;**

**// Error will be displayed if they were added together because *int* != *float***

This may give an illusion that C is determinedly a strongly typed language. It also has the stronger argument for C being a weakly typed language stems from its ability to: a) cast pointers, b) implicit type conversion. A quick example would be,

**C Programming Language**

**int x;**

**char y;**

**x = y;**

**// the program would continue running without displaying an error. This proves**  **// difficult to debug later should there be a bug.**

**Or**

**long x = 10;**

**int y = x;**

**printf("x=%ld\ny=%d\n",x,y);**

**// Output -> x=10**

**y=10**

Notice that although there is explicit type declaration, the programs in these 2 codes seem to run successfully. This is called implicit type conversion. A second characteristic of weakly typed C is casting, which is basically a technical term for explicit conversion. Functions like **atoi()**, **atof()** - string to integer and string to float, respectively - are examples of casting. The difference between casting and implicit conversion is that for casting, the conversion is commanded by the programmer during code design, and implicit conversion is done by the compiler’s intelligence. Casting also enables conversion between two incompatible data types, such as *int* to *byte* via **(byte)** function [4].

Personally, I am biased towards interpreter scenarios such as executing code during runtimes so my preference heavily sides with dynamic typing. Explicit type declaration is not inconvenient, so I am not nitpicky about strong and weak typing. If I had a choice, I would work with data visualisation and data science at the present time. Here, dynamically typed languages such as Python and R would meet the requirements, firstly based on their library functions and capabilities for the field, but secondly because I am not building robust software, rather I am running lines of code to aggregate data and achieve a product quickly. This scenario does not require optimised and high-performance code, because the code is being run to achieve an outcome immediately. The great thing about a dynamically typed language like Python is that as the interpreter runs the code during execution, it allows me to debug which line had an error because it would run until it finds an error and would interrupt at that line. A primary drawback of this approach would be that if I were to be given a task to work with prediction models in neural networks, the performance of my code would be severely hindered by the dynamic nature of my programming language of choice. A better option, in that case, would be C++, which is a statically typed language. The benefit of this would be optimised performance by the compiler, assuming that the code is running bug-free. One major disadvantage, though, is that coding takes a very long time due to the compiler’s debugging nature.

References

1. Luca Cardelli, "Typeful programming", from <http://www.lucacardelli.name/Papers/TypefulProg.pdf>>
2. N. Milojkovic, M. Ghafari and O. Nierstrasz, "It's Duck (Typing) Season!," *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 312-315, doi: 10.1109/ICPC.2017.10.
3. Eckel B. (2005) Strong Typing vs. Strong Testing. In: The Best Software Writing I. Apress. <https://doi-org.libproxy.library.wmich.edu/10.1007/978-1-4302-0038-3_11>
4. Type Casting in C Language - Implicit and Explicit Types of Type casting. (2019). Retrieved 21 October 2021, from <https://www.improgrammer.net/type-casting-c-language/#:~:text=0-,Type%20Casting%20in%20C%20Language,explicitly%20using%20the%20cast%20operator>.