CSC 432
Lecture Note 3_6

Programming Concepts Series

Compiled and Interpreted Languages

In programming, we depend on tools such as compilation and interpretation in order to get our written code into a form that the computer can execute. Code can either be executed natively through the operating system after it is converted to *machine code* (via compilation) or can be evaluated line by line through another program which handles executing the code instead of the operating system itself (via interpretation).

A **compiled language** is one where the program, once compiled, is expressed in the instructions of the target machine; this machine code is undecipherable by humans. An **interpreted language** is one where the instructions are not directly executed by the target machine, but instead read and executed by some other program (which normally *is* written in the language of the native machine). Both compilation and interpretation offer benefits and pitfalls, which is mainly what we're going to talk about.

Interpreted Compiled Languages

The major advantage of compiled languages over interpreted languages is their execution speed. Because compiled languages are converted directly into machine code, they run significantly faster and more efficiently than interpreted languages, especially considering the complexity of statements in some of the more modern scripting languages which are interpreted.

Lower-level languages tend to be compiled because efficiency is usually more of a concern than cross-platform support. Additionally, because compiled languages are converted directly into machine code, this gives the developer much more control over hardware aspects such as memory management and CPU usage. Examples of pure compiled languages include C, C++, Erlang, Haskell, and more modern languages such as Rust and Go.

Some of the pitfalls of compiled languages are pretty substantial however. In order to run a program written in a compiled language, you need to first manually compile it. Not only is this an extra step in order to run a program, but while you debug the program, you would need to recompile the program each time you want to test your new changes. That can make debugging very tedious. Another detriment of compiled languages is that they are not platform-independent, as the compiled machine code is specific to the machine that is executing it.

Interpreting Languages

In contrast to compiled languages, interpreted languages do not require machine code in order to execute the program; instead, interpreters will run through a program line by line and execute each command. In the early days of interpretation, this posed a disadvantage compared to compiled languages because it took significantly more time to execute the program, but with the advent of new technologies such as just-in-time compilation, this gap is narrowing. Examples of some common interpreted languages include PHP, Perl, Ruby, and Python. Some of the programming concepts that interpreted languages make easier are:

- Platform independence
- Reflection
- Dynamic typing
- Smaller executable program size
- Dynamic scoping

The main disadvantage of interpreted languages is a slower program execution speed compared to compiled languages. However, as mentioned earlier, just-in-time compilation helps by converting frequently executed sequences of interpreted instruction into host machine code.

Bytecode Languages

Bytecode languages are a type of programming language that fall under the categories of both compiled and interpreted languages because they employ both compilation and interpretation to execute code. Java and the .Net framework are easily the most common examples of bytecode languages (dubbed **Common Intermediate Language** in .Net). In fact, the Java Virtual Machine (JVM) is such a common virtual machine to interpret bytecode that several languages have implementations built to run on the JVM.

In a bytecode language, the first step is to compile the current program from its human-readable language into bytecode. **Bytecode** is a form of instruction set that is designed to be efficiently executed by an interpreter and is composed of compact numeric codes, constants, and memory references. From this point, the bytecode is passed to a virtual machine which acts as the interpreter, which then proceeds to interpret the code as a standard interpreter would.

In bytecode languages, there is a delay when the program is first run in order to compile the code into bytecode, but the execution speed is increased considerably compared to standard interpreted languages because the bytecode is optimized for the interpreter. The largest benefit of bytecode languages is platform independence which is typically only available to interpreted languages, but the programs have a much faster execution speed than interpreted languages. Similar to how interpreted languages make use of just-in-time compilation, the virtual machines that interpret bytecode can also make use of this technique to enhance execution speed.

Concepts of Type Checking

When learning about programming languages, you've probably heard phrases like *statically-typed* or *dynamically-typed* when referring to a specific language. These terms describe the action of **type checking**, and both static type checking and dynamic type checking refer to two different **type systems**. A type system is a collection of rules that assign a property called type to various constructs in a computer program, such as variables, expressions, functions or modules, with the end goal of reducing the number of bugs by verifying that data is represented properly throughout a program.

A Type

A **type**, also known as a data type, is a classification identifying one of various types of data. I hate to use the word type in its own definition, so in a nutshell a type describes the possible values of a structure (such as a variable), the semantic meaning of that structure, and how the values of that structure can be stored in memory. If this sounds confusing, just think about Integers, Strings, Floats, and Booleans – those are all types. Types can be broken down into categories:

- **Primitive types** – these range based on language, but some common primitive types are integers, booleans, floats, and characters.
- **Composite types** – these are composed of more than one primitive type, e.g. an array or record (not a hash, however). All composite types are considered **data structures**.
- **Abstract types** – types that do not have a specific implementation (and thus can be represented via multiple types), such as a hash, set, queue, and stack.
- **Other types** – such as **pointers** (a type which holds as its value a reference to a different memory location) and functions.

Certain languages offer built-in support for different primitive types or data structures than other languages, but the concepts are the same. A type merely defines a set of rules and protocols behind how a piece of data is supposed to behave.

Type Checking

The existence of types is useless without a process of verifying that those types make logical sense in the program so that the program can be executed successfully. This is where type checking comes in. **Type checking** is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically). Type checking is all about ensuring that the program is **type-safe**, meaning that the possibility of type errors is kept to a minimum. A type error is an erroneous program behavior in which an operation occurs (or trys to occur) on a particular data type that it’s not meant to occur on. This could be a situation where an operation is performed on an integer with the intent that it is a float, or even something such as adding a string and an integer together:

```
x = 1 + "2"
```

While in many languages both strings and integers can make use of the + operator, this would often result in a type error because this expression is usually not meant to handle multiple data types.

When a program is considered not type-safe, there is no single standard course of action that happens upon reaching a type error. Many programming languages throw type errors which halt the runtime or compilation of the program, while other languages have built-in safety features to handle a type error and continue running (allowing developers to exhibit poor type safety). Regardless of the aftermath, the process of type checking is a necessity.

Now that we have a basic understanding of what types are and how type checking works, we can start getting into the two primary methods of type checking: **static type checking** and **dynamic type checking**.

Static Type Checking

A language is statically-typed if the type of a variable is known at **compile time** instead of at runtime. Common examples of statically-typed languages include Ada, C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal, and Scala.

The big benefit of static type checking is that it allows many type errors to be caught early in the development cycle. Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code (i.e. faster and/or using less memory). Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed.

A static type-checker will quickly detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. However, a detriment to this is that static type-checkers make it nearly impossible to manually raise a type error in your code because even if that code block hardly gets called – the type-checker would almost always find a situation to raise that type error and thus would prevent you from executing your program (because a type error was raised).

Dynamic Type Checking

Dynamic type checking is the process of verifying the type safety of a program at **runtime**. Common dynamically-typed languages include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.

Most type-safe languages include some form of dynamic type checking, even if they also have a static type checker. The reason for this is that many useful features or properties are difficult or impossible to verify statically. For example, suppose that a program defines two types, A and B, where B is a subtype of A. If the program tries to convert a value of type A to type B, which is known as **downcasting**, then the operation is legal only if the value being converted is actually a value of type B. Therefore, a dynamic check is needed to verify that the operation is safe. Other language features that dynamic-typing enable include **dynamic dispatch**, **late binding**, and **reflection**.

In contrast to static type checking, dynamic type checking may cause a program to fail at runtime due to type errors. In some programming languages, it is possible to anticipate and recover from these failures – either by error handling or poor type safety. In others, type checking errors are considered fatal. Because type errors are more difficult to determine in dynamic type checking, it is a common practice to supplement development in these languages with **unit testing**.

All in all, dynamic type checking typically results in less optimized code than does static type checking; it also includes the possibility of runtime type errors and forces runtime checks to occur for every execution of the program (instead of just at compile-time). However, it opens up the doors for more powerful language features and makes certain other development practices significantly easier. For example, **metaprogramming** – especially when using ***eval*** **functions** – is not impossible in statically-typed languages, but it is much, much easier to work with in dynamically-typed languages.

Common Misconceptions

A common misconception is to assume that all statically-typed languages are also strongly-typed languages, and that dynamically-typed languages are also weakly-typed languages. This isn't true, and here's why:

A **strongly-typed language** is one in which variables are bound to specific data types, and will result in type errors if types to not match up as expected in the expression – regardless of when type checking occurs. A simple way to think of strongly-typed languages is to consider them to have high degrees of type safety. To give an example, in the following code block repeated from above, a strongly-typed language would result in an explicit type error which ends the program's execution, thus forcing the developer to fix the bug: x = 1 + "2"

We often associate statically-typed languages such as Java and C# as strongly-typed (which they are) because data types are explicitly defined when initializing a variable – such as the following example in Java:

Java

```
// Java
String foo = new String("hello world");
```

However, ruby, python, and javascript (all of which are dynamically-typed) are also strongly-typed languages and the developer makes no verbose statement of data type when declaring a variable. Below is the same java example above, but written in ruby.

Ruby

```
# Ruby
foo = "hello world"
```

Both of the languages in these examples are strongly-typed, but employ different type checking methods. Languages such as ruby, python, and javascript which do not require manually defining a type when declaring a variable make use of **type inference** – the ability to programmatically infer the type of a variable based on its value. Some programmers automatically use the term weakly typed to refer to languages that make use of type inference, often without realizing that the type information is present but implicit. Type inference is a separate feature of a language that is unrelated to any of its type systems.
A **weakly-typed language** on the other hand is a language in which variables are not bound to a specific data type; they still have a type, but type safety constraints are lower compared to strongly-typed languages. Take the following PHP code for example:

PHP

```
// PHP
$foo = "x";
$foo = $foo + 2; // not an error
echo $foo;       // 2
```

Because PHP is weakly-typed, this would not error. Just as the assumption that all strongly-typed languages are statically-typed, not all weakly-typed languages are dynamically-typed; PHP is a dynamically-typed language, but C – also a weakly-typed language – is indeed statically-typed.

It is true that most statically-typed languages are usually compiled when executed, and most dynamically-typed languages are interpreted when executed – but you can't always assume that, and there's a simple

reason for this:
When we say that a language is statically- or dynamically-typed, we are referring to that **language as a whole**. For example, no matter what version of Java you use – it will always be statically-typed. This is different from whether a language is compiled or interpreted, because in that statement we are referring to a **specific language implementation**. Thus in theory, any language can be compiled or interpreted. The most common implementation of Java is to compile to bytecode, and have the JVM interpret that bytecode – but there are other implementations of Java that compile directly to machine code or that just interpret Java code as is.

Garbage Collection

At its core, GC is a process of automated memory management so that you as a developer have one less thing to worry about. When you allocate memory – like by creating a variable – that memory is allocated to either the stack or the heap (check out my post on **the stack vs. the heap** if you want to learn more about these two). You allocate to the stack when you're defining things in a local scope where you know exactly the memory block size you need, such as primitive data types, arrays of a set size, etc. The stack is a self-managing memory store that you don't have to worry about – it's super fast at allocating and clearing memory all by itself. For other memory allocations, such as objects, buffers, strings, or global variables, you allocate to the heap.

Compared to the stack, the heap is not self-managing. Memory allocated to the heap will sit there throughout the duration of the program and can change state at any point in time as you manually allocate/deallocate to it. The garbage collector is a tool that removes the burden of manually managing the heap. Most modern languages such as Java, the .NET framework, Python, Ruby, Go, etc. are all garbage collected languages; C and C++, however, are not – and in languages such as these, manual memory management by the developer is an extremely important concern.

Why Do We Need GC

GC helps save the developer from several memory-related issues – the foremost being **memory leaks**. As you allocate more and more memory to the heap, if the program doesn't consistently release this memory as it becomes unneeded, memory size will begin to add up – resulting in a **heap overflow**. Even if heap memory is diligently managed by the developer – all it takes is one variable to be consistently left undeleted to result in a memory leak, which is bad.

Even if there are no memory leaks, what happens if you are attempting to reference a memory location which has already been deleted or reallocated? This is called a **dangling pointer**; the best case scenario here is that you would get back gibberish, and hopefully throw or cause a validation error soon after when that variable is used – but there’s nothing stopping that memory location from being overwritten with new data which could respond with seemingly valid (but logically incorrect) data. You’d have no idea what would be going on, and it’s these types of errors – memory errors – that are often times the most difficult to debug.

That’s why we need GC. It helps with all of this. It’s not perfect – it does use up extra resources on your machine to work and it’s normally not as efficient as proper manual memory management – but the problems it saves you from make it more than worth it.

How and When does the Garbage Collector Run?

This depends entirely on the algorithm used for GC. There isn’t one hard and fast way to do it, and just like compilers and interpreters, GC mechanisms get better over time. Sometimes the garbage collector will run at pre-determined time intervals, and sometimes it waits for certain conditions to arise before it will run. The garbage collector will just about always run on a separate thread in tandem with your program – and depending on the language’s implementation of GC, it can either stall your program (i.e. stop-the-world GC) to sweep out all the garbage at once, run incrementally to remove small batches, or run concurrently with your program.

It’s difficult to get deeper than this without getting into specific languages’ implementations of GC, so let’s move onto the common GC algorithms.

Garbage Collection Algorithms

There’s a bunch of different GC algorithms out there – but here are some of the most common ones you’ll come across. It’s interesting to note how many of these common algorithms build on one another.

Reference Counting

Reference counting is perhaps the most basic form of GC, and the easiest to implement on your own. The way it works is that anytime you reference a memory location on the heap, a counter for that particular memory location increments by 1. Every time a reference to that location is deleted, the counter decrements by 1. When that counter gets to 0, then that particular memory location is garbage collected.

One of the big benefits of GC by reference counting is that it can immediately tell if there is garbage (when a counter hits 0). However, there are some major problems with reference counting; circular references just flat out can’t be garbage collected – meaning that if object A has a reference to object B, and object B has a reference back to object A, then neither of these objects can ever be garbage collected according to reference counting. On top of this, reference counting is very inefficient because of the constant writes to the counters for each memory location.

Because of these problems, other algorithms (or at least refined versions of reference counting) are more commonly used in modern GC.

Mark-Sweep

Mark-sweep – as well as just about all modern GC algorithms other than reference counting – is a form of a **tracing** GC algorithm, which involves *tracing* which objects are reachable from one or multiple "roots" in order to find unreachable (and thus unused) memory locations. Unlike reference counting, this form of GC is not constantly checking and it can theoretically run at any point in time.

The most basic form of mark-sweep is the **naïve mark-sweep**; it works by using a special bit on each allocated memory block that's specifically for GC, and running through all memory currently allocated on the heap twice: the first time to **mark** locations of dead memory via that special bit, and the second time to **sweep** (i.e. deallocate) those memory locations.

Mark-sweep is more efficient than reference counting because it doesn't need to keep track of counters; it also solves the issue of not being able to remove circularly referenced memory locations. However, naïve mark-sweep is a prime example of stop-the-world GC because the entire program must be suspended while it's running (non-naïve tracing algorithms can run incrementally or concurrently). Because tracing GC can happen at any point in time, you don't ever have a good idea of when one of these stalls will happen. Heap memory is also iterated over twice – which slows down your program even more. On top of that, in mark-sweep there's no handling of fragmented memory; to give you a visual representation of this, imagine drawing a full grid representing all of your heap memory – mark-sweep GC would make that grid look like a very bad game of Tetris. This fragmentation almost always leads to less efficient allocation of memory on the heap. So – we continue to optimize our algorithms.

Mark-Compact

Mark-compact algorithms take the logic from mark-sweep and add on at least one more iteration over the marked memory region in an effort to *compact* them – thus defragmenting them. This address the fragmentation caused by mark-sweep, which leads to significantly more efficient future allocations via the use of a "bump" allocator (similar to how a stack works), but adds on extra time and processing while GC is running because of the extra iteration(s).

Copying

Copying (also known as **Cheney's Algorithm**) is slightly similar to mark-compact, but instead of iterating potentially multiple times over a single memory region, you instead just copy the "surviving" memory blocks of the region into an entirely new empty region after the mark phase – which thus compacts them by default. After the copying is completed, the old memory region is deallocated, and all existing references to surviving memory will point to the new memory region. This relieves the GC of a lot of processing, and brings down the specs to something even quicker than a mark-sweep process since the sweep phase is eliminated.

While you’ve increased speed though, you now have an extra requirement of needing an entirely available region of memory that is at least as large as the size of all surviving memory blocks. Additionally, if most of your initial memory region includes surviving memory, then you’ll be copying a lot of data – which is inefficient. This is where GC *tuning* becomes important.

Generational

Generational GC takes concepts from copying algorithms, but instead of copying all surviving members to a new memory region, it instead splits up memory into *generational* regions based on how old the memory is. The rationale behind generational GC is that normally, young memory is garbage collected much more frequently than older memory – so therefore the younger memory region is scanned to check for unreferenced memory much more frequently than older memory regions. If done properly, this saves both time and CPU processing because the goal is to scan only the necessary memory.

Older memory regions are certainly still scanned – but not as often as younger memory regions. If a block of memory in a younger memory region continues to survive, then it can be promoted to an older memory region and will be scanned less often.