UNIT-6

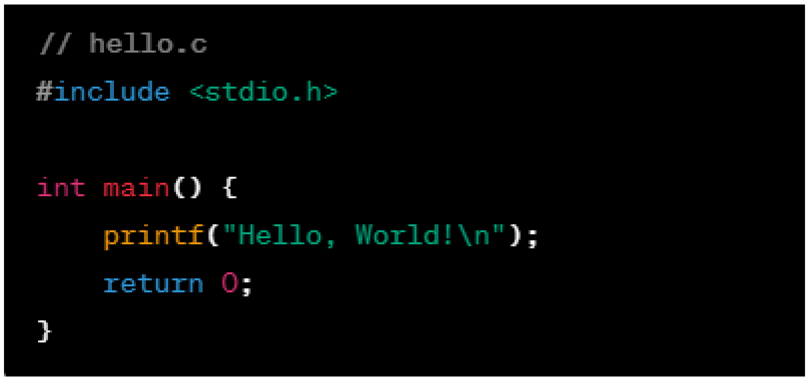
Interpreter & Debuggers

Compiler:

A compiler is a program that translates the entire source code of a program written in a high-level programming language into machine code or an intermediate code. This translation process is called compilation, and the result is an executable file that can be run independently of the original source code.

Example-

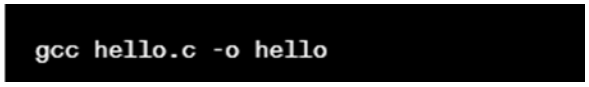
Let's say you have a program written in the C programming and the source code is in a file named hello.c. you use a C compiler GCC to compile it-



```c
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

In the command line, you will run-



```
gcc hello.c -o hello
```

This command tells the compiler GCC to compile hello.c and create an executable named hello. Now, we can run hello executable and it will print hello to the console.

Interpreter:

An interpreter, on the other hand, processes the source code of a program line by line at runtime. Instead of generating a separate executable, the interpreter directly executes the high-level code. It reads and executes the source code one statement at a time.

Example:

Consider a simple Python script named hello.py

```
# hello.py

print("Hello, World!")
```

To run this script, you don't need to compile it. You use the Python interpreter:

```
python hello.py
```

The Python interpreter reads each line of the script and executes it on the fly (means doing something immediately as needed, without any delay or pre-processing.), printing "Hello, World!" to the console.

In summary, a compiler's main role is to translate source code into machine code or an intermediate code, but it does not execute the code itself. Execution is typically handled by a separate program or environment.

Debugger-A debugger is a tool used by software developers to find and fix errors, or "bugs," in their code. It allows developers to inspect the state of a running program, track the flow of execution, and identify issues that may be causing the program to behave unexpectedly.

Let's go through a simple example to illustrate how a debugger works:

Example: Python Code with a Bug

Suppose you have the following Python code that is intended to calculate the sum of numbers from 1 to 10:

```
def calculate_sum():

sum_result = 0

    for i in range(1, 10):

sum_result += i

    return sum_result

result = calculate_sum()

print("The sum is:", result)
```

However, there's a bug in the code. Instead of calculating the sum from 1 to 10, it is currently

calculating the sum from 1 to 9 and the result is 45. Let's use a debugger to find and fix the issue.

Using a Debugger:

Set a Breakpoint: In your code, you can set a breakpoint at a specific line where you want the program to pause. In this case, let's set a breakpoint at i==5.

```
def calculate_sum():

sum_result = 0

    for i in range(1, 10):

        # Set a breakpoint here (the condition ensures it stops on i == 5)

        if i == 5:

breakpoint()

sum_result += i

    return sum_result

result = calculate_sum()

print("The sum is:", result)
```

In this example, the breakpoint() function is called when i equals 5. The breakpoint() function is a built-in Python function that triggers the debugger. When you run your code with a debugger, it should pause at the breakpoint() line when i is 5, allowing you to inspect the current state of the program. Depending on the result, you can change the value of I to analyze the code.

you can use the command python -m pdb your_script.py to run your script with the Python debugger.

How range function behaves in python(why do it calculates only 9 values when I iterates through 1 to 10)?

In Python, the range function generates a sequence of numbers up to, but not including, the specified end value. Therefore, when you use range(1, 11), it generates values from 1 to 10, inclusive. The loop iterates over these values and calculates the sum.

So, The variable i takes on values 1, 2, 3, ..., 9 in the above code. The loop iterates over these 9 values, and the sum is calculated accordingly. If we want to get sum of 1 to 10 values, we will change the range i.e. (1, 11).

Benefits of Interpretation-

An interpreter is a type of system software that translates and executes code line by line, rather than translating the entire program before execution, like a compiler. Here are some benefits of using an interpreter in simple terms, along with an example:

Easy Debugging:

- With an interpreter, you can identify and fix errors in your code one step at a time.
- **Example**: Suppose you have a program with 100 lines of code. If there's an error on line 25, the interpreter will stop at that line, making it easier to find and correct the mistake.

Platform Independence:

- Interpreters allow programs to be written in a high-level language that can run on different platforms without modification.
- **Example:** A Python program written on a Windows computer can be executed on a Mac or Linux machine using a Python interpreter without needing to change the code.

Quick Development:

- Since interpreters execute code line by line, you can see the results immediately, making the development process faster.
- **Example:** If you're writing a script to automate a task, you can test and see the results of each command as you go along.

Flexibility:

- Interpreters provide flexibility in terms of dynamic typing and late binding, allowing for changes during runtime.
- **Example:** In languages like JavaScript, you can add or modify functions and variables on the fly, making it adaptable to different situations.

Memory Efficiency:

- Interpreters typically consume less memory than compilers since they don't produce an intermediate machine code or executable file.
- **Example:** Running a Python script may use less memory compared to compiling a similar program written in a language like C++.

JAVA Language Environment-

JAVA Virtual Machine-The Java Virtual Machine (JVM) plays a crucial role in the execution of Java programs. It serves as both a compiler and an interpreter, and its role is significant from a system software point of view.

1. Compiler:

- **Just-In-Time (JIT) Compilation:** When we compile a Java source code file, it is not translated directly into machine code for the underlying hardware. Instead, it is translated into an intermediate form called byte code. This byte code is a set of instructions that is independent of the hardware and platform.
- **JVM as a Compiler:** The JVM includes a Just-In-Time compiler that takes the byte code and translates it into native machine code at runtime. This process is known as JIT compilation. The JVM monitors the execution of the program and identifies frequently executed code segments. It then compiles these segments into native machine code, optimizing the performance of the program (identifying hotspots (means identifying code segments i.e. loops, methods etc. which are frequently executed), in lining means replacing the actual code of a method or function where it is called just like macro, loop unrolling, constant folding etc.).

2. Interpreter:

- **Bytecode Interpretation:** Before JIT compilation, the JVM interprets the bytecode directly. The bytecode is executed line by line by the JVM's interpreter. This interpretation allows Java programs to be platform-independent, as the bytecode can run on any system with a compatible JVM.
- **Interpretation vs. Compilation:** Interpretation is generally slower than native machine code execution. However, interpretation provides the advantage of portability, as the same bytecode can be executed on any system with a JVM. The combination of interpretation and JIT compilation in the JVM aims to strike a
- balance between portability and performance.

3. Execution Process:

- **Loading:** The JVM loads the bytecode produced by the Java compiler. This bytecode is stored in .class files.
- **Verification:** Before execution, the JVM verifies the bytecode to ensure it adheres to certain security and structural constraints. This helps in preventing security vulnerabilities (While Java's design principles and bytecode verification contribute to a more secure environment, it doesn't mean that writing malicious code in Java is impossible. The security is dynamic thing, and vulnerabilities may be discovered over time. Regular updates and patches from Java providers are crucial for maintaining a secure environment.).
- **Execution:** The JVM interprets the bytecode initially. As the program runs, the JIT compiler identifies hotspots (frequently executed code) and compiles them into native machine code for better performance.

- **Memory Management:** The JVM manages memory, including garbage collection to automatically reclaim memory occupied by objects that are no longer needed.

4.Portability and Abstraction:

- **Platform Independence:** Java's "Write Once, Run Anywhere" principle is made possible by the JVM. The bytecode, being an intermediate representation, abstracts the underlying hardware and operating system, allowing Java programs to run on diverse platforms without modification.
- **System Interface:** The JVM provides a bridge between the Java program and the underlying system. It abstracts system-specific details, making it easier to develop cross-platform applications.

How do Interpreter and JIT Compiler work together?

1. Bytecode Interpretation:

When: Initially, when a Java program is executed, the Java Virtual Machine (JVM interpreter) starts by interpreting the bytecode.

What Happens: The JVM reads the bytecode (compiled from the Java source code) line by line and executes the corresponding instructions.

Why: Interpretation allows Java programs to be platform-independent. The same bytecode can be executed on any system with a compatible JVM, as long as the bytecode conforms to the Java Virtual Machine Specification.

2.Just-In-Time (JIT) Compilation:

When: As the program continues to run, the JVM monitors the execution of the bytecode and identifies frequently executed code segments, known as "hotspots."

What Happens: The JVM employs a Just-In-Time compiler to translate these hotspots from bytecode into native machine code for the specific hardware architecture. This compilation occurs dynamically during the program's execution.

Why: JIT compilation aims to improve the overall performance of the program by providing a more efficient form of code execution compared to bytecode interpretation.

3. Combined Execution:

Hybrid Approach: The JVM uses a hybrid approach that combines both interpretation and compilation.

Interpretation vs. Compilation: Initially, less frequently executed parts of the code are interpreted to achieve platform independence. However, frequently executed portions are identified and compiled into native machine code for better performance.

Adaptive Compilation: The process is adaptive, meaning that the JVM may adjust its compilation strategy based on the changing runtime behavior of the program.

Note-Once hotspots are identified and compiled into native machine code by the Just-In-Time (JIT) compiler, the interpreter doesn't continue to interpret those specific hotspots. Instead, the JVM switches to using the compiled machine code for the

```
public class Example {

    public static void main(String[] args) {

        int result = 0;



        for (int i = 0; i< 1000000; i++) {

            result += square(i);

        }



System.out.println("Result: " + result);

    }

    public static int square(int x) {

        return x * x;

    }

}
```

In this example, the main method contains a loop that iterates a million times, calling the square method in each iteration to calculate the square of the loop variable i. The square method is a simple function that squares its input.

Now, let's walk through the execution process:

1. Interpretation Phase:

- Initially, when the program starts running, the JVM uses the interpreter to execute the bytecode of the entire program.
- The loop in the main method is interpreted, and the square method is called in each iteration.

2. Hotspot Identification:

- As the loop executes, the JVM monitors the program and identifies the square method as a hotspot because it is being frequently executed in the loop.

3. JIT Compilation Phase:

- The identified hotspot, the square method, is then subjected to JIT compilation.
- The JIT compiler translates the bytecode of the square method into native machine code.

4. Switch to Compiled Code:

- After JIT compilation is completed for the square method, the JVM switches to using the compiled machine code for subsequent calls to the square method within the loop.
- The loop's performance is now improved, as the repeated calls to the square method are executed using the more efficient compiled machine code.

5. Continued Interpretation:

- Other parts of the program, such as the loop in the main method and the method calls outside the loop, may continue to be interpreted.

6. Adaptive Compilation:

- The process is adaptive, meaning that the JVM may dynamically adjust its compilation strategy based on the changing runtime behavior of the program. If other parts of the code become hotspots during execution, they may be compiled as well.

Debugging Procedures

- Whenever there is a gap between an expected output and an actual output of a program, the program needs to be debugged.
- An error in a program is called bug, and debugging means finding and removing the errors in a program.
- Debugging involves executing the program in a controlled fashion.
- During debugging, the execution of a program can be monitored at every step.
- In the debug mode activities i.e. starting the execution and stopping the execution are done by the debugger.

- The debugger provides the facility to execute a program up to the specified instruction by inserting a breakpoint.
- It gives a chance to examine the value assigned to the variables in the program at any instant and, if required, offers an opportunity to update the program.

Types of debugging procedures:

Debug Monitors:

Debug monitors are software tools or components that assist in debugging by providing real-time information about the program's execution. These monitors often run alongside the program being debugged, allowing developers to inspect the program's state, memory, and other relevant information. They can help identify issues such as memory leaks, variable values, and the flow of execution.

Example:

Let's say you are developing a complex algorithm, and you suspect there is an issue with the values of variables at a certain point in the code. You can use a debug monitor to set breakpoints or watch points at specific locations in your code. When the program reaches these points, the monitor can pause execution and allow you to inspect the variables' values, helping you identify the source of the problem.

Assertions:

Assertions are statements or conditions embedded in the code that developers believe should always be true. They act as checks at runtime, helping to catch issues early in the development process. If an assertion fails, it indicates a logical error in the program, and developers can use this information to trace and fix the problem.

Example:

```
def divide(x, y):
assert y != 0, "Division by zero is not allowed"
result = x / y
return result
```

In this Python example, an assertion is used to check if the denominator y is not zero before performing the division operation. If the assertion fails (i.e., if y is zero), an error message is displayed, helping the developer identify the problem. The purpose of this assertion is to catch a potential issue during runtime: division by zero. Division by zero is undefined in mathematics, and in many programming languages, attempting to divide by zero will result in an error. By using an assertion, the developer is explicitly stating a condition that they believe should always be true, and if it turns out not to be true (i.e., y is zero), it indicates a logical error in the code.

Debug monitors and assertions are complementary tools, and both play crucial roles in the debugging process. Debug monitors provide a dynamic view of the program's execution, allowing developers to inspect runtime information, while assertions help developers express and enforce assumptions about the correctness of the code. Used together, these tools make the debugging process more efficient and effective.

Classification of Debuggers-

Static Debuggers-

- Static debugging in system software refers to the process of identifying and fixing programming errors, anomalies, or potential issues in a program without executing it. Unlike dynamic debugging, which involves analyzing the program while it is running, static debugging is performed on the program's source code without actually running the program.
- Static debugging focuses on semantic analysis.
- In a program, suppose there are two variables: var1 and var2. The type of var1 is an integer, and the type of var2 is a float. Now, the program assigns the value of var2 to var1; then, there is a possibility that it may not get correctly assigned to the variable due to truncation. This type of analysis falls under static debugging
- Static code analysis may include detection of the following situations:
    - ➢ Dereferencing of variable before assigning a value to it
    - ➢ Truncation of value due to wrong assignment
    - ➢ Redeclaration of variables
    - ➢ Presence of unreachablecode

Example-

```
#include <stdio.h>
int main() {
int x = 5;
if (x > 10)
{
printf("This block will never be executed because x is not greater than 10.\n");
      return 1;  // This return statement ends the function
   }
else
{
  printf("This block will always be executed.\n");
   }
```

```
        return 0;
// Unreachable code
   printf("This statement will never be executed because it comes after a return statement.\n");
```

The return 0, statement ends the main function, and any code after it is considered unreachable. Therefore, the printf statement after the return 0, is unreachable because the program will never reach that point during its execution.

Benefits of Static Debuggers:

- Early Issue Detection: Static debuggers catch issues before runtime, allowing developers to fix problems early in the development process.
- Code Quality Improvement: By identifying coding standards violations and potential bugs, static debuggers contribute to overall code quality.
- Security Assurance: Tools like static code analyzers can identify security vulnerabilities, helping to create more secure software.

While static debuggers are valuable, they are not a substitute for dynamic debuggers, which analyze the code during runtime. Both types of debuggers complement each other to provide a comprehensive approach to software debugging and analysis.

Dynamic/Interactive Debugger

A dynamic or interactive debugger is a software tool designed to help programmers identify and resolve issues in their code during the development and debugging process. Unlike static analysis tools that analyze code without executing it, dynamic debuggers operate on the running program, allowing developers to inspect its state, control its execution, and diagnose problems in real-time.

Here are key features and concepts associated with dynamic/interactive debuggers:

Breakpoints:

Developers can set breakpoints at specific lines of code or conditions. When the program reaches a breakpoint, it pauses its execution, allowing the developer to inspect variables, memory,etc.

Stepping:

Debuggers enable developers to step through the code one line at a time. This includes stepping into functions (entering a function and debugging it), stepping over (executing a function without diving into it), and stepping out (exiting the current function and returning to the calling code).

Variable Inspection:

Developers can inspect the values of variables at different points in the program's execution. This is crucial for understanding how data changes during runtime and identifying the source of bugs.

Call Stack:

The call stack shows the sequence of function calls leading to the current point in the program. It helps developers understand the flow of execution and can be used to trace the origin of errors.

Interactive Console:

Some debuggers provide an interactive console where developers can enter commands during program execution. This can be handy for quick evaluations, modifications, or executing specific functions.

Conditional Breakpoints:

Developers can set breakpoints that only trigger when a specified condition is met. This allows for more targeted debugging based on specific circumstances.

Memory Inspection:

Debuggers often provide tools to inspect the program's memory, helping developers identify issues such as memory leaks, buffer overflows, or other memory-related problems.

Exception Handling:

Debuggers can catch and pause on exceptions (runtime errors) in the code, allowing developers to inspect the state of the program when an error occurs.

Remote Debugging:

Some debuggers support remote debugging, allowing developers to debug a program running on a different machine or device.