CA341 - Comparing Procedural and Object-Oriented Programming

Group 40: Benjamin Olojo (19500599) and Przemyslaw Majda (20505049)

Introduction

This assignment consisted of implementing a phonebook program in both a procedural and object-oriented language using Binary Trees for the insertion, deletion and searching of contacts inputted by users. The languages chosen for the procedural and object-oriented implementations of the program were C and Java respectively.
These choices were made due to the contrasting nature of the C and Java programming languages, with key differences in terms of memory management, abstraction and compilation processes. Upon initial analysis, we identified potential differences in implementation such as how contacts could be stored through classes and structures, manual and automatic memory management, dissimilarities in how variables are stored and accessed through pointers and references and the resultant effects on each program's control flow, structure and performance.
While implementing the phonebook programs in both of these languages, the core differences were highlighted with advantages and disadvantages of each being noted.

Within the programs, contacts are stored as nodes within the Binary Trees and consist of names, phone numbers and addresses. Reverse search by phone numbers rather than names is also supported within both programs, with two separate Binary Trees being used to store and sort names and phone numbers in each.
The programs also feature a basic command line interface to allow users select commands and enter contact information used within the programs.

C Structs vs Java Classes

C Structs

C structures, or structs, offer developers the ability to organise data by grouping variables together under one structure. These variables can be of any type, and can even be structures themselves as illustrated within our C source code in Figure 1.1 of Appendix 1, where the `tree_node` structure contains a variable defined by the `pbentry` structure. However a structure can only house variables, and any methods that would perform operations on the structure must be declared explicitly in the program and take the structure as an input, creating limitations on the complexity of abstract data types within C.

Multiple instances of a structure may be created, each with unique values. Values may also be modified either through functions or explicitly written in the program. This is supported by abstraction through structures within the C programming language, where abstract data types can be defined by the user and later declared with unique values.

In Figure 1.1 previously mentioned, the procedural C code uses structures to define a phone book entry (the name, phone and address of each person), and the nodes that populate the name and number trees.
Functions that perform operations on the structures must be declared at the top of the file along with any regular functions.

Java Classes

Java classes are similar to C structs in that they allow for easy grouping of variables under one class, but there are some major differences. Java classes can have methods written directly inside the class which can perform operations on the data contained within the class. In C one has to explicitly provide the struct as a parameter to a function, whereas in Java this can be done implicitly using the `object.method()` syntax.

Java classes also have the option of using a constructor method which is called during the instantiation of a class to create an object. This gives the user the option to create a class object with specific values or default values when none are provided.

In the object-oriented Java code shown in Figure 1.2 of Appendix 1, classes are used for the tree nodes in the same way we used structs in the C implementation, however in Java a constructor method can be used to initialise the node with input data. The node constructor shown in Figure 1.2 enables the implicit initialisation of initial parts of newly created nodes based on the parameters passed, advantageously giving the programmer additional options for abstract data types definitions as highlighted by Sebesta [1].
In addition, Java supports inheritance through the use of subclasses that acquire base properties from an encapsulated superclass with the option of specifying additional properties. This further benefits programmers through more readable and easier to maintain abstract data types.

Memory Management & Garbage Collection

Memory management in C

In C, programmers are given relative freedom over the aspect of memory management within programs. Static memory is calculated at compile time, and automatically allocated when the program starts. Memory that is statically allocated is automatically freed based on its scope, as soon as the scope is over, the memory that has been allocated from the stack is freed.

However, dynamic memory allocation must be handled by the programmer and must be explicitly allocated using the `malloc()` or `calloc()` functions, which are used when the compiler cannot determine how much memory will be needed. This memory is allocated from the heap, a bank of memory that the program has for this exact purpose.

An example of this can be seen in Figure 2.1 of Appendix 2, where memory is dynamically allocated for creating a new tree node. It must be allocated dynamically as we give the user the option to input a variable number of nodes, and do not know how much memory will be needed beforehand.

C disadvantageously does not have any in-built garbage collection, so memory must be freed manually when it is no longer being used, which can be seen in Figure 2.2 of Appendix 2 through the deletion of a tree node using the `free()` function.

Memory management in Java

In contrast to C, Java handles the majority of memory management for the programmer. Memory does not need to be explicitly allocated by the developer. Instead of a function such as `malloc()`, the `new` keyword in Java allocates memory for a new class object. The Java Virtual Machine (JVM) determines the amount of memory the object requires and allocates the necessary amount from the heap. This can be seen within the Java phonebook implementation shown in Figure 2.3 of Appendix 2 when we create a new tree node, which requires the JVM to automatically allocate the amount of memory needed to store the tree node when the program is loaded into memory.

Through the JVM, Java also handles the automatic deallocation of unreferenced objects using a garbage collector through either a reference count or mark and sweep algorithm. This advantageously makes memory management simpler for the programmer, but can have a negative effect on the speed of the overall program, as garbage collection is a stop-the-world event, wherein all other threads and activities are suspended until the garbage collection process finishes [2].

C compilation vs Java Interpretation

C Compilation

C source code requires a compilation process before it can be executed on a machine. Compilation involves converting human-readable source code into machine code that can be executed by a computer's CPU [3]. The compilation process also includes checking the syntax and semantics of code to determine any syntax errors or warnings present in the source code [3]. The overall compilation process includes four stages: pre-processing, compilation, assembly and linking depicted in Figure 3.1 of Appendix 3.
Since C compilation uses a static compilation or Ahead-Of-Time (AOT) method, our source code needs to be compiled on the machine it is being executed on.

Java Interpretation

There is a two step compilation process for the execution of Java programs highlighted in Figure 3.2 of Appendix 3. The Java source code is compiled down to bytecode by the java compiler such as javac. This bytecode is then executed by a Java Virtual Machine (JVM) through Just-In-Time (JIT) compilation, also known as dynamic compilation. This execution process involves compiling bytecode to native instructions that are optimised for the running machine's CPU architecture.
Due to the use of the JVM, Java programs are platform independent once compiled to bytecode. This has the advantage of allowing our Java program to be run on any system regardless of the platform it was developed on.

Java References vs C Pointers

C Pointers

A pointer is a variable whose value is the memory address of another variable. Pointer variables point to a variable of the same data type and can be incremented or decremented to point to other memory addresses within contiguous blocks. Within C, we can access the memory address of an existing variable through the reference operator `&` as depicted in Figure 3.3 of Appendix 3 , where a user-defined input is copied into the memory address of the variable `option`. The dereference operator `*` can also be used to get the value at the memory address stored by a pointer.
The manipulation of C pointers provides the advantage of flexibility to developers through lower-level control over resources such as memory blocks, but can be problematic and error-prone if used incorrectly with user inputs.

Java References

In Java, reference variables point to objects or values. Classes, arrays, interfaces, enumerations and annotation are all reference types in Java that can be stored by reference variables. The `new` keyword in Java can be used to create a reference to a particular object defined by a class as previously highlighted in Figure 2.3 of Appendix 2. Java references are

strongly-typed and can also be categorised by how they are managed by the garbage collector of the JVM. These categories include strong, weak, soft and phantom references. Since references are initialised at the time of their declaration, programmers don't need to handle uninitialized wild pointers that point to arbitrary memory addresses as in C. Java references also don't support arithmetic like C pointers, resulting in a more robust security model for programs. This is due to a code-centric security model based on controlling the operations that a class can perform when it is loaded into a running environment as mentioned within the Oracle Fusion Middleware Security Guide [6].

Figure 1 - Comparison of features of C and Java programming languages

C Programming Language (Procedural)	Java Programming Language (Object-Oriented)
Advantages	Advantages
- Speed of compilation - Can serve as a building block for other high-level languages - Offers lower-level memory management useful for Data Structures and Algorithms	- Supports Object-Oriented concepts such as inheritance, operator overloading and operator overriding - Achieves platform-independence through use of JVM - Increased security through distinctions between public/private class properties and a code-based security model
Disadvantages	Disadvantages
- Does not support inheritance due to absence of Object-Oriented features - No built-in error handling feature - Unavailability of run-time type checking - Offers minimum information hiding and exclusive visibility reducing the overall language security	- Just-In-Time (JIT) compilation slows down the overall compilation process - No reliable control over releasing memory due to non-deterministic garbage collection process - Lack of access to low-level programming constructs such as pointers

References

[1] R.W. Sebesta, “Abstract Data Types and Encapsulation Constructs”, in Concepts of Programming Languages, 11 ed. Boston: Pearson, 2015, pp. 446-447

[2] IBM, (2022, Feb. 10), *Pause-less Garbage Collection with Java on IBM Z* [Online], Available: https://www.ibm.com/support/pages/node/6320797

[3] A. Chandra, (2022, Jun. 09), *Compilation Process in C* [Online], Available: https://www.scaler.com/topics/c/compilation-process-in-c/

[4] J. Villegas, (2019, Nov. 19), *Steps of compilation in C* [Online], Available: https://medium.com/basic-command-ls-linux/steps-of-compilation-in-c-737600b43bec

[5] GeeksForGeeks, (2021, Oct. 1), *Compilation and Execution of a Java Program* [Online], Available: https://www.geeksforgeeks.org/compilation-execution-java-program/

[6] Oracle, (2009, May. 06), *Overview of Java Security Models* [Online], Available: https://docs.oracle.com/cd/E12839_01/core.1111/e10043/introjps.htm#JISEC1801

Appendix 1

Figure 1.1 - C structures for contacts and BSTs

```c
/*Create phone book entry structure.*/
typedef struct pbentry {
    char name[50];
    char phone[20];
    char address[30];
}Entry;


/*Create tree node structure.*/
struct tree_node {
    Entry data;
    struct tree_node *left;
    struct tree_node *right;
};
```

Figure 1.2 - Depiction of Java abstract classes

```java
// Node class
class Node {
    String name;
    String addr;
    String number;
    Node left;
    Node right;

    public Node(String name, String addr, String number) { // Node constructor
        this.name = name;
        this.addr = addr;
        this.number = number;
        left = null;
        right = null;
    }
}
```

Appendix 2

Figure 2.1 - Dynamic allocation of memory for a new tree node structure

```
/*Creates a new node.*/
struct tree_node *create_node (struct tree_node *q, struct tree_node *r, Entry e)
{
    struct tree_node* newnode;
    newnode = (struct tree_node*)(malloc(sizeof(struct tree_node)));
    newnode->data = e;
    newnode->left = q;
    newnode->right = r;
    return newnode;
}
```

Figure 2.2 - Freeing dynamically allocated memory used to store tree node structure

```
// node to be deleted
else {
    // node with one or no child
    if (p->left == NULL) {
        struct tree_node *tmp = p->right;
        free(p);
        return tmp;
    }

    else if (p->right == NULL) {
        struct tree_node *tmp = p->left;
        free(p);
        return tmp;
    }
```

Figure 2.3 - Automatic allocation of memory for tree node in Java

```
// Searching by number rather than name
public void numInsert(String name, String addr, String number){
    int i = 0;
    Node temp = new Node(name, addr, number);
```

Appendix 3

Figure 3.1 - C compilation process stages based on [4]

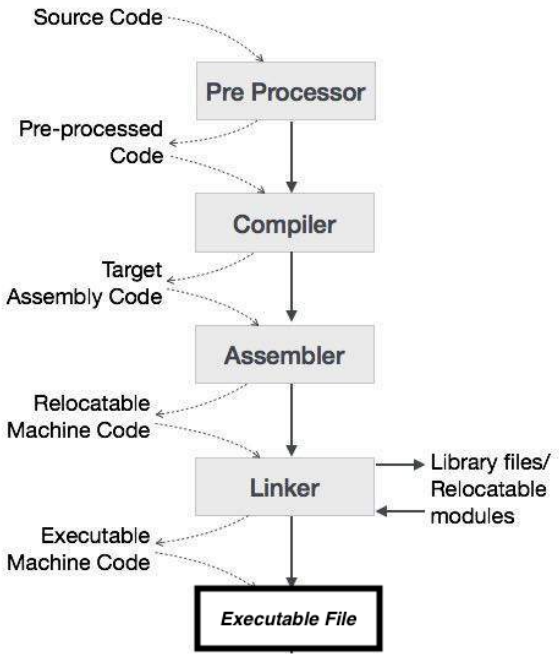

Figure 3.2 - Java compilation process based on [5]

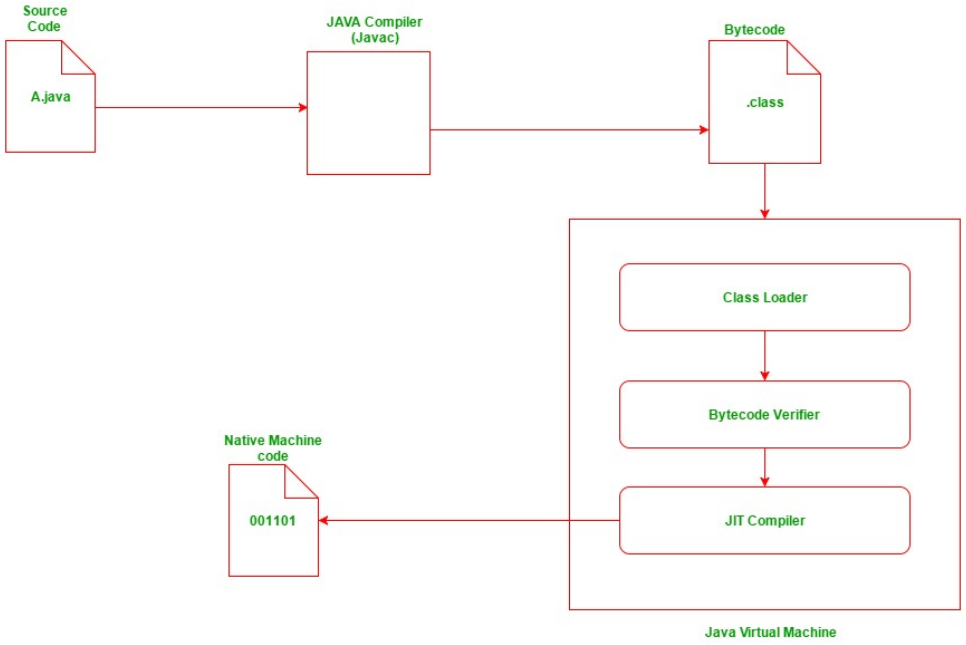

Figure 3.3 - Use of reference operator to store input in memory location of variable

```
/*Get option from the user.*/
printf("\nPlease select an option: ");
scanf("%d", &option);
```