COMPSCI 3MI3 Fall, 2022
By: Nathan Agbomedarho
Student ID: 400081762
Date: December 5 2022

Question 1

In **static arrays**, storage allocation is static, meaning that it is performed in compile time (before execution). The subscript ranges are also statically bound. Its efficiency is the advantage of static arrays. For static arrays, in comparison with dynamic arrays, there is no need for allocation and deallocation. It has efficient execution time and the lifetime of static allocation is the entire run time of program. The only drawback of static arrays is that it has fixed storage that cannot be changed during the program execution.

In **fixed stack-dynamic arrays**, the storage allocation is performed during execution time. However, its subscript ranges are statically bound. It is an advantage over static arrays is space efficiency and it supports recursion. The drawback of fixed stack-dynamic arrays is the time consumed for the allocation and deallocation. Subprograms can share the same space for large arrays, which is applicable for inactive blocks as well.

In **Stack Dynamic arrays**, the stack data structure has a length or size which increases in real time based on operations performed such as insertion or deletion. It has the advantage of flexibility, size doesn't need to be known until array needs to be used.

In **fixed heap-dynamic arrays**, the storage binding and subscript ranges are fixed once storage is allocated. The only difference between fixed stack-dynamic arrays is that, for the former one, the operations mentioned above are performed only after the request of the user during execution. Since this type of array is allocated from the heap section, it takes more time compared with the stack section allocation. Compared to other types of array this type of array is very flexible.

In **heap-dynamic arrays**, the subscript ranges and storage allocation are bound dynamically. However, it is not fixed, the number of bindings can change during the lifetime of the array. In case of necessity of changing space, the size of arrays can string and grow during program run-time, therefore, it is more flexible. The only drawback of this type of array is that since the space of arrays can alter, allocation and deallocation may occur many times, therefore, it takes longer.

Question 2

Decimals are stored as character strings, with the use of binary codes for the decimal digits. The name ascribed to these character representations is Binary

Coded Decimal Format (BCD). They are either stored as one digit per byte or included as two digits per byte.

BCD wastes memory, as 4 bits of memory are needed to represent all potential values. Therefore, N decimal digits require $4N$ to be stored in decimal format. If we were to store 3 characters, this would require 3 bits for storage. However, if the same amount of decimal digits were stored as a binary number, this operation would require $log_2(10N)$ bits $(3.32N)$ bits. Thus, the storage of decimal representations is approximately 20% more expensive, as it requires a higher number of bits than binary format to represent decimal numbers.

Question 3

Tombstones: Tombstones are a mechanism to detect dangling pointers that can appear in certain computer programming languages. A tombstone is a structure that acts as an intermediary between a pointer and the heap-dynamic data in memory. Once the data is deallocated, the variable is set to null (otherwise no longer exists). This prevents the use of invalid pointers, which otherwise would access areas of the memory belonging to the deallocated variable, potentially leading to data corruption.

Lock & Key Method: This is also a solution to dangling pointers. With this approach pointers are represented as ordered pairs which contain keys and addresses where the key is N integer value. Upon deallocation of variables, the key of its pointer is altered and it then possesses a value, differing from that of the variable.

Comparison between both methods: Tombstones unfortunately require a larger computational overhead and memory consumption, extra memory is required to locate the data from the pointer through the tombstone, and also to retain tombstones for each pointer in the program; while lock and key requires less memory consumption.
Tombstones benefit from not requiring additional CPU time however, it requires more time and space complexity, hence its higher cost; the inverse hold for lock and key(i.e. require less time, thus less expensive).
Tombstones are also less secure because there is no protection from memory access violations, while the lock and key method is more secure, as it functions for objects in heap.

Question 4

A language that allows many type coercions can weaken the advantages of strong typing by allowing many potential type errors to be masked by coercing the type of an operand from its incorrect type given in the statement to an

acceptable type, instead of reporting it as an error.

For example, consider the factors $a = 10$ and $b = 20$. Once a division operation is performed(i.e. $c = \frac{a}{b}$), the compiler expects whole numbers, however the result would be a number that is assessed down, instead of a decimal number.

Question 5

Garbage collection is a form of automatic memory management. The garbage collector or collector attempts to reclaim garbage, or memory used by objects that will never be accessed or mutated again by the application. It is a special case of resource management, that assists programmers with memory allocation and deallocation.

Garbage collection in C and C++ are complicated topics because:

1. Pointer can be typecast to integers, and the reverse holds. Which means that a potential memory block reachable by typecasting an integer to a pointer then dereferencing it. A garbage collector could easily mistake a block of memory as unreachable when it fact it is accessible.
2. Pointers aren't opaque. A number of garbage collector (i.e. Stop & Copy collectors), prefer to shift memory blocks around to optimize space. In C++ you can explicitly view pointer values, as such this can be a challenge to implement effectively.
3. Explicit memory management is possible, thus a garbage collector will need to account for the user having the ability to free memory blocks.
4. Allocation/deallocation and object construction/destruction are distinct operations in C++. Memory block can be allocated given sufficient space for an object to exist without actually being constructed at said location. Efficient garbage collectors upon reclaiming memory, needs to know whether to call destructors for objects potentially allocated at a location in memory.
5. Memory can be allocated from different locations. C++ obtains memory from malloc/free or new/delete or from the OS system calls or even a third party library. Good garbage collectors must be able to keep track of references to memory blocks in these areas, and will have to be able to clean them.
6. Pointers can reference in the middle of arrays or objects. In a number of languages that garbage-collect, object references will always point to the beginning of the object. In C++ this increases the complexity of the language's logic and complicates the detection of reachable areas.

Considering the aforementioned points, building a garbage collector in C++ doesn't make sense, as C++ prefers performance and flexibility over safety, and it would also be an extremely difficult and arduous task to achieve. Libraries that perform garbage collection in C++ are not the best, as they make a number

of assumptions that affect the integrity of the program/language logic(i.e. assuming a value in memory that matches the size of a pointer, could be a pointer, so won't free inaccessible memory on the chance a pointer exists).

Consider the following C++ program:

```
class A {
  int x;
public:
  A() { x = 0; ++x; }
};

int main() {
 for (int i = 0; i < 1000000000; ++i) {
   A *a = new A();
   delete a;
 }
 std::cout << "DING!" << std::endl;
}
```

A widely used library that supplies this function is Hans Boehm's conservative GC. C++ supports the powerful idiom **RAII (resource acquisition is initialization)** which can safely and automatically manage memory resources. However this idiom is subject to the aforementioned restrictions.

Question 6

Operator precedence:

Operator precedence stipulates the priority for grouping operators with their operands. It governs rules for which procedures to perform in which order first, to evaluate a given mathematic expression. Operators with higher precedence are evaluated first, and the lower precedence operators follow.

Consider the expression:

$$x = 1 - 4 * 3$$

In the above expression, the precedence of multiplication is higher than subtraction, as such multiplication will be evaluated first and the subtraction will be performed and the result will be:

$$x = -11$$

Operator associativity:

Operator Associativity governs the left-to-right or right-to-left order for grouping operands to operators that have the same precedence. It determines how

operators of the same precedence are executed if parentheses don't exist in the expression.

The right-associative means the operators are grouped from the right. The assignment operators = and $\widehat{}$ are right-associative in most programming languages.

Consider the following expressions:

$$a = b = c$$

First, the value of c is assigned to b and then assign to variable a.

$$x = 2^{3^2}$$

The above expression will be evaluated as follows:

$x = (2)^{(3^2)}$

$x = 2^{(9)}$

$x = 512$

Question 7

(a) $(((a * b)^1 - 1)^2 + c)^3$

(b) $(((a * (b - 1)^1)^2/c)^3 \text{mod d})^4$

(c) $(((a - b)^1/c)^2$ & $(((d * e)^3/a)^4 - 3)^5)^6$

(d) $(((-a)^1 \text{or}(c = d)^2)^3 \text{and e})^4$

(e) $((a > b)^1 \text{xor}(\text{c or}(d \leq 17)^2)^3)^4$

(f) $(-(a + b)^1)^2$

Question 8

Question A :

The value of x after evaluation of the assignment statement: $3 + 4 = 7$.

Question B :

The value of x after evaluation of the assignment statement: $8 + 4 = 12$.

Question C : The GNU C compiler GCC operates by translating C code into assembly code and then assembling it into an executable file. The assembly code is then linked with the standard C library to create the final executable file.

To determine the precedence level of the function call in C, I modified the code by switching the order of x and fun(&x) and I switch + for * . The output showed that the function call occur first. This indicates that the function call in C has the highest precedence level.

Question 9

```
#include <iostream>

using namespace std;

int main()

{

int arr[2];

arr[0] = 1;

arr[1] = 4;

arr[3] = 7;

arr[4] = 8;

cout << arr[3] << endl;

cout << arr[4] << endl;

return 0;

}
```

The size of the array is 3 and it's last index is 2 as defined above and index 3 and 4 are out of bounds. However values are stored at an index out of bounds and print that index it still prints the value stored in it even though it is out of bounds. This is how I know that C/C++ does not check for array bounds for static one dimensional arrays. C/C++ doesn't check array bounds and it can store the data at any memory index.

Question 10

Static scoping This is resolved at compile time, in this the variable referred is always the top level value, the value is independent of the run stack (same value is returned regardless of the base call)

Dynamic scoping Dynamic scoping refers to the most recent variable value

and is decided at run time, the value returned is dependent on the run stack.

```
def foo():
    global x
    print(x)
    x=10
x='global'
foo() #print 'global'
foo() #print '10'

#Dynamic

def foo:
    x=10 #refers to the local (dynamic) variable
    print(x) #print local 'x'
x='global'
foo() #print '10'
foo() #print '10'
```

If a variable is assigned a value anywhere within the function's body, it's assumed to be local unless explicitly declared as global {static}.

When a name is used in a code block, it is resolved using the nearest enclosing scope. Since the variable x is assigned inside foo() it is assumed to be a local variable starting from the beginning of the function foo(). It is possible to treat x within the entire function block as global using the global keyword.