2021315385
이건 Gun Daniel Lee

Q1: Answer The Following Questions with Short Answers

1. What is the potential danger of case-sensitive names in a programming language?
The potential danger of case-sensitive names in a programming language lies in its reduced readability due to names that look alike but are different.

2. Which category of C++ reference variables is always aliases?
The l-value reference variables of C++ are always aliases.

3. Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?
The advantage of having a programming language with no types is its flexibility since it allows for the creation of generic program units that can operate on different types of data.

The disadvantages are the high cost of dynamic type checking and interpretation, and the difficulty of type error detection before execution by the compiler.

4. How does a decimal datatype waste memory space?
A decimal datatype stores a fixed number of decimal digits separately in binary coded decimal (BCD) form. This means that the numerical value of each decimal digit is stored separately using 4 bits per digit. The use of 4 bits per digit causes a huge waste of memory space.

5. What are all of the differences between the enumeration types of C++ and those of Java?
In C++, enumeration type variables are coerced into integer types, so it is only a convenient way of naming integers.

In Java, the enumeration type is a type-safe class. This means that the variables are not coerced into integers. Therefore, Java′s enumeration type variables provide stronger and better support for enumeration, as well as the OOP and safety features that are seen in Java classes.

6. Search and write a comparison of C′s malloc and free functions with C++′s new and delete operators. Mention about safety in the comparison.
C′s malloc and free do not call constructors or destructors, instead they allocate and deallocate raw memory, which can lead to resource leaks. The malloc function is not type safe as it returns a generic void* pointer, requiring explicit casting. Upon error, malloc returns a NULL pointer, which requires manual checking to prevent segmentation faults. As for the free function, a pointer still holds the address of the deallocated memory, creating a dangling pointer, so this must be handled explicitly.

C++′s new and delete operators are type safe. The new operator returns a pointer of the exact data type requested. These operators call constructors and destructors when allocating and deallocating memory. This makes new and delete safer for OOP by ensuring objects are properly initialized and cleaned up. Robust error handling is possible since the new operator throws an exception when memory allocation fails.

Q2: Consider the following skeletal C program:

```
void fun1(void); /* prototype */
void fun2(void); /* prototype */
void fun3(void); /* prototype */
void main() {
  int a, b, c;
    . . .
}
void fun1(void) {
  int b, c, d;
   . . .
}
void fun2(void) {
  int c, d, e;
   . . .
}
void fun3(void) {
 int d, e, f;
    . . .
}
```

Given the following calling sequences and assuming that dynamic scoping is used, what **variables** are **visible** during execution of the **last function called**? Include with each visible variable the name of the function in which it was defined.

a. `main` calls `fun1`; `fun1` calls `fun2`; `fun2` calls `fun3`.
b. `main` calls `fun1`; `fun1` calls `fun3`.
c. `main` calls `fun2`; `fun2` calls `fun3`; `fun3` calls `fun1`.
d. `main` calls `fun3`; `fun3` calls `fun1`.
e. `main` calls `fun1`; `fun1` calls `fun3`; `fun3` calls `fun2`.
f. `main` calls `fun3`; `fun3` calls `fun2`; `fun2` calls `fun1`.

Question	Visible variables	Function where the variable declared
a	a	main
	b	fun1
	c	fun2
	d, e, f	fun3
b	a	main
	b, c	fun1
	d, e, f	fun3
c	a	main
	e, f	fun3
	b, c, d	fun1
d	a	main
	e, f	fun3
	b, c, d	fun1
e	a	main
	b	fun1
	f	fun3
	c, d, e	fun2
f	a	main
	f	fun3
	e	fun2
	b, c, d	fun1

Q3: Consider the following Python program

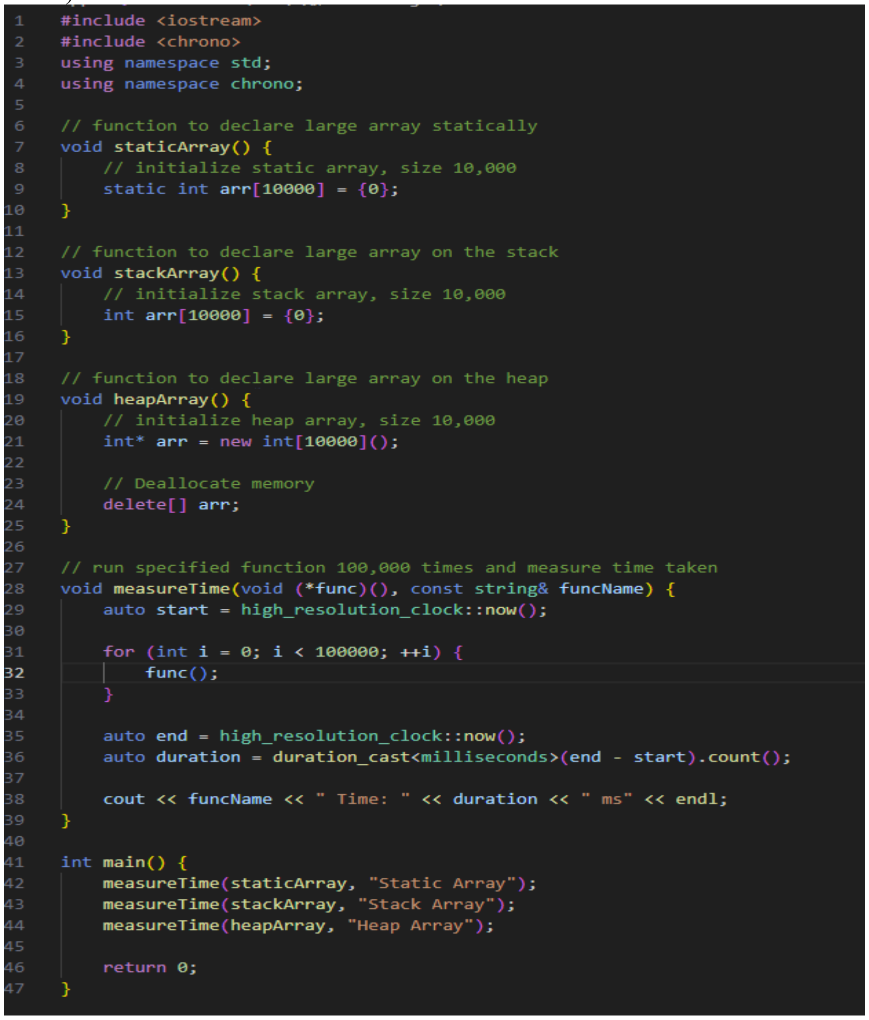
```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
  ...

def sub2():
  global x;

  a = 13;
  x = 15;
  w = 17;
  ...

  def sub3():
    nonlocal a;
    a = 19;
    b = 21;
    z = 23;
    ...
...
```

Similar to Q2, list all the variables, along with the function where they are declared, that are visible in the bodies of sub1, sub2 and sub3 assuming static scoping is used.

variables	function where they are declared	Where it is visible
a	**sub1, sub2**	**sub1(sub1), sub2(sub2), sub3(sub2)**
b	**sub3**	**sub3**
w	**sub2**	**sub2, sub3**
x	**global**	**sub1, sub2, sub3**
y	**global, sub1**	**sub1(sub1), sub2(global), sub3(global)**
z	**global, sub1, sub3**	**sub1(sub1), sub2(global), sub3 (sub3)**

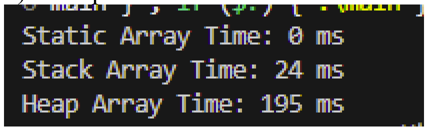
Q4: Write three functions in C or C++: one that declares a large array statically, one that declares the same large array on the stack, and one that creates the same large array from the heap. Call each of the subprograms a large number of times (at least 100,000) and output the time required by each. 1)Include the code, 2)run snapshot show that time of each function, and 3) explain why you get this results.

1) Code



```cpp
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;

// function to declare large array statically
void staticArray() {
    // initialize static array, size 10,000
    static int arr[10000] = {0};
}

// function to declare large array on the stack
void stackArray() {
    // initialize stack array, size 10,000
    int arr[10000] = {0};
}

// function to declare large array on the heap
void heapArray() {
    // initialize heap array, size 10,000
    int* arr = new int[10000]();

    // Deallocate memory
    delete[] arr;
}

// run specified function 100,000 times and measure time taken
void measureTime(void (*func)(), const string& funcName) {
    auto start = high_resolution_clock::now();

    for (int i = 0; i < 100000; ++i) {
        func();
    }

    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start).count();

    cout << funcName << " Time: " << duration << " ms" << endl;
}

int main() {
    measureTime(staticArray, "Static Array");
    measureTime(stackArray, "Stack Array");
    measureTime(heapArray, "Heap Array");

    return 0;
}
```

2) Snapshot



```
Static Array Time: 0 ms
Stack Array Time: 24 ms
Heap Array Time: 195 ms
```

3) Explanation of results

The results show that static allocation is the fastest, while heap allocation is the slowest.

Static allocation is the fastest because the array variable is bound to the memory cell in the data segment before program execution begins and remains there for the entire program's lifetime. Therefore, each function call does not have any allocation overhead. It just accesses memory that is already reserved.

Stack allocation was the second fastest since the array variable is allocated on the stack when the declaration is executed and deallocated when the function exits. Stack allocation is very quick and only involves adjusting the stack pointer. However, repeating this for 100,000 times creates a bit of overhead, resulting in a small increase in time taken.

Despite the efficiency and flexibility, heap allocation was the slowest due to its complexity compared to the other allocations. An explicit heap-dynamic array variable is allocated and deallocated during runtime. The heap manager must find a suitable free block of memory, perform bookkeeping, and later manage deallocation and fragmentation. Due to its complexity, it is significantly slower than the other two allocations.