2022310853 박연우 (Younwoo Park)

Q1: Answer The Following Questions with Short Answers

1. **What is the potential danger of case-sensitive names in a programming language?**

Case sensitive names can lead to reduced readability, causing confusion and difficulty in debugging. For instance, in case-sensitive programming languages, variables "Count" and "count" are different, making the code harder to read, maintain, and debug.

2. **Which category of C++ reference variables is always aliases?**

Reference variables are always aliases for the objects they refer to. To be specific, L-value references (ex: int &a = var;) are always aliases in C++. An L-value reference creates an alias to an existing object, meaning any changes made through the reference affect the original object. They refer to existing objects and cannot be null or unbound, which is why they're considered permanent aliases.

3. **Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?**

Advantages:
- Faster prototyping since no type declarations are needed
- More flexible as variables can hold any value
- Dynamic behaviour as functions can handle different data types without strict type constraints

Disadvantages:
- Less efficient because the code cannot be optimized based on known types and dynamic type checking slows down execution
- Poor readability and maintainability because data type of variable is not explicit and harder to understand and refactor

4. **How does a decimal datatype waste memory space?**

Decimal datatypes wastes memory because it stores numbers with high precision which often takes more bits than necessary. By allocating a fixed amount of storage based on its declared precision and scale, it creates inefficiency when values don't require the full precision.

Ex: 12 = 1100 in binary, but 0001 0010 in decimal (BCD code) → decimal requires an additional 4 bits

5. **What are all of the differences between the enumeration types of C++ and those of Java?**

C++ Enums
- Enums are basically a set of named, integral constants (aka. Simple named integers)
- Can implicitly convert to/from integers
- No methods, fields, or constructors
- Can assign custom int values
- Two forms: enum (unscoped) and enum class (scoped, type-safe) → can have scoping issues
- Inheritance is not possible

Java Enums
- Each constant is an object of the enum class
- No implicit int conversion; conversion must be explicit (Type safety)
- Can have field, methods, and constructors
- Can implement interfaces, override methods, and have built-in methods
- Clear scope provided
- Inherits from `java.lang.Enum`

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 & free functions:

- Allocates memory from the heap
- Memory must be manually managed: programmer specifies size and use free() to release it
- Allocates raw memory without initialization, which can lead to garbage values
- Not type safe; requires casting from `void*`
- No constructor or destructors calls
- Manual size calculation with sizeof
- Returns NULL of failure

C++′s new and delete operators

- Operates from free store, with memory management built in
- Initializes memory during allocation
- Type safe; automatically returns correct type
- Calls constructors and destructors automatically
- Automatic size handling
- Returns std::bad_alloc on failure

C′s malloc and free is more error prone compared to C++′s new and delete operators because of manual size handling, no constructor/destructor calls, and lack of type safety. The manual memory management can lead to erros such as memory leaks, resource leaks, and dangling pointers. C++ is has safer object management features, managing memory safely and reducing the possibility of errors.

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

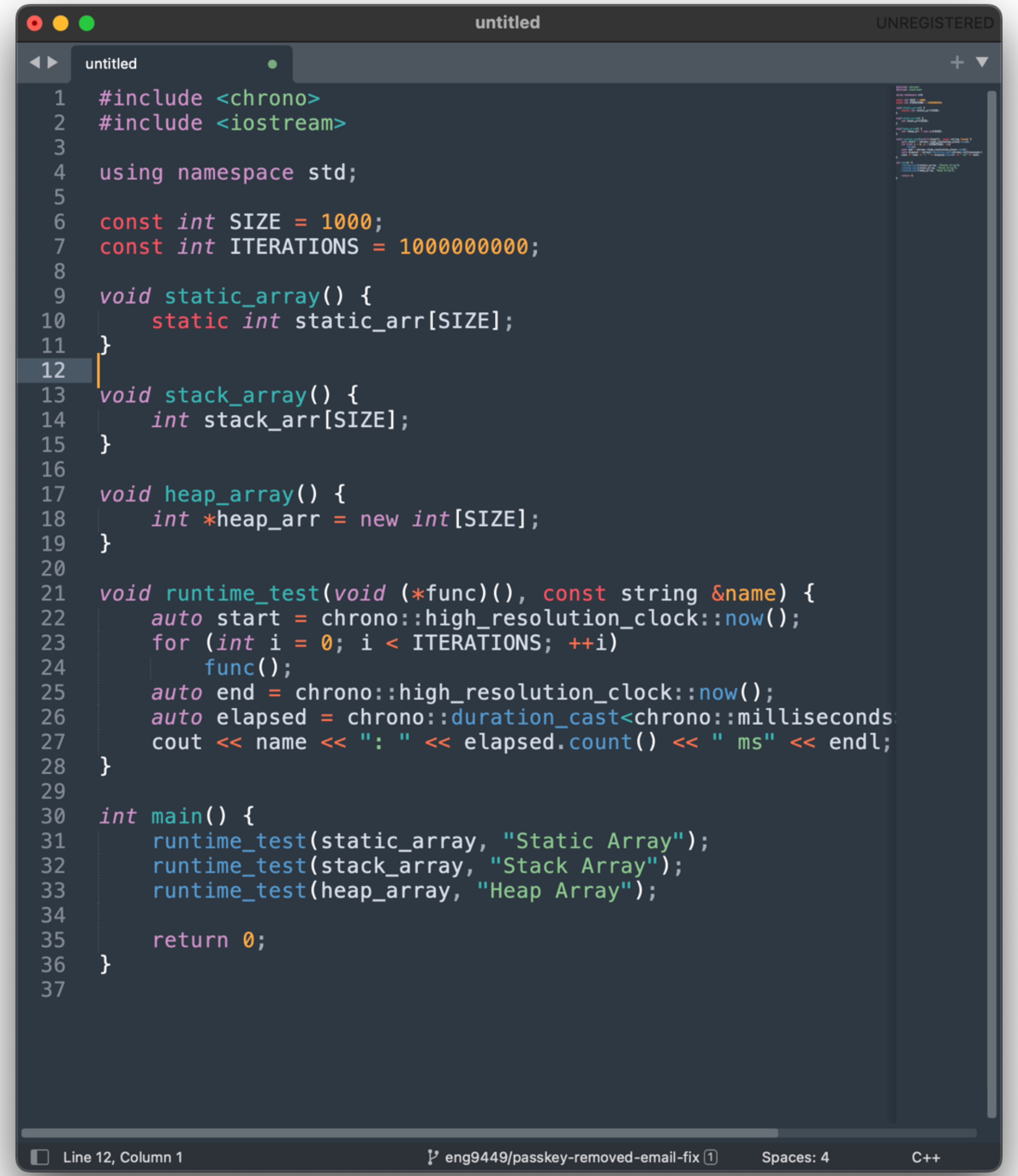
```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
  ...

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
x	Declared globally	Global scope (reassigned in sub2)
y	Declared globally, sub1()	Global scope, sub1() (shadowed)
z	Declared globally, sub1(), sub3()	Global scope, sub2() (shadowed), sub3() (shadowed)
a	sub1(), sub2()	Local to sub1(), local to sub2() (as nonlocal reassigned in sub3())
w	sub2()	sub2(), sub3()
b	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**

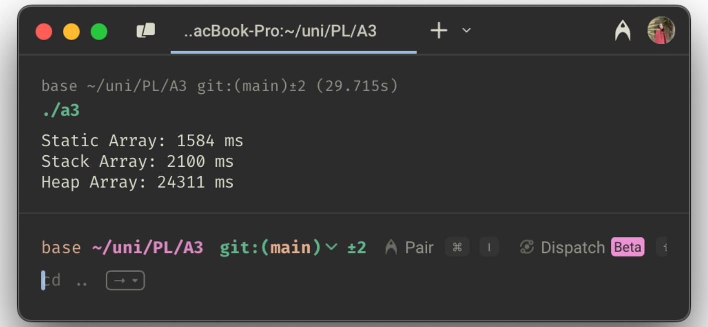


```
#include <chrono>
#include <iostream>

using namespace std;

const int SIZE = 1000;
const int ITERATIONS = 1000000000;

void static_array() {
    static int static_arr[SIZE];
}

void stack_array() {
    int stack_arr[SIZE];
}

void heap_array() {
    int *heap_arr = new int[SIZE];
}

void runtime_test(void (*func)(), const string &name) {
    auto start = chrono::high_resolution_clock::now();
    for (int i = 0; i < ITERATIONS; ++i)
        func();
    auto end = chrono::high_resolution_clock::now();
    auto elapsed = chrono::duration_cast<chrono::milliseconds
    cout << name << ": " << elapsed.count() << " ms" << endl;
}

int main() {
    runtime_test(static_array, "Static Array");
    runtime_test(stack_array, "Stack Array");
    runtime_test(heap_array, "Heap Array");

    return 0;
}
```

untitled UNREGISTERED

Line 12, Column 1 eng9449/passkey-removed-email-fix 1 Spaces: 4 C++

2. Results



```
base ~/uni/PL/A3 git:(main)±2 (29.715s)
./a3
Static Array: 1584 ms
Stack Array: 2100 ms
Heap Array: 24311 ms

base ~/uni/PL/A3 git:(main) ±2
cd ..
```

3. Explanation

Static Array (1584 ms)

Static array is allocated only once globally and reused across all function calls. The memory for this static array persists throughout the lifetime of the program, and since there is no repeated allocation or deallocation cost, it has the shortest runtime.

Stack Array (2100 ms)

A new array is allocated and deallocated in the stack every time the function is called. Stack allocation is quick as it simply moves the stack pointer (single instruction), allocating a fixed-size region of memory. However, it is slower compared to allocating a stack array because of the repeated allocation/deallocation overhead in creating a new array each time.

Heap Array (24311 ms)

Memory is dynamically allocated within the heap during runtime. Heap allocation is much more expensive due to fragmentation handling, overhead in memory management, and slower allocation/deallocation compared to the stack; a suitable block of memory needs to be found and returned by the operating system.