

# Assignment 3

2020310083 Hyungjun Shon

Dept. of Systems Management Engineering  
Sungkyunkwan University

October 19, 2024

## Contents

<b>1 Q1: Answer the following questions</b>	<b>2</b>
1.1 Potential danger of case-sensitive names . . . . .	2
1.2 Category of C++ reference variables is always aliases . . . . .	2
1.3 Advantages and disadvantages of having no types in a language . . . . .	2
1.4 How does a decimal datatype waste memory space . . . . .	2
1.5 Differences between the enumeration types of C++ and those of Java . . . . .	2
1.6 Comparison of C's malloc and free with C++'s new and delete . . . . .	3
1.6.1 Memory Allocation / Deallocation / Return type . . . . .	3
1.6.2 Error Handling . . . . .	3
1.6.3 Initialization . . . . .	3
1.6.4 Safety and Efficiency . . . . .	3
1.6.5 Reallocation and Arrays . . . . .	3
1.6.6 Constructors and Destructors . . . . .	3
1.6.7 Conclusion . . . . .	3
<b>2 Q2: Variable Visibility (dynamic scoping)</b>	<b>4</b>
2.1 (a) main calls fun1; fun1 calls fun2; fun2 calls fun3. . . . .	4
2.2 (b) main calls fun1; fun1 calls fun3. . . . .	4
2.3 (c) main calls fun2; fun2 calls fun3; fun3 calls fun1. . . . .	4
2.4 (d) main calls fun3; fun3 calls fun1. . . . .	4
2.5 (e) main calls fun1; fun1 calls fun3; fun3 calls fun2. . . . .	4
2.6 (f) main calls fun3; fun3 calls fun2; fun2 calls fun1. . . . .	4
<b>3 Q3: variable visibility (static scoping)</b>	<b>5</b>
3.1 Given code . . . . .	5
3.2 Variables with Where They Are Declared and Visible . . . . .	5
3.2.1 variable x . . . . .	5
3.2.2 variable y . . . . .	5
3.2.3 variable z . . . . .	5
3.2.4 variable a . . . . .	5
3.2.5 variable w . . . . .	5
3.2.6 variable b . . . . .	5
<b>4 Q4: static / stack / heap array manipulation</b>	<b>6</b>
4.1 Code . . . . .	6
4.2 Result snapshot . . . . .	6
4.3 Explanation . . . . .	7
4.3.1 Static Array (staticArrayFunc) . . . . .	7
4.3.2 Stack Array (stackArrayFunc) . . . . .	7
4.3.3 Heap Array (heapArrayFunc) . . . . .	7
4.3.4 Conclusion . . . . .	7

# 1 Q1: Answer the following questions

## 1.1 Potential danger of case-sensitive names

Case-sensitive names have disability in readability and maintainability due to the potential for confusion between similar-looking identifiers. (The names that look alike may be different)

This can cause bug that are difficult to spot, especially in large programs where developers might inadvertently use the wrong case.

## 1.2 Category of C++ reference variables is always aliases

In C++, reference variables are always aliases for the objects they refer to. Specifically, lvalue references (e.g., `int &ref = var;`) are always aliases for existing variables.

## 1.3 Advantages and disadvantages of having no types in a language

**Advantages: writeability**

- Flexibility in manipulating different types of data without explicit type conversions.

**Disadvantages: reliability**

- Lack of type checking increases the likelihood of runtime errors.
- Debugging can be more challenging due to obscure errors that occur when types are misused. (lack of type checking in compiler)

## 1.4 How does a decimal datatype waste memory space

Decimal datatypes can waste memory space because they store all digits in a fixed-size container, even if the number is small. So when the number get larger, the memory space for decimal datatype will be way larger than the integer or float datatype. (each digit encode sperately thatn encode the whole number)

$$\begin{aligned} 10 &\rightarrow 1010 \text{ in binary} \\ &\rightarrow 0001\ 0000 \text{ in decimal} \end{aligned} \quad (1)$$

For example, the decimal number 10 requires 4 bits in binary but need 8 bits in decimal.

## 1.5 Differences between the enumeration types of C++ and those of Java

The differences between enumeration types in C++ are as follows:

Aspect	C++	Java
<b>Definition</b>	Simple set of constants	Enum behaves like a class
<b>Underlying Type</b>	Integer-based	Enum has its own type
<b>Method/Field Addition</b>	Not possible	Possible, enums can have fields and methods
<b>Scoping</b>	Traditional <code>enum</code> can have scoping issues (fixed by <code>enum class</code> )	Clear scoping provided
<b>Value Comparison</b>	Compares integer values	Enum constant objects are compared
<b>Reflection</b>	Not supported	Supported, can access enum constants via reflection

Aspect	C++	Java
Inheritance	Not possible	Implicitly inherits from <code>java.lang.Enum</code>
Serialization/Deserialization	Handled as integers	Automatically handled

Furthermore, C++ enumeration types can perform arithmetic/bitwise operations (since they are integer-based), but Java cannot.

In C++, if you want to assign the operation result to an enumeration variable, the result should be in the range of the enumeration values

## 1.6 Comparison of C's `malloc` and `free` with C++'s `new` and `delete` <sup>1</sup>

### 1.6.1 Memory Allocation / Deallocation / Return type

- C's `malloc()` and `free()` allocate memory from the **heap**. The memory must be manually managed, requiring the programmer to specify the size and use `free()` to release it. It returns a `void*`, requiring explicit casting to the correct type.
- C++'s `new` and `delete` operate from the **Free Store** <sup>2</sup>, with memory management built into the language. `new` returns a fully typed pointer, and memory initialization occurs automatically for objects. `delete` releases memory and calls destructors.

### 1.6.2 Error Handling

- C's `malloc()` returns NULL if the memory allocation fails, requiring manual error checking.
- C++'s `new` throws a `std::bad_alloc` exception if allocation fails, enabling exception handling. If exceptions are undesirable, a `nothrow` version of `new` can be used to return NULL, similar to `malloc()`.

### 1.6.3 Initialization

- `malloc()`: Allocates raw memory without initialization, often containing garbage values.
- `new`: Initializes memory during allocation. For primitive types, a default constructor can be specified (e.g., `new int()`; initializes to 0), and for objects, constructors are automatically invoked.

### 1.6.4 Safety and Efficiency

- `malloc()` / `free()`: Manual memory management can lead to errors such as **memory leaks** (failing to `free()` memory) and **dangling pointers** (using memory after it is freed). Additionally, `free()` does not call destructors, which can lead to resource leaks when using complex objects.
- `new` / `delete`: While also prone to **memory leaks** and **dangling pointers** if improperly used, C++ offers tools like **smart pointers** (e.g., `std::unique_ptr`, `std::shared_ptr`) to manage memory more safely. These tools automatically free memory when no longer in use, reducing the chance of leaks or errors.

### 1.6.5 Reallocation and Arrays

- `malloc()`: Requires explicit size calculations for arrays and handling reallocation is not intuitive. The memory allocated must be manually tracked.
- `new[]`: Directly supports arrays, though the reallocation is not handled efficiently within `new`—users typically use containers like `std::vector` for dynamic arrays to avoid manual memory management.

### 1.6.6 Constructors and Destructors

- `malloc()` does not call constructors or destructors, making it unsuitable for object management in C++.
- `new` / `delete` ensure proper initialization and cleanup by calling constructors and destructors automatically, providing safer object-oriented memory handling.

### 1.6.7 Conclusion

In summary, `new` / `delete` offer improved safety and integration in C++ compared to `malloc` / `free`, particularly in areas such as error handling, type safety, and object management. However, both approaches require caution to avoid common memory issues like leaks and dangling pointers.

<sup>1</sup><https://stackoverflow.com/questions/240212/what-is-the-difference-between-new-delete-and-malloc-free>

<sup>2</sup>The **heap** refers to the general area for dynamic memory allocation, while the **free store** specifically refers to the portion of the heap managed by C++'s `new` and `delete` operators.

## 2 Q2: Variable Visibility (dynamic scoping)

### 2.1 (a) **main** calls **fun1**; **fun1** calls **fun2**; **fun2** calls **fun3**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun2)
- d (declared in fun3)
- e (declared in fun3)
- f (declared in fun3)

### 2.2 (b) **main** calls **fun1**; **fun1** calls **fun3**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun1)
- d (declared in fun3)
- e (declared in fun3)
- f (declared in fun3)

### 2.3 (c) **main** calls **fun2**; **fun2** calls **fun3**; **fun3** calls **fun1**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun1)
- d (declared in fun1)
- e (declared in fun3)
- f (declared in fun3)

### 2.4 (d) **main** calls **fun3**; **fun3** calls **fun1**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun1)
- d (declared in fun1)
- e (declared in fun3)
- f (declared in fun3)

### 2.5 (e) **main** calls **fun1**; **fun1** calls **fun3**; **fun3** calls **fun2**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun2)
- d (declared in fun2)
- e (declared in fun2)
- f (declared in fun3)

### 2.6 (f) **main** calls **fun3**; **fun3** calls **fun2**; **fun2** calls **fun1**.

- a (declared in main)
- b (declared in fun1)
- c (declared in fun1)
- d (declared in fun1)
- e (declared in fun2)
- f (declared in fun3)

### 3 Q3: variable visibility (static scoping)

All of the variables with where they are declared and where they are visible are as follows:

Thank you for providing the clarification about the indentation. Given the correct structure of the code with proper indentation, I now understand that **sub3()** is **nested within sub2()** and has access to variables in **sub2()**'s scope. Let me walk through the visibility of variables with this updated structure in mind.

#### 3.1 Given code

```
x = 1
y = 3
z = 5

def sub1():
    a = 7
    y = 9 # local to sub1 (shadows global y)
    z = 11 # local to sub1 (shadows global z)

def sub2():
    global x # global x is modified
    a = 13 # local to sub2
    x = 15 # global x is modified
    w = 17 # local to sub2

    def sub3():
        nonlocal a # refers to `a` from sub2
        a = 19 # modifies `a` in sub2
        b = 21 # local to sub3
        z = 23 # local to sub3 (shadows global z)
```

#### 3.2 Variables with Where They Are Declared and Visible

##### 3.2.1 variable x

- **Declared in:** declared globally
- **Visible in:** Global scope (reassigned in sub2())

##### 3.2.2 variable y

- **Declared in:** declared globally, locally declared in sub1()
- **Visible in:** Global scope, shadowed in sub1() (local a will be visible in sub1())

##### 3.2.3 variable z

- **Declared in:** declared globally, locally declared in sub1() and sub3()
- **Visible in:** Global scope, shadowed in sub1() and sub3() (local z will be visible in sub1() and sub3())

##### 3.2.4 variable a

- **Declared in:** Local to sub2(), nonlocally declared in sub3() (refers to a that locally declared in sub2())
- **Visible in:** Local to sub2(), sub3() (due to nonlocal declaration, a in sub3() refers to a declared in sub2())

##### 3.2.5 variable w

- **Declared in:** Local to sub2()
- **Visible in:** Local to sub2(), sub3() (due to nesting of sub3() inside sub2())

##### 3.2.6 variable b

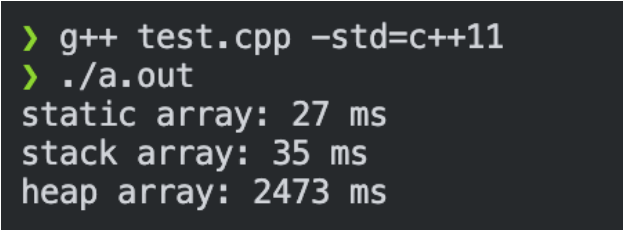
- **Declared in:** Local to sub3()
- **Visible in:** Local to sub3()

## 4 Q4: static / stack / heap array manipulation

### 4.1 Code

```
1  #include <iostream>
2  #include <chrono>
3
4  const int SIZE = 1000000;
5
6  // only declaration (no initialization)
7
8  // static array
9  void staticArrayFunc() {
10     static int arr[SIZE];
11 }
12
13 // stack array
14 void stackArrayFunc() {
15     int arr[SIZE];
16 }
17
18 // heap array (no delete for fair comparison)
19 void heapArrayFunc() {
20     int* arr = new int[SIZE];
21 }
22
23 void measureTime(void (*f)(), const std::string& type, int iter) {
24     using clock = std::chrono::high_resolution_clock;
25     std::chrono::high_resolution_clock::time_point start = clock::now();
26
27     for (int i = 0; i < iter; ++i)
28         f();
29
30     std::chrono::high_resolution_clock::time_point end = clock::now();
31     std::chrono::milliseconds dur = std::chrono::duration_cast<
32         std::chrono::milliseconds>(end - start);
33
34     std::cout << type << ": " << dur.count()
35         << " ms\n";
36 }
37
38 int main() {
39     int iter = 1000000;
40     measureTime(staticArrayFunc, "static array", iter);
41     measureTime(stackArrayFunc, "stack array", iter);
42     measureTime(heapArrayFunc, "heap array", iter);
43     return 0;
44 }
```

### 4.2 Result snapshot



```
> g++ test.cpp -std=c++11
> ./a.out
static array: 27 ms
stack array: 35 ms
heap array: 2473 ms
```

Figure 1: Time Comparison for Static, Stack, and Heap Arrays

## 4.3 Explanation

### 4.3.1 Static Array (`staticArrayFunc`)

- **Execution Time:** 27 ms (fastest)
- **Memory Allocation:** Compile-time (allocated once for the entire program)

A static array is declared with a fixed size and allocated once at compile time. The memory for this array persists throughout the lifetime of the program, meaning that even when the function is called repeatedly, the array is not reallocated. Instead, the same memory block is reused each time. This eliminates the overhead of repeatedly allocating or deallocating memory, which allows for fast access. The memory is placed in the data segment of the program, which is initialized when the program starts. Since no additional allocations are required during runtime, static arrays offer very efficient access times.

### 4.3.2 Stack Array (`stackArrayFunc`)

- **Execution Time:** 35 ms (slightly slower than static)
- **Memory Allocation:** Runtime (allocated and deallocated on each function call)

A stack array is a local array, allocated automatically each time the function is called. The memory comes from the stack, which is a limited, fixed-size region of memory. Allocating memory on the stack is typically very fast because it only involves adjusting the stack pointer (a single instruction), making stack arrays very efficient. However, since the array is allocated and deallocated with every function call, there is some overhead involved in managing this process, although it's still much faster than heap allocation. Stack memory is automatically freed when the function exits, so there is no risk of memory leaks.

### 4.3.3 Heap Array (`heapArrayFunc`)

- **Execution Time:** 2473 ms (significantly slower)
- **Memory Allocation:** Runtime (allocated and deallocated manually)

Heap arrays are allocated dynamically at runtime using functions like `new` or `malloc`. The heap is a large pool of memory managed by the operating system, but allocating memory from the heap is slower than stack or static allocation because it involves finding a suitable block of memory, updating management structures, and possibly handling fragmentation. Each time the function is called, a new block of memory is requested, which incurs a significant performance cost. Additionally, the heap memory must be manually deallocated, and failure to do so can lead to memory leaks.

### 4.3.4 Conclusion

- **Static Arrays:** Fastest due to a single allocation at compile-time with no overhead for repeated allocations.
- **Stack Arrays:** Slightly slower due to allocation and deallocation on each function call, but still efficient thanks to the quick stack pointer adjustments.
- **Heap Arrays:** Slowest because dynamic memory allocation involves significant overhead in finding, allocating, and managing memory at runtime.