2021312738 소프트웨어학과 김서환

Q1: Answer The Following Questions with Short Answers

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

In case-sensitive languages, even the same name is treated as a completely different identifier if the case is different. This may result in several similar names, resulting in bugs caused by typos, and disadvantage that is less readability.

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

The reference variable in C++ becomes another name for an existing object immediately upon declaration. That is, all C++ reference variables are essentially aliases

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

The advantage of typeless programming languages is writability, because variables can be used straightforwardly without type declarations, which is advantageous for simple scripting and faster writing. The disadvantage of typeless programming languages is that the lack of type inspection can lead to type errors at runtime, resulting in poor stability and reliability.

4. How does a decimal datatype waste memory space?

Because the decimal type stores values in a binary-coded decimal (BCD) manner, it represents only one decimal number, 0 to 9, for every four bits. This means that more memory is used to store the same value compared to the binary representation method.

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

The difference between c++ and java's enum data types can be summarized into three categories.
The first is the integer transformation for enum values. In c++, enum values are implicitly transformed into integer values and arithmetic operations are allowed. java says that enum is an object type and is not compatible with integer types, so if I want to use integer values, I can use the .ordinal() method.
The second is the reference method. Although c++ allows to refer directly to variables, Java must refer to the enum value through a class identifier. (example: Week.values())
The Third is that in c++, enum is a simple constant set of data values for objects, but in java, enum is much more flexible because it can define fields, methods, constructors, and so on.

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.

The malloc of c is simply borrowing and allocating dynamic memory. There is a high possibility of runtime errors because it is not initialized, and there may be a garbage value in the allocated memory, and free is a function that simply releases the allocated memory. Unlike C++, there is no call from a destructor. On the other hand, C++'s new allocates dynamic memory and automatically calls the constructor to initialize variables. Also, delete calls the destructor to safely clean up the object and releases the allocated memory. And when comparing from a safety perspective, malloc/free is not initialized, so there may be garbage values inside and errors can occur. In addition, it is less safe in that everything has to be managed by the developer because the developer must release the allocated memory. new/delete is more secure than c in that there is no trash value by initializing constructor calls and variables through new, and new/delete also has to automatically organize objects safely through the extinction call, but it is difficult to say that safety is perfect because the developer must release the allocated memory directly.

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;
    ...
...
```

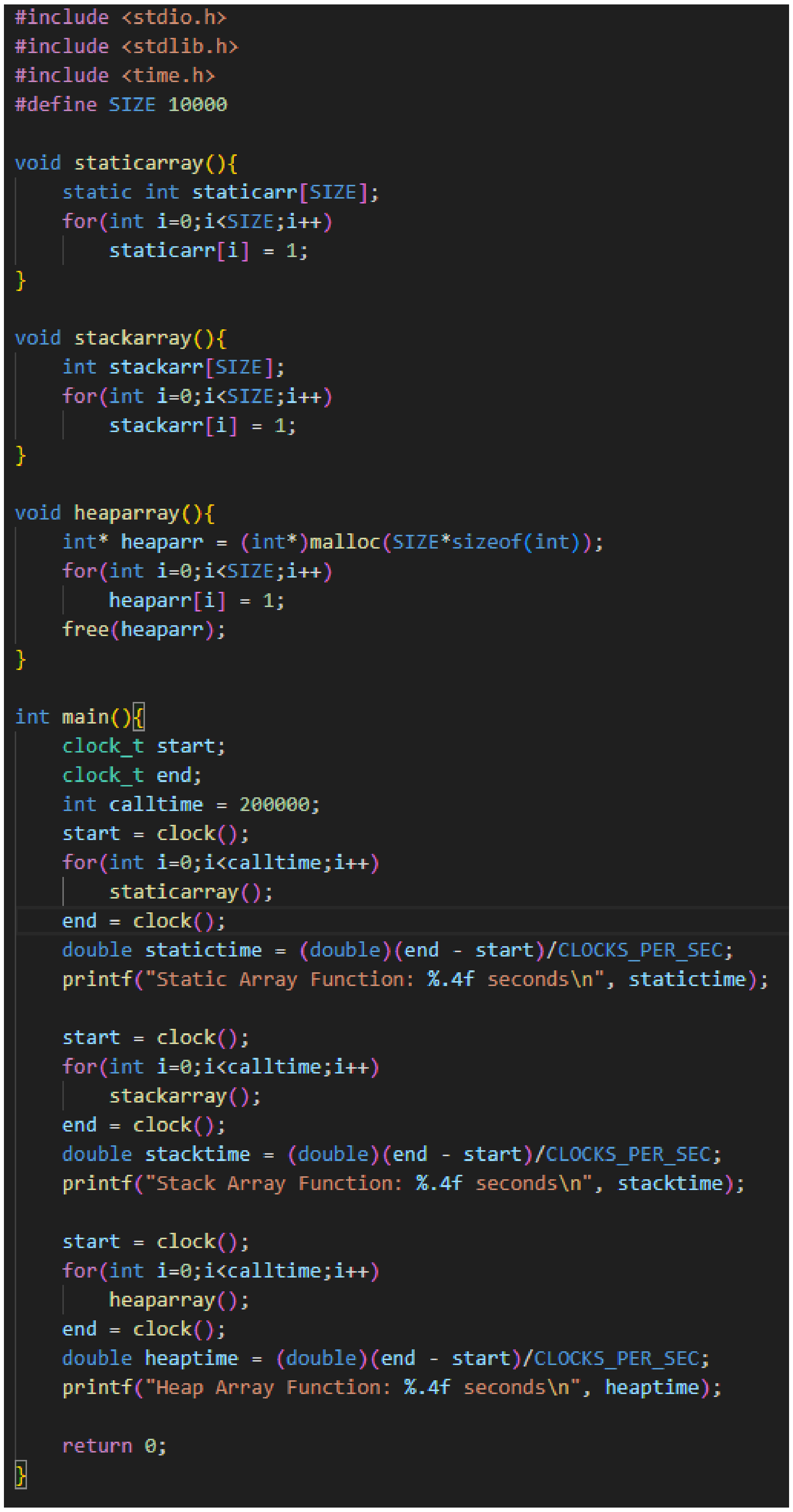
Similar to Q2, list all the variables, along with the function where they are declared, that are visible in the bodies of sub1, sub2 and sub3 assuming static scoping is used.

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

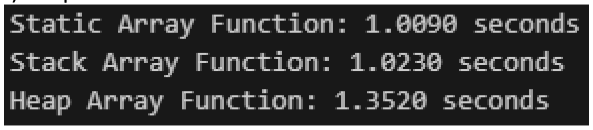
1) <C code>

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10000

void staticarray(){
  static int staticarr[SIZE];
  for(int i=0;i<SIZE;i++)
    staticarr[i] = 1;
}

void stackarray(){
  int stackarr[SIZE];
  for(int i=0;i<SIZE;i++)
    stackarr[i] = 1;
}

void heaparray(){
  int* heaparr = (int*)malloc(SIZE*sizeof(int));
  for(int i=0;i<SIZE;i++)
    heaparr[i] = 1;
  free(heaparr);
}

int main(){
  clock_t start;
  clock_t end;
  int calltime = 200000;
  start = clock();
  for(int i=0;i<calltime;i++)
    staticarray();
  end = clock();
  double statictime = (double)(end - start)/CLOCKS_PER_SEC;
  printf("Static Array Function: %.4f seconds\n", statictime);

  start = clock();
  for(int i=0;i<calltime;i++)
    stackarray();
  end = clock();
  double stacktime = (double)(end - start)/CLOCKS_PER_SEC;
  printf("Stack Array Function: %.4f seconds\n", stacktime);

  start = clock();
  for(int i=0;i<calltime;i++)
    heaparray();
  end = clock();
  double heaptime = (double)(end - start)/CLOCKS_PER_SEC;
  printf("Heap Array Function: %.4f seconds\n", heaptime);

  return 0;
}
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10000

void staticarray(){
    static int staticarr[SIZE];
    for(int i=0;i<SIZE;i++)
        staticarr[i] = 1;
}

void stackarray(){
    int stackarr[SIZE];
    for(int i=0;i<SIZE;i++)
        stackarr[i] = 1;
}

void heaparray(){
    int* heaparr = (int*)malloc(SIZE*sizeof(int));
    for(int i=0;i<SIZE;i++)
        heaparr[i] = 1;
    free(heaparr);
}

int main(){
    clock_t start;
    clock_t end;
    int calltime = 200000;
    start = clock();
    for(int i=0;i<calltime;i++)
        staticarray();
    end = clock();
    double statictime = (double)(end - start)/CLOCKS_PER_SEC;
    printf("Static Array Function: %.4f seconds\n", statictime);

    start = clock();
    for(int i=0;i<calltime;i++)
        stackarray();
    end = clock();
    double stacktime = (double)(end - start)/CLOCKS_PER_SEC;
    printf("Stack Array Function: %.4f seconds\n", stacktime);

    start = clock();
    for(int i=0;i<calltime;i++)
        heaparray();
    end = clock();
    double heaptime = (double)(end - start)/CLOCKS_PER_SEC;
    printf("Heap Array Function: %.4f seconds\n", heaptime);

    return 0;
}
```

2) snapshot of each function time



```
Static Array Function: 1.0090 seconds
Stack Array Function: 1.0230 seconds
Heap Array Function: 1.3520 seconds
```

3) explain why you get this results.

Look at the 2)'s snapshot time, I can see that it is a static array < stack array < heap array. This means that the static array is the fastest, then next is the stack array, and finally the heap array is the slowest.
I think the reasons for these results are as follows. The static array is allocated memory at the start of the program through the static keyword and is deallocated at the end of the program. That is, there is no overhead for memory allocation/deallocation that occurs every time a function is called because the allocation is made only once and the allocation is deallocated at the end of the program. However, the stack array is allocated to the stack memory when calling the function and is automatically released at the end of the function. The stack memory allocation/deallocation speed is fast, but it is slower than the static array because there is some overhead that occurs in the process of creating and releasing the stack frame whenever a function is called. Finally, the heap array is the slowest because dynamically allocating/deallocating(malloc/free) must be executed every time a function is called, resulting in additional overhead.