# 2019 Mock

1.

a) **Strings** are character arrays. If a character is 1 byte and the string has 11 (visible) characters, the size of “Hello World” is 11 + 1 (additional 1 because of ‘\0’ at the end of all strings).

b) 18 Mark code-writing question? Hell no!

2.

a) **Void pointers** can be used for general operations that can be specified.

b) **Stack** is automatically managed by compiler and size of every variable must be known before executing program. **Heap** is managed explicitly (by using malloc() to request memory and free() to deallocate it)

c) **Pass-by-value** – the value of a variable is copied into a new variable (a new address) in order not to change the original. Arrays are **passed-by-copy** by copying their memory address, which means that changing that array in any scope changes it globally.

d) **Segmentation fault** raised by hardware notifying OS that program has attempted to access a restricted area of memory. Strategy to find responsible part – debugging with the backtrace feature, which shows call stack at time of the error. Most common cause – dereferencing a NULL pointer.

e) **Ownership** – identify a single entity to be responsible for managing a location in memory. Benefits over malloc() and free(): common problems of manual memory management are avoided by construction, i.e., double-free errors, memory leaks.

RAII ties management of memory to the lifetime of a variable on the stack. Constructor – a function which allocates resource (calls malloc()). Destructor – function that deallocates resource (calls free()). A variable of this struct type on the stack will allocate the memory when it is constructed and automatically deallocate it when the variable goes out of scope.

f) **std::unique\_ptr** models unique ownership when it is possible to have a unique owner of the resource that is responsible for managing its lifetime. For example, children of a node in a tree will automatically be deallocated when their parent node is deallocated.

**std::shared\_ptr** models shared ownership when it is not possible to have a unique owner of the resource. *For example, a graph without cycles (e.g., DAG) where multiple nodes can have multiple incoming edges. A node is automatically deallocated when the last incoming edge disappears, but not before*. For example, multiple threads can be responsible for operations on a list, which will only be deallocated when the last thread has ended.

g)

struct node {  
 std::string label;  
 std::vector<std::shared\_ptr<struct node>> edges;  
};

Outgoing edges of a node are stored in a vector (an array that can dynamically grow or shrink). As there is no unique ownership, a shared pointer is used to model shared ownership.

Design is problematic if there are cycles in the graph as then the shared\_ptr forms a cycle as well, and no memory is released.

An implementation is C would need to ensure that all nodes in the graph are freed after that are no longer accessible. For this, a counter could be maintained in the node that counts the incoming edges. Once that reaches zero, the node can be freed.

3.

a) **Thread** – independent sequence of program instructions. Multiple threads can be executed simultaneously. They share the same address space of a single **process**. OS ensures that address spaces of processes are protected from each other.

b) Simultaneous removal of elements from a linked list: a->b->c->d->NULL. Two threads simultaneously start removing elements from the list.

The first thread removes b by moving pointer of a to c: a--->c->d->NULL.

The second removes c by moving pointer of b to d.

The resulting list still contains node c (even though second tried to remove it).

c) **Future** – handle representing a value that is not yet computed. This handle can be passed around or stored in memory. Once the value is required, .get() can be called, which waits until the value has been computed.

**Promise** – allows to provide a value once it has been computer.

Without future and promise, the value would have to be explicitly protected by a mutex (or another low-level primitive) and a condition variable that could be used to wait for the value to be computed.

d) Iterators pointing to an element could be invalidated by another thread by removing/adding elements.

Scenarios:

* A call to find returns an iterator, but immediately after the call the corresponding element is removed from the set, so that the iterator is now invalid.
* Calls to find and end test if a value is in the set and, only if this is the case, a call to insert should add the value. This is impossible to achieve with this interface as the mutex is released between the individual operations and allows for the tested property (that the value is not in the set) to become false.

e) Ew code

# 2019

1.

a) “**data types give bits meaning**” – since bits patterns can have many different meanings based on interpretation, data types assign a fixed meaning to each bit pattern and the type safety of the language ensure that this meaning is not lost when calculations are performed. For example, the addition operation will result in different instructions and modification of bit patterns depending if an int or a float is modified.

b) Code, ew

2.

a) Forbidden to **dereference a void pointer** since it holds an address to a value of unknown data type, i.e., it is unclear what the bits at this memory location mean. The programmer first has to re-establish the meaning of the bits by casting the pointer.

b) The **stack** is the automatically managed area. Variables are put on the stack when a block is entered and removed in reverse order when the block is exited.

The **heap** is managed manually by the programmer who is responsible to call free correctly.

c) **Lifetimes**:

* automatic (stack) – deallocated at the end of the block they are declared in
* static – deallocated at the end of the program
* allocated – managed manually by the programmer

d) Dangerous to **return a pointer to a local variable** from a function because the pointer has a longer lifetime than the variable. Any access to the original variable will be invalid as it has been (potentially) deallocated.

e) Challenges of malloc and free:

* Double free errors
* Dangling pointers
* Memory leaks
* ???

f) With **ownership** we identify a single entity which is responsible for managing a resource. In C++ this is enforced via **Resource Allocation Is Initialization**: resources are allocated in the constructor of an object and deallocated in the destructor of the object. By creating such an object handler on the local stack, the management of the resource is tied to the lifetime of a stack variable.

g) Code.

3.

a) **Concurrency** is a programming paradigm about **dealing** with lots of things at once.

**Parallelism** is about making programs faster by doing lots of things at once.

b) Mutual exclusion which is, e.g., ensured via mutexes (but possibly also via other related mechanisms such as semaphores).

c)

i) No, this program is not safe, as the buffer is unprotected, but two threads simultaneously access it. It is possible that bufferPtr->pop can be called on an empty buffer.

ii) This program might crash or result in a buffer with (almost) arbitrary size.

d) Code, ew

# 2021

1.

a)

i. int a = 10; (4 bytes)

ii. char b = ‘C’; (1 byte)

iii. double c = 255.1234567; (8 bytes). This sort of precision seems to be important, possibly for future operations with the variable, so precision should be kept.

iv. name = 9 bytes (8 characters + ‘\0’); int = 4 bytes. Total = 13 bytes

b)

i)

21. lex = malloc(sizeof(struct A));

Given in Docs:

freeNodes(struct A\* node) {  
 if(node == NULL) return;  
 freeNodes(node->right);  
 free(node);  
}

26. freeNodes(lex);

ii)

12. !\*l;  
13. \*l = li;

Text

Description automatically generated

iii) Adds values 1, 2 and 3 to the doubly linked list. The newL function creates the node for each value, and the addL function adds the newly created node to the list. The entire linked list is then printed on line 25.

Diagram consists of nodes that point to each other (both in front of them and behind them, as it is a doubly linked list). The end of the list does not point to the front of the list.

2.

a)

i)

|  |  |  |
| --- | --- | --- |
|  | Version 1 | Version 2 |
| Stack | i | i |
|  | sum | j |
|  | arr | sum |
|  |  |  |
| ---------------------------------------- | ---------------------------------------- | ----------------------------------------- |
|  |  |  |
|  |  |  |
|  | pt | pt |
| Heap | n | n |

ii) Array of structs means that all three values would be very close to each other in memory and possibly the same data block, in which case version 1 would be more cache efficient.

b)

i. Double-free. An already freed pointer is freed again.

ii. Dangling pointer. A pointer points to an already freed memory address.

iii. Memory leak. Impossible to free the first malloc’d address.

c)

Text

Description automatically generated

The RAII version is better as we tie the management of a resource to the lifetime of a variable on the stack.

3.

a) Issue 1: The lock should be acquired before find() is used on the table, without this it is not thread-safe.  
Issue 2: The mutex is unlocked automatically by applying RAII in C++, no need to unlock it explicitly.

b) Graphical user interface, application

Description automatically generated

c) Mutexes, since semaphores require thinking about order of threads, but there is no order necessary in this application.

d)

i. Text

Description automatically generated

ii. Graphical user interface, text, email

Description automatically generated

[LARGE CHUNKS OF CODE NOT IN 2022 EXAM]