Part I — Recognition (20)

1. A variable's **lifetime** is
   a) How long its name is visible in the source
   b) The time from allocation until deallocation of its storage
   c) The time it stays in a register
   d) The time it is cached by CPU
2. A variable's **scope** is
   a) The textual region where its binding is visible
   b) The OS process lifetime
   c) The same as lifetime
   d) The stack frame where it's stored
3. In C/C++, **global** objects (with external linkage) are allocated in
   a) Stack segment
   b) Static storage (data/BSS)
   c) Heap
   d) Register file
4. A **static local** (e.g., `static int z;`) has
   a) Stack storage, lifetime = call
   b) Static storage, lifetime = whole program
   c) Heap storage, lifetime = call
   d) Static storage, lifetime = function call only
5. An object from `malloc/new` has
   a) Stack storage, lifetime = block
   b) Heap storage, lifetime = until `free/delete`
   c) Static storage, lifetime = program
   d) Heap storage, lifetime = function call
6. A **dangling reference** happens when
   a) Two names alias the same cell
   b) A pointer refers to already-freed storage
   c) A pointer is `NULL`
   d) A global shadows a local
7. A **memory leak** happens when
   a) You write past array bounds
   b) You lose the last reference to allocated heap memory
   c) You allocate on stack
   d) You use `static`
8. An **activation record (stack frame)** typically contains
   a) Only machine code
   b) Return address, parameters, locals, saved registers, links
   c) Only global variables
   d) Only heap pointers
9. The **dynamic link** in a frame points to
   a) Callee's frame

b) Caller's frame (for return/unwind)
c) Global table
d) Heap block header

10. In lexically nested languages, the **static link/access link** enables
    a) Jump to caller
    b) Access to variables in enclosing lexical scopes
    c) Faster malloc
    d) Garbage collection
11. **Static (lexical) scope** resolves names using
    a) The call chain at runtime
    b) The source nesting at compile time
    c) The loader's symbol map
    d) OS environment variables
12. **Dynamic scope** resolves names using
    a) Source nesting only
    b) The most recent active binding along the call chain
    c) The global table only
    d) The import list only
13. The **referencing environment** of a statement under static scope is
    a) Locals only
    b) Locals + visible bindings in all enclosing blocks
    c) Globals only
    d) All active frames
14. The **referencing environment** under dynamic scope is
    a) Locals + all visible bindings in all active subprograms
    b) Locals only
    c) Globals only
    d) Imports only
15. In C/C++, a pointer returned to a **stack variable's address** is
    a) Valid if `const`
    b) Undefined (dangling)
    c) Valid until next call
    d) Valid for the program
16. In C/C++, a `static` data member of a class lives in
    a) Each object's subobject
    b) Static storage (one per class)
    c) Stack of each method call
    d) Heap per object
17. In row-major languages, parameters are passed and a new AR is pushed
    a) When the ELF is linked
    b) At each function call (on invocation)
    c) At compile time
    d) Only for recursive calls
18. **Early binding** vs **late binding** primarily distinguish
    a) Stack vs heap

b) Compile-time vs run-time resolution
c) Static vs dynamic storage duration
d) Public vs private

19. In C, a function parameter `int y` typically has
    a) Static storage
    b) Automatic (stack) storage
    c) Heap storage
    d) Register storage only
20. If a closure captures a local variable, its lifetime is
    a) Still limited to the call
    b) Extended (boxed/heap cell) as long as closure exists
    c) Moved to global section
    d) Ignored by runtime

Part II — Application (20) — Lifetime/Allocation & Outputs

21. Consider C++:

```
class MyClass { public: int a; static int b; }; int MyClass::b = 0; MyClass*
createObject() { MyClass obj1; MyClass* obj2 = new MyClass; static MyClass obj3; return
obj2; }
```

Which statement is correct about **memory regions**?
a) obj1 heap, obj2 stack, obj3 data segment
b) obj1 stack, obj2 stack, obj3 data segment
c) obj1 stack, obj2 heap, obj3 data segment
d) obj1 heap, obj2 heap, obj3 stack

22. Given C:

```
int x; void foo(int y){ static int z; int *t = malloc(sizeof(int)); // ... }
```

Which statement is **WRONG** about **lifetime**?
a) `x` lives for the whole program
b) `y` lives for the duration of `foo` call
c) `z` lives for the duration of `foo` call
d) `*t` lives until `free(t)` (or process ends)

23. With the same code, which statement is **WRONG** about **allocation**?
    a) `x` in static storage
    b) `y` in stack
    c) `z` in static storage
    d) `t` in heap and auto-freed at end of `foo`

24. C++:

```
void foo(int a, int *b, int &c, int *arr) { a += *b; *b *= 2; c += a; arr[0] = 42; arr = new int[2]{7,8}; } int main() { int x=5, y=3, z=10; int arr[2]={1,2}; int *par = arr; foo(x,&y,z,par); printf("%d %d %d | {%d,%d} | par-> {%d,%d}\n", x,y,z, arr[0],arr[1], par[0],par[1]); }
```

What prints?

a) `5 6 18 | {42,2} | {7,8}`
b) `5 5 10 | {42,2} | {1,2}`
c) `5 3 18 | {1,2} | {42,2}`
d) `5 6 18 | {1,2} | {7,8}`

25. C++:

```
int *p = new int(5); int *q = p; delete p; *q = 7; cout << *q;
```

Result is
a) Prints 7
b) Prints 5
c) Undefined behavior (dangling write)
d) Compile error

26. C++:

```
int *p = new int(1); p = nullptr;
```

Which is true?
a) Safe; no leak occurs
b) Leak (original address lost)
c) Double free
d) UB only if later dereferenced

27. C (assume typical calling convention):

```
int g = 1; void A(){ int g = 2; } void B(){ printf("%d", g); } int main(){ A(); B(); }
```

Output is
a) 1 b) 2 c) UB d) compile error

28. Python (static scope):

```
x=1 def outer(): x=10 def inner(): return x+1 return inner() print(outer(), x)
```

Output
a) 11 1 b) 11 10 c) 2 1 d) NameError

29. Python lifetime/escape:

```
def make(): bag=[] def add(v): bag.append(v) return sum(bag) return add f=make()
```

```
print(f(3), f(4), f(1))
```

a) 3 4 1 b) 3 7 8 c) 3 7 4 d) 8 7 3

30. C++:

```
int *bad(){ int x=5; return &x; } int *good(){ return new int(5); }
```

Which is correct?
a) Both OK
b) `bad` returns dangling; `good` OK (must delete)
c) Both dangling
d) Both leak always

31. C++:

```
const string& r = string("hi"); cout << r << '\n';
```

Which is correct?
a) Always dangling
b) Lifetime is extended for const lvalue ref to temporary → safe
c) Moved to static storage
d) UB at link time

32. C++:

```
int a=2,b=3; int &r=a, &s=r; s=b; cout<<a;
```

Output
a) 2 b) 3 c) UB d) compile error

33. C++:

```
int *p=new int(5); delete p; delete p;
```

a) Defined no-op
b) Undefined behavior (double free)
c) Compile error
d) Silently leaks but defined

34. Python (import precedence last):

```
from math import sqrt sqrt = lambda z: z+1 x=3 def f(): return sqrt(x) print(f())
```

a) 2.0 b) 4 c) NameError d) 3

35. Pseudo-Pascal — **what prints under static vs dynamic scope?**

```
var a: integer := 1; procedure P; var a: integer := 2; procedure Q; begin writeln(a) end;
```

```
begin Q end; begin P end.
```

a) Static: 2 ; Dynamic: 1
b) Static: 1 ; Dynamic: 2
c) Both 2
d) Both 1

36. Pseudo-Pascal:

```
var a: integer := 1; procedure P; var b: integer := 2; procedure Q; begin writeln(a+b)
end; begin Q end; begin P end.
```

Under **dynamic** scope (not real Pascal), Q prints
a) 3 b) 2 c) 1 d) depends on call site

37. Pseudo-Pascal:

```
var x: integer := 5; procedure A; var x: integer := 7; procedure B; begin writeln(x) end;
begin B end; begin A end.
```

Under **static** scope ⇒ prints … ; under **dynamic** scope ⇒ prints …
a) 7 ; 5 b) 5 ; 7 c) 7 ; 7 d) 5 ; 5

38. C++ (lifetime categories):
    Pick the **incorrect** mapping.
    a) global → static storage, lifetime=program
    b) local automatic → stack, lifetime=call
    c) `new` object → static storage, lifetime=program
    d) `static` local → static storage, lifetime=program

39. C (frame content):
    Which is **not** typically stored in an activation record?
    a) Return address
    b) Saved registers
    c) Heap free list
    d) Parameters

40. Python (closure escape & lifetime):

```
def counter(): n=0 def inc(): nonlocal n; n+=1; return n return inc c=counter()
print(c(), c(), c())
```

a) 1 1 1 b) 1 2 3 c) 0 1 2 d) 2 3 4

Part III — Advanced Application (20) — Longer Case Studies & Static vs Dynamic Scope

41. **Static vs Dynamic scope (sum updates):**

```
var A,B: integer; procedure P(X: integer); var B: integer; procedure Q(Y: integer); begin A := A + Y; B := B + X end; begin B := 2; Q(3); writeln(A, B) end; begin A := 1; B := 5; P(4) end.
```

What prints inside P?
a) Static: `4 6` ; Dynamic: `4 9`
b) Static: `4 9` ; Dynamic: `4 6`
c) Both `4 6`
d) Both `4 9`

42. **Output trace (C++, mixed parameters & heap):**

```
void foo(int a, int* b, int& c, int*& par) { a += *b; *b += 1; c -= a; par = new int[3] {a, *b, c}; } int main(){ int x=4, y=5, z=20; int arr[3]={0,0,0}; int* p=arr; foo(x,&y,z,p); printf("x=%d y=%d z=%d | arr={%d,%d,%d} | p={%d,%d,%d}\n", x,y,z,arr[0],arr[1],arr[2],p[0],p[1],p[2]); }
```

a) `x=4 y=6 z=11 | {0,0,0} | {9,6,11}`
b) `x=4 y=6 z=11 | {9,6,11} | {9,6,11}`
c) `x=5 y=5 z=??` (UB)
d) `x=4 y=6 z=16 | {0,0,0} | {10,6,16}`

43. **Dangling misuse after free (C++):**

```
int *p=new int(5); int &r=*p; { int *q=p; *q=6; delete p; } cout<<r<<'\n';
```

a) reliably prints 6
b) undefined behavior (dangling)
c) prints 5
d) compile error

44. **Python — deep closure/AR chain:**

```
def outer(a): b=2 def mid(c): nonlocal b b += a + c def inner(d): return a + b + c + d return inner f = mid(3) print(f(4))
```

Output
a) 10 b) 12 c) 14 d) 16

45. **Lifetime question (C++):**

```
int* bad(){ int x=5; return &x; } int* good(){ return new int(5); }
```

Choose the best statement.
a) `bad` ok; `good` ok
b) `bad` dangling; `good` ok (must delete)
c) both dangling
d) both leak even if deleted

46. **Static vs dynamic reading shadowed `x`:**

```
var x: integer := 1; procedure A; var x: integer := 2; procedure B; begin writeln(x) end;
begin B end; begin A end.
```

a) Static→2 ; Dynamic→1
b) Static→1 ; Dynamic→2
c) Both→2
d) Both→1

47. **C++ recursion + heap lifetime:**

```
int sumUp(int n){ if(n<=1) return 1; int *p=new int(n); int r=*p + sumUp(n-1); delete p;
return r; } int main(){ cout<<sumUp(3); }
```

Prints
a) 3 b) 4 c) 6 d) 7

48. **Python — star args & kwargs inside nested default:**

```
def K(a=1,*args,**kw): def M(b=2,*t,**m): return a + b + len(t) + len(m) return M(3, 7,
8, x=9, y=10) print(K())
```

a) 5 b) 6 c) 7 d) 8

49. **Dynamic scope thought experiment**
In a dynamic-scoped language, with call chain `Main→S1→S2` and each declares `x`, a read of `x` inside `S2` binds to
a) Main.x b) S1.x c) S2.x d) global.x

50. **Python — closure list mutation persists (lifetime evidence):**

```
def make(): bag=[] def add(v): bag.append(v) return bag[:] return add f=make()
print(f(2), f(5), f(7))
```

a) [2] [5] [7]
b) [2] [2,5] [2,5,7]
c) [2] [5] [2,7]
d) [2] [2] [2]

51. **Union punning (C/C++):**

```
union U{ unsigned i; float f; }; U u; u.i=0x3F800000; cout<<u.f;
```

Which is most accurate?
a) Always 1.0 by the standard across languages
b) Implementation/aliasing dependent; often 1.0 on many ABIs
c) Always 0.0
d) Compile-time error in C++

52. **Python — global write then read via inner:**

```
x=1 def f(): global x; x = x + 2 def g(): return x return g() print(f(), x)
```

a) 3 1 b) 3 3 c) 2 3 d) Error

53. **Referencing environment summary**
    Pick the correct statement.
    a) Static RE = locals + lexically enclosing visibles; Dynamic RE = locals + all active frames' visibles
    b) Both REs = locals + globals only
    c) Static RE = activation record contents
    d) Dynamic RE ignores parameters

54. **Python — nonlocal ladder shows lifetime extension:**

```
def T(): a=0 def U(): nonlocal a; a+=1 def V(): nonlocal a; a+=2 def W(): nonlocal a;
a+=3 return a return W() return V()+a return U()+a print(T())
```

a) 3 b) 4 c) 6 d) 7

55. **C++ — heap without delete:**

```
int* mk(int v){ return new int(v); } int main(){ int* q=mk(4); cout<<*q; /* no delete */
}
```

a) Prints 4, leaks if not deleted
b) Freed automatically at scope end
c) Compile error
d) Undefined

56. **Name resolution order (Python):** locals → nonlocals → globals → builtins/imports. For

```
from math import sqrt sqrt = lambda z: z*z x=3 def f(): return sqrt(x) print(f())
```

a) 9 b) 1.732… c) NameError d) 3

57. **C++: global vs local with same name**

```
int g=1; void A(){ int g=2; } void B(){ cout<<g; } int main(){ A(); B(); }
```

What concept explains the output?
a) Dynamic scope
b) Static scope & different bindings (local hides only inside A)
c) Heap lifetime
d) RAII

58. **Python — higher-order closure shows lifetime:**

```
def power(k): def p(x): return x**k return p sq=power(2); cu=power(3) print(sq(3), cu(2))
```

a) 6 8 b) 9 8 c) 9 6 d) 8 9

59. **AR links**
When inner function accesses an outer local, implementation typically uses
a) Static link / closure environment to reach enclosing frame/cell
b) Only dynamic link
c) Only symbol table
d) Only global register

60. **Python — mixed nested with captured update (determine output):**

```
x=10 def F(): x=1 def G(y): nonlocal x x+=y def H(): return x + y return H() return
G(2)+x print(F(), x)
```

a) 5 10 b) 3 10 c) 4 1 d) Error

Answer Key

1 b
2 a
3 b
4 b
5 b
6 b
7 b
8 b
9 b
10 b
11 b
12 b
13 b
14 a
15 b
16 b
17 b
18 b
19 b
20 b
21 c
22 c
23 d
24 a
25 c
26 b
27 a
28 a
29 b
30 b

31 b
32 b
33 b
34 b
35 a
36 a
37 a
38 c
39 c
40 b
41 a
42 a
43 b
44 c
45 b
46 a
47 d
48 d
49 b
50 b
51 b
52 b
53 a
54 d
55 a
56 a
57 b
58 b
59 a
60 a