

# CS101A DATA STRUCTURE(H)

## QUIZ [1] C++ BASICS

NAME: \_\_\_\_\_

STUDENT ID: \_\_\_\_\_

EMAIL: \_\_\_\_\_

Start Time: 20:15 PM

End Time: 20:30 PM

### PART I. MULTIPLE CHOICES I - Completely fill the circles as shown: ○○●○

```
class Buf {
    std::size_t n_; int* d_;
    static inline std::size_t live_ = 0;
public:
    explicit Buf(std::size_t m): n_(m), d_(new int[m]{} ) { ++live_; }
    Buf(const Buf&) = delete;
    Buf& operator=(const Buf&) = delete;
    Buf(Buf&& o) noexcept : n_(o.n_), d_(o.d_) { o.n_=0; o.d_=nullptr; ++live_; }
    Buf& operator=(Buf&& o) noexcept {
        if (this != &o) { delete[] d_; n_ = o.n_; d_ = o.d_; o.n_=0; o.d_=nullptr; }
        return *this;
    }
    ~Buf(){ delete[] d_; --live_; }
    int& at(std::size_t i){ if(i>=n_) throw; return d_[i]; }
    const int& at(std::size_t i) const { if(i>=n_) throw; return d_[i]; }
    int* begin() noexcept { return d_; }
    int* end()    noexcept { return d_ + n_; }
    const int* begin() const noexcept { return d_; }
    const int* end()    const noexcept { return d_ + n_; }
    static std::size_t live() noexcept { return live_; }
};
```

**Q. 1** You expose `begin()/end()` so clients can iterate `Buf`. Judge the statements below. (10 pts)

- ☐ Dereferencing `end()` is undefined, even though it is a valid one-past-the-end pointer.
- ☐ Pointer arithmetic like `begin()+k` is defined only if the result remains in `[begin(), end())`.
- ☐ Using `at(i)` provides checked access, while `begin()[i]` requires the caller to ensure `i<n`.
- ☐ For `const` objects, dedicated `const` overloads of `begin()/end()` aren't needed; the non-`const` versions can still be called and return writable iterators.

**Q. 2** You want a robust interface for `Buf`. Judge the statements about the object model.(7.5 pts)

- ☐ The `const` overload of `at` allows modifying elements through a `const Buf&`.
- ☐ Since `live_` appears before `public:` label, it is `private`.
- ☐ Exposing `n_` and `d_` as public would risk representation corruption.

**Q. 3** `Buf` is intended for `std::vector<Buf>`. Judge the statements below.(7.5 pts)

- ☐ Marking move operations `noexcept` guarantees containers always move.
- ☐ A moved-from `Buf` being `n_=0`, `d_=nullptr` makes destruction safe.
- ☐ Because `Buf` manages a raw resource and defines a destructor, it should explicitly declare all five special members (destructor, copy constructor/assignment, move constructor/assignment).This implementation does so by deleting copy and providing move.

## PART II. MULTIPLE CHOICES II

```
class IUser {
public:
    virtual ~IUser() = default;
    virtual int maxBooks() const = 0;
    virtual bool isAdmin() const = 0;
};

class Member : public IUser {
    int cap_ = 2;
public:
    int maxBooks() const override { return cap_; }
    bool isAdmin() const override { return false; }
};

class VIPMember : public Member {
public:
    int maxBooks() const override { return 5; }
};

class Admin : public IUser {
    Admin() = default;
public:
    static Admin create() { return Admin(); }
    int maxBooks() const override { return 0; }
    bool isAdmin() const override { return true; }
};
```

**Q. 1** You want a uniform handle to different user roles. Judge the statements about `IUser` and its implementations. **(7.5 pts)**

- ☐ `IUser` is abstract because it declares at least one pure virtual function.
- ☐ `Member`, `VIPMember`, and `Admin` can be stored as `std::unique_ptr<IUser>` to avoid slicing.
- ☐ Calls through `IUser&` bind to the most specific override of `maxBooks()` in the dynamic type.

**Q. 2** Consider the roles' behaviors and construction rules. Judge the statements about overriding and construction. **(10 pts)**

- ☐ `VIPMember::maxBooks()` overrides the inherited interface and specializes the capacity.
- ☐ Making `Admin`'s constructor private and exposing `Admin::create()` prevents uncontrolled construction.
- ☐ You can pass a `VIPMember` wherever an `IUser&` is expected; calls through the `IUser` interface still invoke the `VIPMember` overrides.
- ☐ Declaring `isAdmin()` non-virtual in `IUser` would still allow overrides in derived classes.

**Q. 3** You need `auditBorrowing` to be easy to change and test, without depending on concrete roles or storage details. Choose the statements that best reflect simple, practical design. **(7.5 pts)**

- ☐ Define a small `IAuditPolicy` (e.g., `bool canBorrow(const IUser&, int currentlyBorrowed)`), and let the auditor depend on this interface so different policies can be swapped in.
- ☐ Hardcode the numeric limits (e.g., 2 or 5) inside for all roles to keep the code shorter.
- ☐ Read and write global variables for audit thresholds so configuration is shared across the app without passing parameters.