

EC327 Spring 2025 Final Exam Instructions

Please write your name on the back of this exam. This will remove unconscious bias during the grading of the exam as the graders will not see names while grading.

This is a closed book exam, absolutely no technology, books, or notes are allowed to be consulted while taking the exam. Violating this rule will result in a 0 grade for the exam.

When asked to write code, you will not be evaluated on exact syntax, rather just the core concepts/ideas.

The exam will be graded out of 100 points and consists of the following problems:

1. True or False, General Knowledge (**14 points**)
2. Multiple Choice, General Knowledge (**12 points**)
3. Templates (**12 points**)
4. Inheritance & Function Invocation Dispatch (**12 points**)
5. Standard Template Library (**20 points**)
6. Android Development (**20 points**)
7. Composition & Inheritance (**10 points**)

True or False - General Knowledge (14 points - 2 points each)

(Whether you respond with True or False, please explain your response in a sentence). You will get 1 point if your True/False response is correct and another if your sentence clearly provides an explanation.

1. Dependency injection eliminates the need for polymorphism in object-oriented design.
2. Event-driven programs in Android typically follow a top-down control flow like in procedural C code.
3. In object oriented C++ code, stack-allocated objects require manual deletion to avoid memory leaks
4. In Scrum, the purpose of Story Point planning is coming up with a detailed plan for implementing the feature.
5. In Android development, UI elements can be freely accessed and updated from any background thread without any additional synchronization.
6. In C++, dynamic dispatch requires that the function being dispatched is declared as virtual

7. In object-oriented programming, it is generally a good idea to allow child classes to directly access private fields of their parent class.

Multiple Choice (12 points - 2 points each)

1. Which of the following is a benefit of dependency injection?
 - a. Tightly couples components for faster access
 - b. Makes code harder to test
 - c. Allows behavior to be swapped out at runtime
 - d. Alleviates the need for manual memory management

2. In Kanban, which of the following best describes how work is assigned?
 - a. The project manager assigns tasks at the start of the sprint
 - b. Developers pull tasks from the backlog continuously
 - c. Tasks are assigned based on component ownership
 - d. Tasks are assigned roughly 6 weeks in advance after a lengthy planning session

3. In Scrum development, what is a “story point”?
 - a. The product requirements for a feature from a user’s perspective
 - b. A presentation on the evolution of requirements for a feature.
 - c. A detailed description of the implementation plan of a feature
 - d. A very rough estimate of the complexity and implementation time for a feature

4. In a continuous integration (CI) pipeline, which of the following are most appropriate times to trigger test suite execution?
 - a. When code is merged into the main branch
 - b. Once a week at midnight
 - c. When a Pull Request is created
 - d. A and C

5. In C++, what is the purpose of the override keyword?
 - a. Allow multiple constructors for a class
 - b. Indicate that a function overloads another function
 - c. Indicate that a function overrides a virtual function from a base class
 - d. Make a function non-virtual

6. What is a key difference between `std::map` and `std::unordered_map` in C++?
- a. `std::unordered_map` guarantees $O(\log N)$ lookups
 - b. `std::map` is backed by a hash table, `std::unordered_map` by a balanced tree
 - c. `std::map` maintains values in sorted order, `std::unordered_map` does not
 - d. `std::map` maintains keys in sorted order, `std::unordered_map` does not

Problem 3 : Templatized Class (12 points)

A Stack is a very useful container data structure that supports Last In - First Out behavior. You can picture a stack of dishes where the dish placed on top of the stack most recently is the dish that will get picked off of the stack next. This data structure is even more powerful if it can be reused across all data types. In other words, we should not need to implement the Stack for integers and then re-implement it for strings.

Your stack should support

- **push** - adds an element to the top of the stack
- **pop** - removes an element from the top of the stack
- **top** - returns the element at the top of the stack without removing it
- **isEmpty** - returns **true** if the Stack contains no elements
- **contains** - takes an element of the generic type and returns true if an element exists in the stack that is **==** to the passed in element. It is OK to assume that all generic types used for this Stack override the **==** operator.

Your implementation should use the **std::vector** container from the **STL**.

Problem 3.1 Declaration & Implementation (9 points)

Provide the **stack.h** declaration & implementation for your generic Stack. Both can be in the **.h** file to avoid templated compilation challenges.

Problem 3.2 Performance (3 points)

What are the Big-O running times for your **push**, **pop**, and **contains** implementations?
(Provide a brief explanation for each.)

Problem 4: Inheritance & Function Invocation Dispatch (12 points)

You are building a notification system that can send different kinds of notifications: email notifications and SMS (text) notifications, to start.

There is a base class **NotificationSender** that defines a **send()** method.

Each specific type of notification will inherit from NotificationSender and implement its own version of send().

Problem 4.1: Abstract Class (3 points)

Let's start by implementing the base class - **NotificationSender**.

It should have a void **send** function that takes a single **std::string** argument with a default implementation that simply prints a message about the base sender sending the specific message.

It should be declared in a way that causes dispatching to route invocations to its descendant classes if those are present at runtime.

It should also have a virtual destructor.

Write your **notificationsender.h** here:

Problem 4.2: Implement SMSSender (2 points)

Now provide the implementation of **SMSSender**. This class should inherit from **NotificationSender** and provide its own implementation of the **send** function. For the purposes of the exam, the implementation can simply print out a message about SMSSender sending the specific passed in message.

SMSSender does not need to take any constructor arguments.

For the purposes of the exam, the entire implementation can be provided in **smssender.h** below:

Problem 4.3: Implement EmailSender (3 points)

Now, let's move onto implementing an email sender. It is going to inherit from **NotificationSender** and be similar to the **SMSSender** that you just implemented.

However, it has a private pointer to a server address and initializes this pointer via a constructor argument. It is its responsibility to clean up this memory address when it is destroyed.

For the purposes of the exam, the implementation of **send** can simply print out a message about EmailSender sending the specific passed in message.

For the purposes of the exam, the entire implementation can be provided in **emailsender.h** below:

Problem 4.4: Invocation (4 points)

Answer the questions below with the following code snippet in context:

```
int main() {  
    std::string* emailsServer = new std::string("localhost");  
  
    NotificationSender* sender = new EmailSender(emailsServer);  
    sender->send("hi there");  
  
    delete sender;  
  
    return 0;  
}
```

- When the **main** function is executed, what is printed out?
- Explain how the **send** function invocation is routed.
- Explain what destructor is invoked when the **main** function completes and how that invocation is routed.

Problem 5: Standard Template Library (20 points)

For this problem, you will be designing a class that keeps track of all users of a messaging application that you are building. The **User** class has been defined for you as:

```
#include <string>

class User {
private:
    std::string username;
    int ageInDays;
public:
    User() {};

    User(std::string username, int ageInDays) :
        username(username), ageInDays(ageInDays) {}

    std::string getUsername() const {
        return username;
    }

    int getAge() const {
        return ageInDays;
    }

    std::string display() const {
        return username + " " + std::to_string(ageInDays);
    }
};
```

The core tasks that your **UserManager** will need to support are:

- **bool registerNewUser(const User* user)**
 - asks the **UserManager** to keep track of a new **User**. This method should return **true** if this user has not yet been registered with the **UserManager** and now is; and return **false** if the **UserManager** has seen this **User** before and therefore did not update anything.
 - The implementation of this function should be $O(\log N)$.
- **const User* getUser(std::string username)**

- asks the **UserManager** to retrieve the reference to the **User** with the given **username**. The function should return a null pointer if a user with the given **username** has not been registered with the **UserManager**.
- The implementation of this function should be $O(1)$.
- **const User* getYoungestUserOlderThan(int daysOld)**
 - asks the **UserManager** to retrieve the youngest known user who is older than **daysOld**, inclusive. If there is not a single user older than **daysOld**, inclusive, then this function should return a null pointer.
 - Please keep in mind that you may have multiple users who are the same age.
 - If multiple users are tied for this distinction (i.e. they are the youngest users over **daysOld** and are the same age), returning any one of them is fine.
 - The implementation of this function should be $O(\log N)$.

Problem 5.1: UserManager header (8 points)

In the space below, write your **usermanager.h** declaration:

Problem 5.2: UserManager implementation (6 points)

In the space below, write your **usermanager.cpp** implementation:

Problem 5.3: Implementation Choice - `getUser` (3 points)

The problem statement required that **`getUser`** be $O(1)$. In a paragraph, explain how you accomplished this and how any underlying library or data structure that you used accomplishes this.

Problem 5.4: Implementation Choice - `getYoungestUserOlderThan` (3 points)

The problem statement required that **`getYoungestUserOlderThan`** be $O(\log N)$. In a paragraph, explain how you accomplished this and how any underlying library or data structure that you used accomplishes this.

Problem 6: Android Development (20 points)

You are building a Shopping List application. The goal of the application is to let its users manage their grocery shopping lists, keep track of what they still need to purchase, and be notified of groceries that may be on sale via API calls.

You are starting with this code snippet from the **MainActivity**:

```
public class MainActivity extends AppCompatActivity {
    private static final String PREFS_NAME = "ShoppingPrefs";
    private static final String KEY_ITEMS = "items";

    private EditText itemEditText;
    private Button addButton;
    private TextView shoppingListTextView;

    private List<String> shoppingItems = new ArrayList<>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        itemEditText = findViewById(R.id.itemEditText);
        addButton = findViewById(R.id.addButton);
        shoppingListTextView = findViewById(R.id.shoppingListTextView);

        addButton.setOnClickListener(v -> {
            String item = itemEditText.getText().toString();
            shoppingItems.add(item);
            updateShoppingList();
        });

        loadItems();
    }

    private void updateShoppingList() {
        shoppingListTextView.setText(TextUtils.join("\n", shoppingItems));
    }

    private void loadItems() {
        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
        String items = prefs.getString(KEY_ITEMS, "");
        shoppingItems = new ArrayList<>(Arrays.asList(items.split(",")));
        updateShoppingList();
    }

    private void saveItems() {
        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
        prefs.edit().putString(KEY_ITEMS, TextUtils.join(",", shoppingItems)).apply();
    }
}
```


Problem 6.1: Code Analysis (6 points)

Problem 6.1.1: Event Registration (2 points)

What event is being registered in the **onCreate** method? What behavior is expected when this event is triggered?

Problem 6.1.2: Non Blocking (2 points)

What core Android principle guides us not to perform any long-running or potentially blocking work in the code that is executed when the event is triggered?

Problem 6.1.3: Quick Additions (2 points)

Support the user adds 50 items in quick sessions. Will the current code structure handle this interaction smoothly? Explain why or why not.

Problem 6.2: Persistence (8 points)

Problem 6.2.1: As Implemented (2 points)

As implemented, what is the current persistence strategy for the application? In other words, where/when is data being saved/loading?

Problem 6.2.2: Persistence Shortcomings (2 points)

What are the shortcomings of the current persistence strategy?

Problem 6.2.3: Addressing Shortcomings (2 points)

If you identified any shortcomings above, address them now. You can address them by providing code snippets below or by explaining what you would do in words.

Problem 6.2.4: General Persistence Guidelines (2 points)

In general, what are some persistence guidelines and trade-offs for Android applications that we talked about in class? The code snippet above utilizes **SharedPreferences**. What other on-disk persistence options are you aware of? What are the trade-offs between the options you are familiar with?

Problem 6.3: Server Sync (6 points)

Problem 6.3.1: Triggering Sync (2 points)

Let's say you wanted to evolve your application by allowing users to save their shopping lists to a server that you maintained for this application. You would want to trigger saving to the server periodically and display a green checkmark (all data saved remotely!) when the latest changes were synced. Could you safely add the server call to the block of code inside **addButton.setOnClickListener(v -> { ?** Why or why not?

Problem 6.3.2: Triggering Sync Improvements (2 points)

If your answer above was that making the server call in **addButton.setOnClickListener(v -> {** is not the best strategy, describe a better way for making that call

Problem 6.3.3: Updating the User (2 points)

Building on your responses above, and with the requirement of rendering a green checkmark and a “all data saved remotely!” message, how, when, and by whom would those UI elements be updated in response to the server receiving the latest state of the user’s items?

Problem 7: Polymorphism, Composition & Inheritance (10 points)

You are building a fairly straightforward, two dimensional, Android game. The game involves the main player, some enemies, and some other objects that the player and enemies can interact with. You have defined the following interface that every entity/object that is part of the game is going to implement.

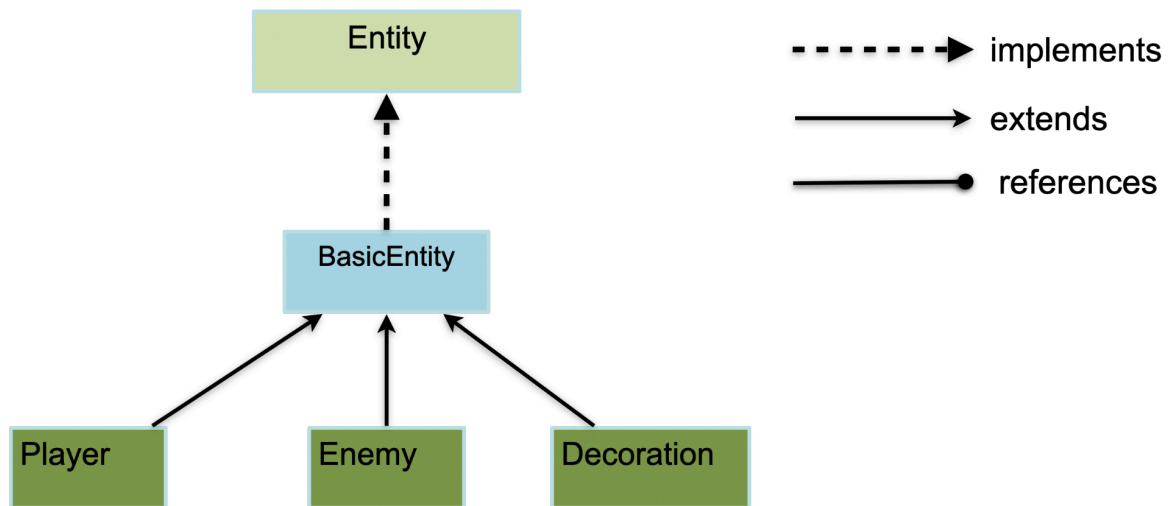
```
/**
 * Defines basics for all objects in our game.
 */
public interface Entity {
    /**
     * Moves the game object along the x and y axes.
     */
    void move(int x, int y);

    /**
     * Renders the entity onto the screen.
     */
    void render(GameScreen gameScreen);

    /**
     * Collide with another entity and optionally return a new resulting
     combined entity.
     */
    Entity collide(Entity otherEntity);
}
```

Problem 7.1: Which one is it? (4 points)

You have started with the following class diagram:



The **BasicEntity** class provides default implementations for **move**, **render**, and **collide**. Its child classes can override those default behaviors as needed.

Problem 7.1.1: Polymorphism (1 point)

Does this structure provide the ability to take advantage of polymorphism?

Problem 7.1.2: is-a or has-a ? (1 point)

Are the relationships between Player-BasicEntity, Enemy-BasicEntity, Decoration-BasicEntity examples of **is-a** or **has-a** relationships? Please explain in a sentence why

Problem 7.1.3: Composition or Inheritance ? (2 points)

Is the diagram above an example of composition or inheritance?

Problem 7.2: An Alternative (2 points)

If your answer above was “This is Composition”, redraw the diagram as it would look like if it were “Inheritance”. If your answer above was “This is Inheritance”, redraw the diagram as it would look like if it were “Composition”.

Problem 7.3: Trade Offs (3 points)

Explain what the pros and cons of composition and inheritance are and when you would favor each.

Problem 7.4: Dependency Injection (1 point)

Explain in which scenario above might dependency injection come into play. How would it be used and what would be its benefit?