# Qt/C++ Object Lifetime Management Exam

## Instructions

This exam tests your understanding of Qt's parent-child object lifetime management model. Answer all questions. Code should be written in modern C++ (C++11 or newer) using Qt 5/6 conventions.

## 1 Theoretical Understanding (20 points)

1. Explain Qt's parent-child ownership model and how it manages object lifetimes. Contrast it with RAII and smart pointers. (5 points)

2. Describe what happens in these scenarios when a QObject parent is deleted:
   (a) A child has Qt::DirectConnection to the parent's signal
   (b) A child has Qt::QueuedConnection to the parent's signal
   (c) A child stores a pointer to its parent

   (5 points)

3. Explain the difference between QObject::deleteLater() and immediate deletion. When would you use each? (5 points)

4. How does Qt handle object trees with multiple inheritance from QObject? What are the limitations? (5 points)

## 2 Implementation (30 points)

1. Implement a QWidget-derived class that:

   - Manages a collection of child QWidgets
   - Properly cleans them up when destroyed
   - Provides methods to safely add/remove children
   - Handles cases where children might be deleted externally

   (10 points)

```
// Your implementation here
```

2. Create a QObject-derived class that maintains a bidirectional parent-child relationship without creating memory leaks. Include:

   - Parent-to-child (strong ownership)
   - Child-to-parent (weak reference)
   - Proper cleanup in all destruction scenarios

   (10 points)

```
// Your implementation here
```

3. Implement a scene graph system where:

   - Each Node is a QObject
   - Nodes can have child Nodes
   - The root Node manages all children's lifetimes
   - Provides a method to reparent subtrees

   (10 points)

```
// Your implementation here
```

## 3 Problem Solving (30 points)

1. Analyze this code and identify all object lifetime issues:

```
class ResourceManager : public QObject {
public:
    QList<QWidget*> resources;

    void addResource(QWidget* res) {
        resources.append(res);
    }

    ~ResourceManager() {
        qDeleteAll(resources);
    }
};
```

How would you fix them? (10 points)

2. Design a system where some QObjects need to outlive their parents. How would you implement this while maintaining clean memory management? Include:
   - A mechanism for ownership transfer
   - Safety considerations
   - Thread-safety aspects

   (10 points)

3. Explain how you would handle object lifetime management in a plugin system where:
   - Plugins are loaded/unloaded dynamically
   - Each plugin creates QObjects
   - Some objects need to persist after plugin unload
   - The host application manages core objects

   (10 points)

## 4 Advanced Concepts (20 points)

1. Explain how Qt's parent-child model interacts with:
   - The event loop
   - Cross-thread object usage
   - QML engine ownership

   (10 points)

2. Design a system that combines Qt's parent-child model with shared ownership (QSharedPointer) for specific objects. Describe:

   - When you would use each approach
   - How to safely convert between them
   - Memory management at system boundaries

   (10 points)

## Bonus Question (10 extra points)

Implement a QObject-derived class that acts as a scoped guard for child objects, automatically reparenting them when the guard goes out of scope. Demonstrate usage in both stack and heap scenarios.

```
// Your implementation here
```