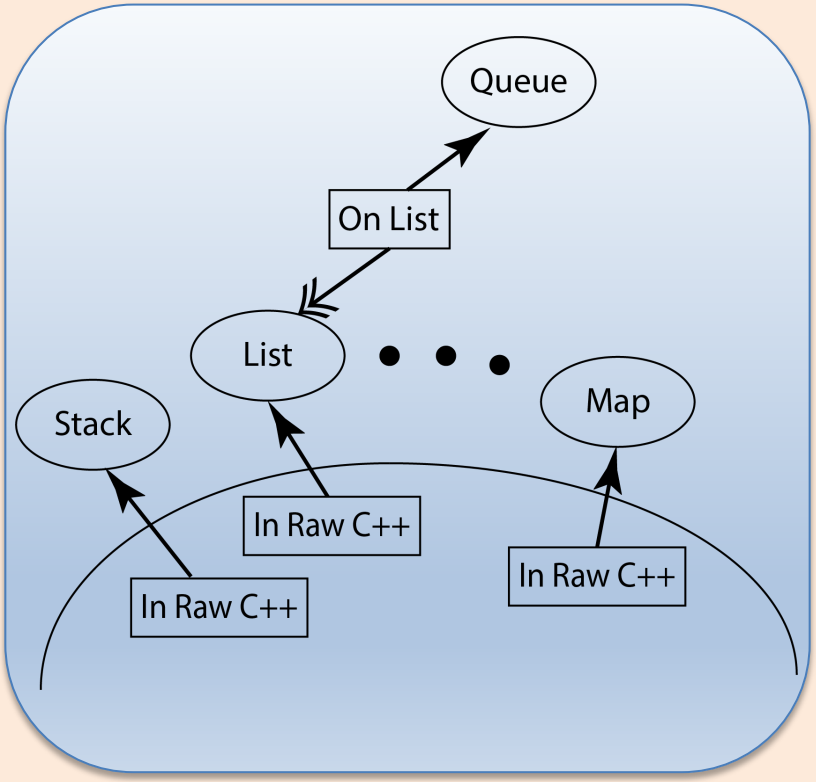
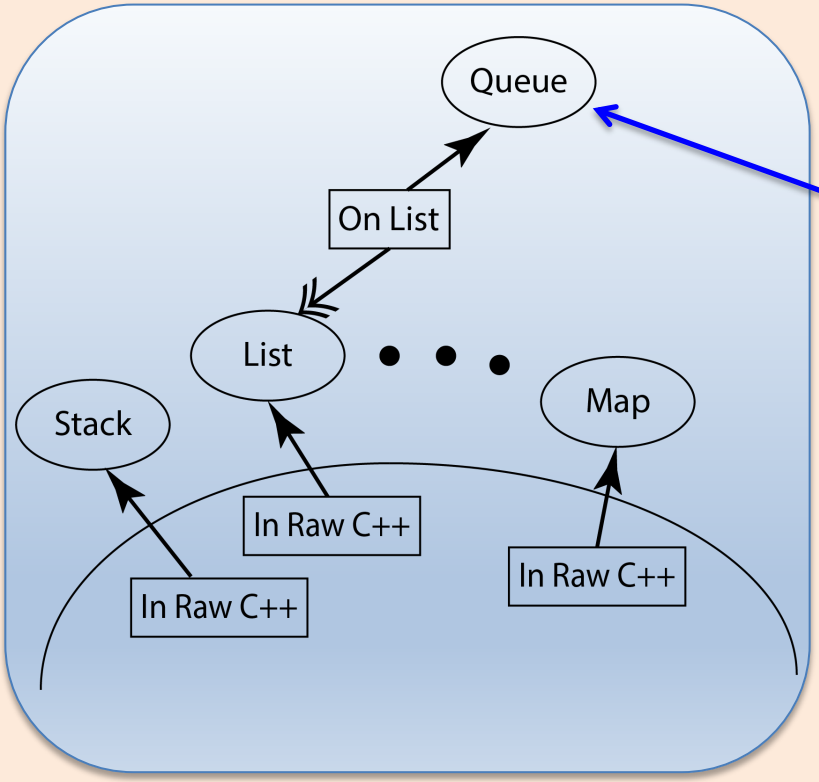
Layering Overview

Diagram of Component Layering



A diagram that allows us to talk about components and their implementations from a design-by-contract perspective

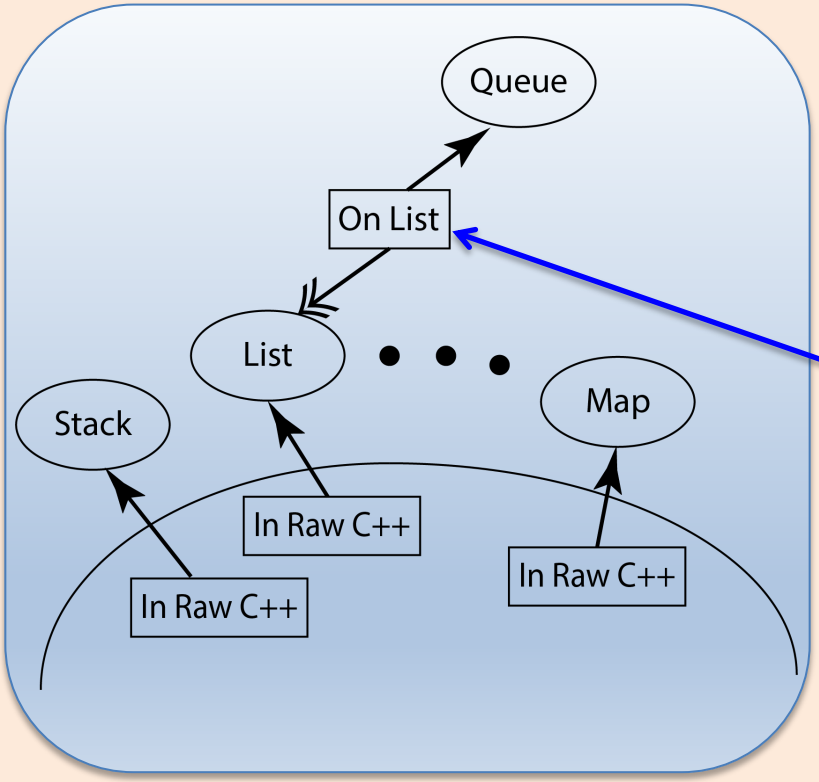
Diagram of Component Layering



How to interpret this diagram:

Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

Diagram of Component Layering

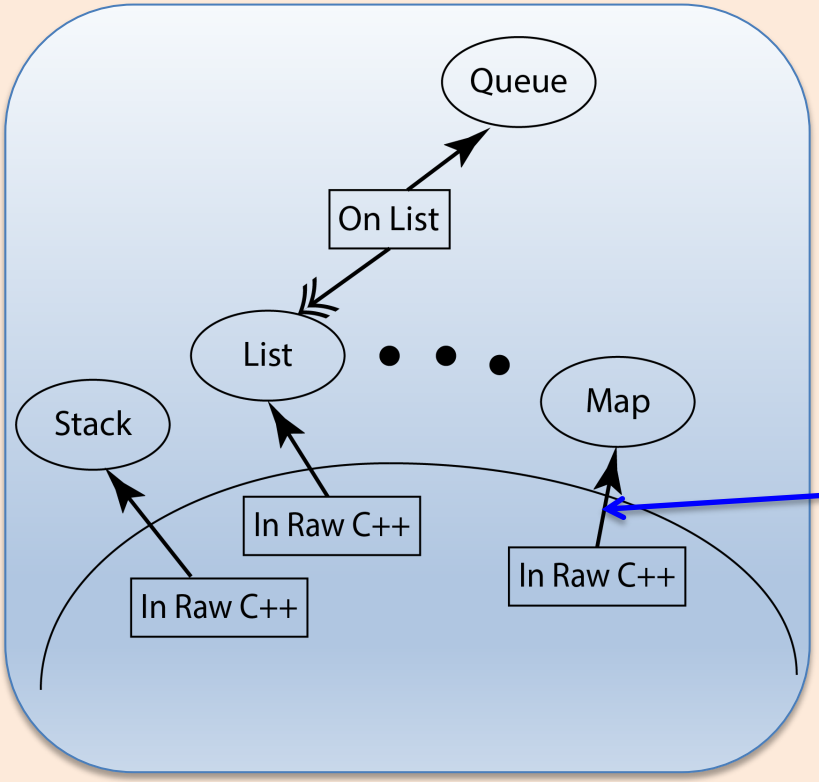


How to interpret this diagram:

Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

Rectangles – represent a component's implementation

Diagram of Component Layering



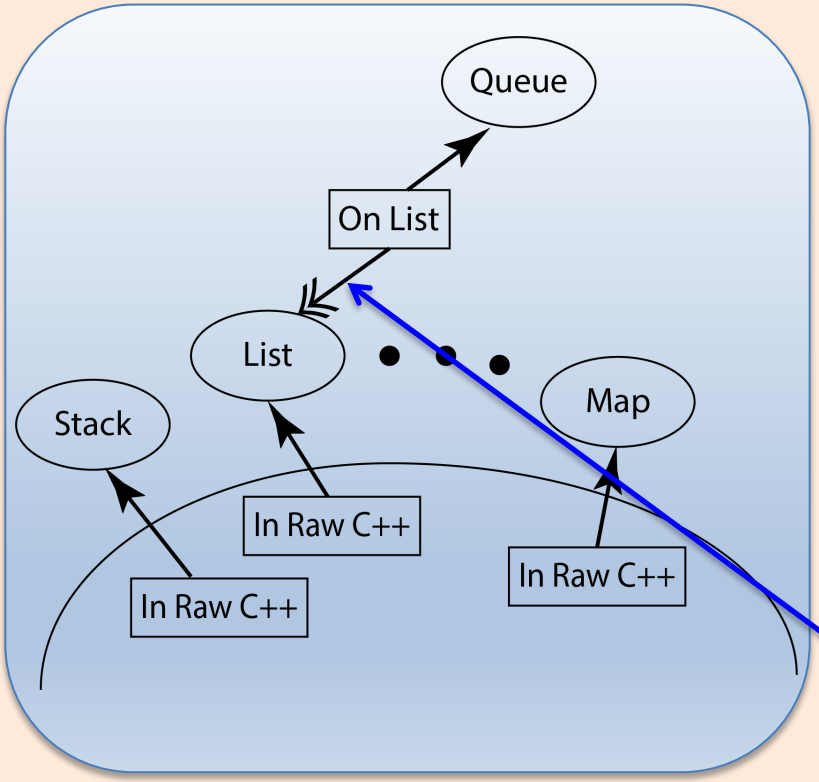
How to interpret this diagram:

Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

Rectangles – represent a component's implementation

Single arrowhead arrows – indicate that the component at the arrowhead *is implemented by* the code found in the arrow's tail

Diagram of Component Layering



How to interpret this diagram:

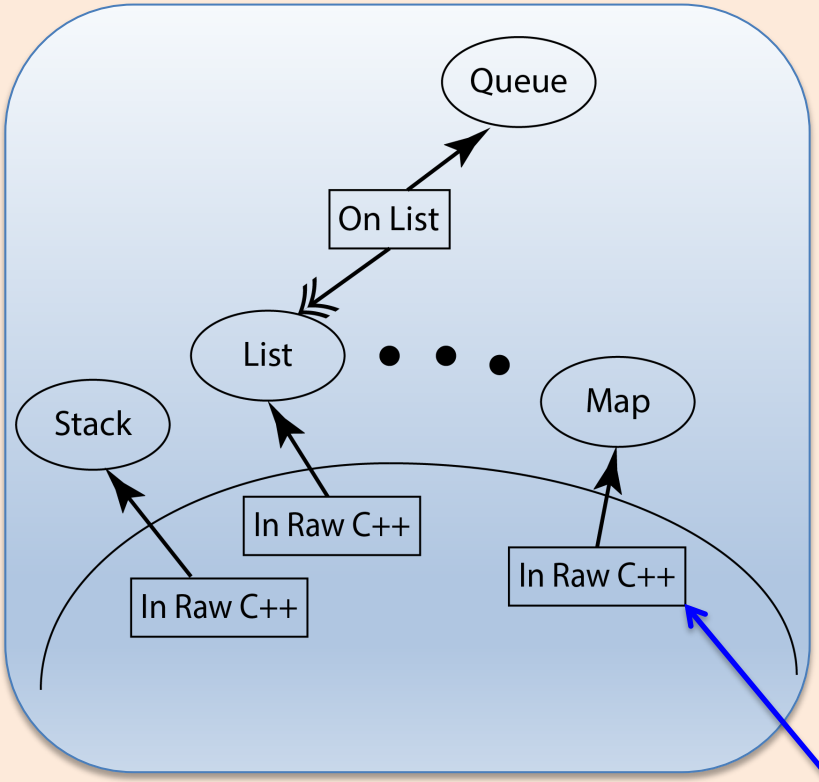
Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

Rectangles – represent a component's implementation

Single arrowhead arrows – indicate that the component at the arrowhead *is implemented by* the code found in the arrow's tail

Double arrowhead arrows – indicate that the code at the tail of the arrow uses the component at the arrowhead

Diagram of Component Layering



How to interpret this diagram:

Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

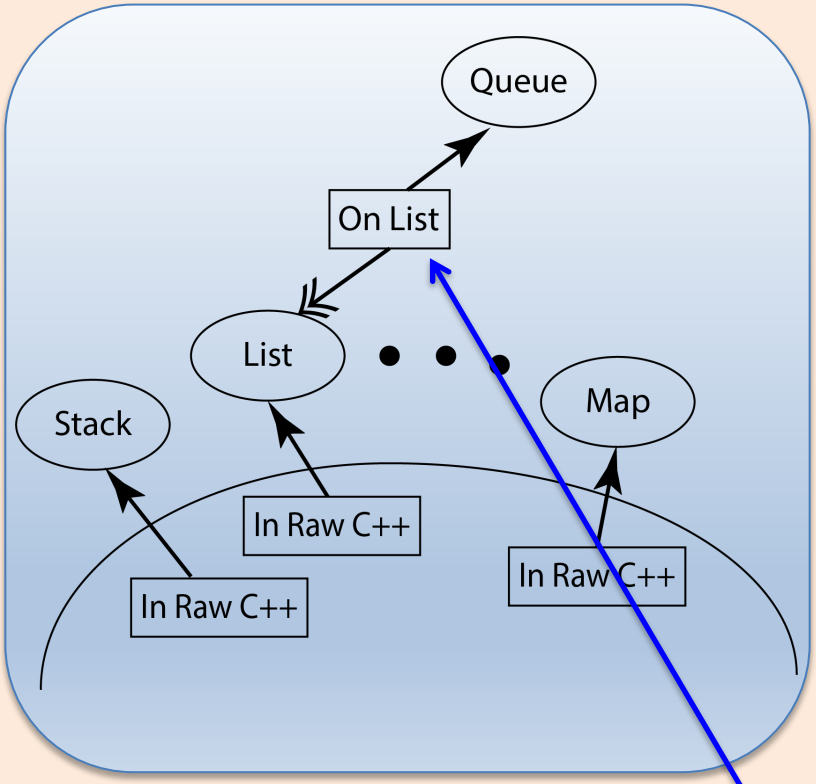
Rectangles – represent a component's implementation

Single arrowhead arrows – indicate that the component at the arrowhead *is implemented by* the code found in the arrow's tail

Double arrowhead arrows – indicate that the code at the tail of the arrow uses the component at the arrowhead

Below the arc implementations – are in the raw programming language, e.g., C++ using nodes & pointers or built in arrays

Diagram of Component Layering



How to interpret this diagram:

Ovals – represent a component's public interface specified using design-by-contract, i.e., a model, and operation contracts

Rectangles – represent a component's implementation

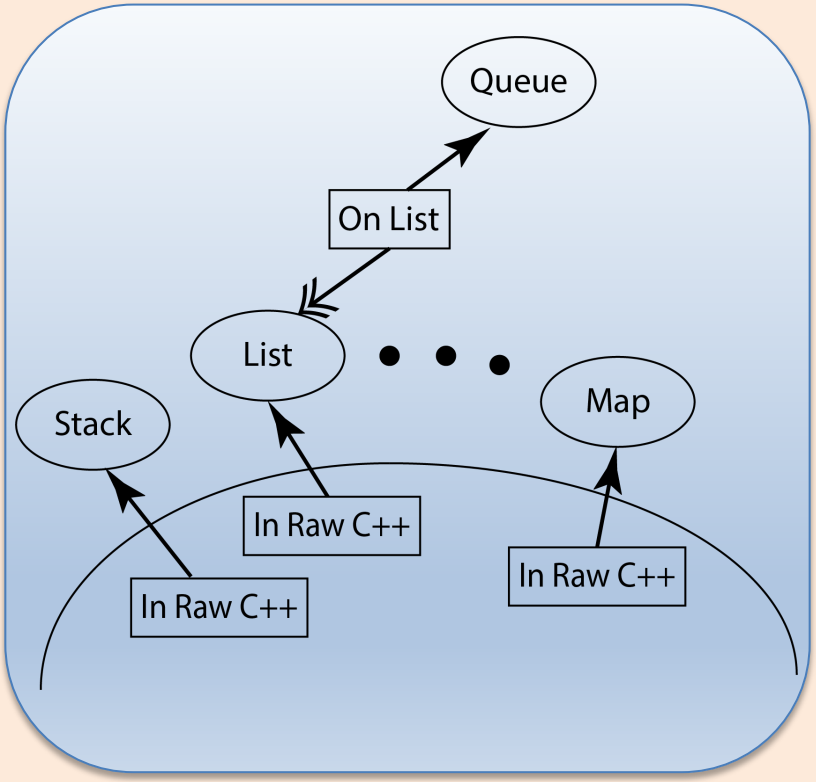
Single arrowhead arrows – indicate that the component at the arrowhead *is implemented by* the code found in the arrow's tail

Double arrowhead arrows – indicate that the code at the tail of the arrow uses the component at the arrowhead

Below the arc implementations – are in the raw programming language, e.g., C++ using nodes & pointers or built in arrays

Above the arc implementations – utilize other components in their implementation and depend on design-by-contract rules, e.g., Queue's implementation depends on and uses the List component

Why We Layer



Below the Arc – Design by contract is informally used because many of the programming language's data structures, e.g., built-in arrays or raw pointer types are not formally specified

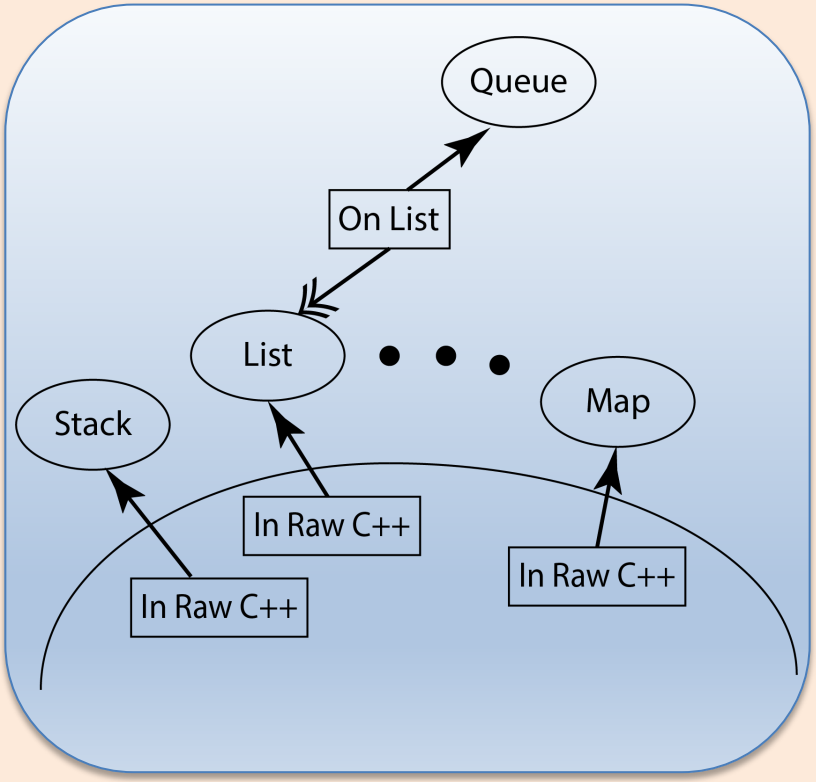
Above the Arc – We can utilize contracts for the interactions among our components and with these contracts we can reason symbolically (using math types) about the values of our variables

Metaphors:

Below the Arc – unrestrained use of any programming construct, where anything goes, like the wild west

Above the Arc – we know where we stand, we're on firmer ground, because we utilize well established software engineering principles to design our component specifications and their implementations

Goals
When Designing a New Component



1. Always design its public interface by using design-by-contract principles
   - provide a math model for reasoning abstractly about variable's values
   - provide operation contracts: parameter modes, requires, ensures

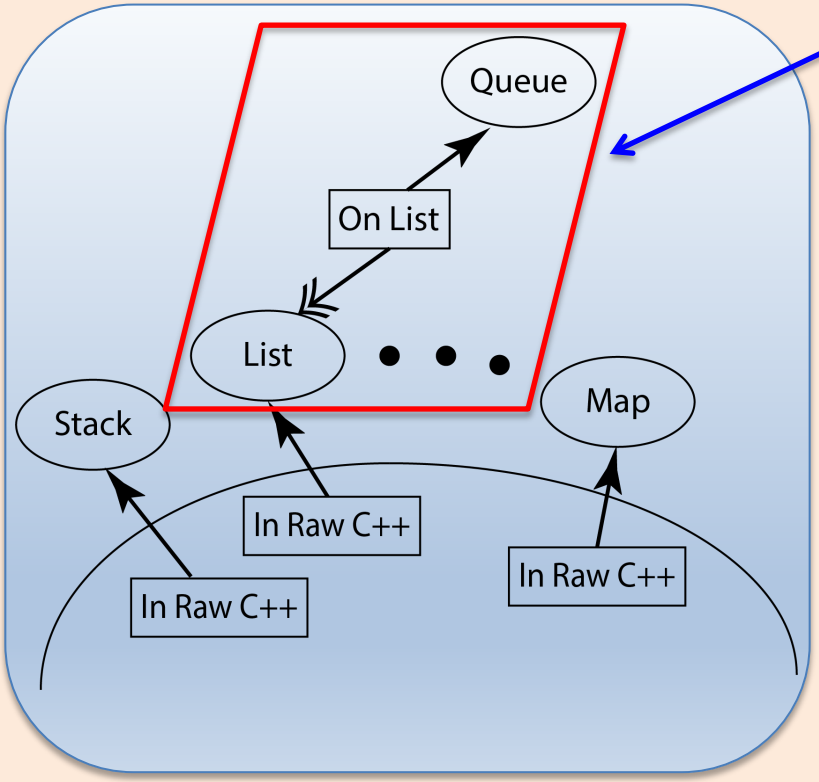
2. When possible, always layer a new component on other existing components, again use design-by-contract principles

3. Implement a new component *in the raw* (i.e., below the arc) only when it's not possible to layer, or when an already existing layered implementation does not provide sufficient performance characteristics

4. Try to never implement *in the raw*

- These goals will lead us to higher quality software

Let's Look at How to Layer



Queue layered on List