These days, we might reasonably elect to implement this functionality using the C++11 standard algorithm `std::any_of`[5]:

```
template <typename InputIterator, typename UnaryPredicate>
bool any_of(InputIterator first, InputIterator last, UnaryPredicate pred);
    // Return true if any of the elements in the range satisfies pred.
```

Prior to C++11, however, using a function template, such as `any_of`, would have required a separate function or function object, defined *outside* of the scope of the function:

```
// C++03 (obsolete)
namespace {

struct IsNodeDirtyOrNew
{
    bool operator()(const SceneNode& node) const
    {
        return node.isDirty() || node.isNew();
    }
};

}  // close unnamed namespace

bool mustRecalculateGeometry(const std::vector<SceneNode>& nodes)
{
    return any_of(nodes.begin(), nodes.end(), IsNodeDirtyOrNew());
}
```

Because unnamed types can serve as arguments to this function template, we can ~~also~~ employ a lambda expression instead of a function object that would be required in C++03:

```
#include <algorithm> // std::any_of
bool mustRecalculateGeometry(const std::vector<SceneNode>& nodes)
{
    return std::any_of(nodes.begin(),             // start of range
                       nodes.end(),               // end of range
                       [](const SceneNode& node)  // lambda expression
                       {
                           return node.isDirty() || node.isNew();
                       }
                      );
}
```

By creating a **closure** of unnamed type via a lambda expression, unnecessary boilerplate, excessive scope, and even local symbol visibility are avoided.

[5] **cpprefa**