```
        if (n) { t->visitInOrder(n);      }  // optionally defined in derived
        if (n) { t->traverse(n->d_right); }  //      "        "     "    "
        if (n) { t->visitPostOrder(n);    }  //      "        "     "    "
    }
};
```

The factored traversal mechanism is implemented in the `Traverser` base-class template. A proper subset of the four customization points, that is, the four member functions invoked from the *public* `traverse` function of the `Traverser` base class, is implemented as appropriate in the derived class, identified by `T`. Each of these customization functions is invoked in order. Notice that the `traverse` function is safe to call on a **`nullptr`** as each individual customization-function invocation will be independently bypassed if its supplied `Node` pointer is null. If a customization function is defined in the derived class, that version of it is invoked; otherwise, the corresponding empty **`inline`** base-class version of that function is invoked instead. This approach allows for any of the three traversal orders to be implemented simply by supplying an appropriately configured derived type where clients are obliged to implement only the portions they need. Even the traversal itself can be modified, as we will soon see, where we create the very data structure we're traversing.

Let's now look at how derived-class authors might use this pattern. First, we'll write a traversal class that fully populates a tree to a specified depth:

```
struct FillToDepth : Traverser<FillToDepth>
{
    using Base = Traverser<FillToDepth>;  // similar to a local typedef

    int d_depth;            //  final "height" of the tree
    int d_currentDepth;     //  current distance from the root

    FillToDepth(int depth) : d_depth(depth), d_currentDepth(0) { }

    void traverse(Node*& n)
    {
        if (d_currentDepth++ < d_depth && !n)  // descend; if not balanced...
        {
            n = new Node;     // Add node since it's not already there.
        }

        Base::traverse(n);    // Recurse by invoking the base version.

        --d_currentDepth;     // Ascend.
    }
};
```

The derived class's version of the `traverse` member function acts as if it overrides the `traverse` function in the base-class template and then, as part of its re-implementation, defers to the base-class version to perform the actual traversal.