achieving runtime polymorphism is through the disciplined use of **union**s of standard-layout class types in a manner that we'll refer to here as **vertical encoding** — i.e., an initial sequence of data whose respective types are (1) common to all encodings and (2) may affect the interpretation of subsequent data in those encodings.

To readily compare the benefits of vertical encoding with those of an **object-oriented** design, we begin with the classic introductory object-oriented "shapes" example, which provides a pure abstract (a.k.a. **protocol**) base class, VShape:

```
#include <iostream>  // std::ostream

struct VShape  // pure abstract base class (a.k.a. protocol)
{
     virtual ~VShape() { }
     virtual std::ostream& draw(std::ostream& stream) const = 0;
         // Format this shape to the specified output stream.

     // ... (any additional methods common across all shapes)
};
```

Even though VShape is an **abstract class**, the destructor, ~VShape, must be defined, not just declared, because derived classes' destructors will invoke the base class's destructor. Note that the destructor for an abstract base class is never called via virtual-function dispatch; it is called only from derived-class destructors. In our example, the empty destructor is **inline** so as to minimize the cost of destroying derived-class objects.

Next, we derive, from our abstract VShape base class, the various concrete shapes in our application:

```
struct VCircle    : VShape    // size 16, alignment 8
{
    double d_radius;
    VCircle(double radius);
    virtual std::ostream& draw(std::ostream& stream) const;
    // ...
};

struct VRectangle : VShape    // size 16, alignment 8
{
    short d_len, d_width;
    VRectangle(short length, short width);
    virtual std::ostream& draw(std::ostream& stream) const;
    // ...
};

struct VTriangle  : VShape    // size 24, alignment 8
{
    int d_side1, d_side2, d_side3;
```