

The Quilts Exercise: Defining A TriangleWidget Class

In the `quilts` sample program, the random quilts are drawn by randomly deciding for each block whether to maintain its rectangular shape or to divide it into two or four triangles. Since the Core Tools do not provide a widget for drawing triangles, the triangles are drawn using `PolygonWidget` objects. This means that the same sequence of code for drawing triangles is repeated 8 times in the `quilts` program. The purpose of this exercise is for you to define a simple `TriangleWidget` class and use the class to simplify the code for random quilts.

Here is an excerpt from one of the functions in the `quilts` program:

```
void FourTriangleBlock(int x1, int y1, int x2, int y2) {
    PolygonWidget PW;
    PW.ChangeMemory(4).SetFill();

    int xc = (x1 + x2)/2;
    int yc = (y1 + y2)/2;

    RandomColor();
    PW.Append(xc, yc);
    PW.Append(x1, y1);
    PW.Append(x2, y1);
    PW.ClosePolygon().Draw();

    PW.ClearPolygon();
}
```

etc.

Most of the polygon code is repeated many times in the program. The common steps are:

1. Introduce a `PolygonWidget` named `PW`.
2. Inform `PW` to allocate enough memory for 4 points.
3. Inform `PW` to use *fill mode* when it draws itself.
4. Append the 3 points of the triangle to `PW`.
5. Close `PW` by adding the initial point again.
6. Tell `PW` to draw itself in the current graphics window.

We want you to create a `TriangleWidget` with a `Draw` operation that will encapsulate this code. The goal is that the above lines of code are reduced to:

```
void FourTriangleBlock(int x1, int y1, int x2, int y2) {
    TriangleWidget T;

    int xc = (x1 + x2)/2;
    int yc = (y1 + y2)/2;

    RandomColor();
    T.Set(xc, yc, x1, y1, x2, y1).Draw();
}
```

etc.

The private member data of the `TriangleWidget` class will consist of 3 fields of type `PointData` that we will label `p`, `q`, `r` (using lower case for the variable names). Thus the definition of the `TriangleWidget` class will begin:

```
class TriangleWidget {  
  
    PointData p;  
    PointData q;  
    PointData r;  
  
public:
```

We now will discuss the public functions of `TriangleWidget`. These functions will fall into four categories:

1. `set` functions to use external information to change the internal fields `p`, `q`, `r`.
2. `get` functions to return the values in the internal fields `p`, `q`, `r`.
3. Constructors
4. The `draw` function that encapsulates the triangle drawing tasks.

In many ways, the first three categories are the *overhead* of class definition. In order to use a class smoothly, it must support convenient mechanisms to create or construct an object and to set and get its internal parameters. Beginners often find these definitions tedious and boring but the rhythm is simple and once learned is fairly automatic. When you deal with more sophisticated classes in which improper changes to internal variables can cause catastrophic failures, the value of learning these encapsulation techniques becomes evident.

For convenience, the `set`, `get`, and `constructor` functions should be able to deal with both `PointData` information and with `short` integer coordinates. Therefore, we request that you define 7 member functions with headers as follows:

```
TriangleWidget& Set(const PointData& P, const PointData& Q, const PointData& R)  
TriangleWidget& Set(short x1, short y1, short x2, short y2, short x3, short y3)  
void Get(PointData& P, PointData& Q, PointData& R) const  
void Get(short& x1, short& y1, short& x2, short& y2, short& x3, short& y3) const  
TriangleWidget()  
TriangleWidget(const PointData& P, const PointData& Q, const PointData& R)  
TriangleWidget(short x1, short y1, short x2, short y2, short x3, short y3)
```

To illustrate how these functions are defined, let us actually give the definition of the second of the `set` functions:

```
TriangleWidget& Set(short x1, short y1, short x2, short y2, short x3, short y3)
{
    p.Set(x1, y1);
    q.Set(x2, y2);
    r.Set(x3, y3);

    return *this;
};
```

As you see, each pair of `x`, `y` parameters is used to set one of the internal `PointData` fields. This task is accomplished by calling a `set` function in the `PointData` class. This is an example of how a more complex class does its work by using the member functions of its internal data.

The function return is designed to make the `TriangleWidget` object that calls `set` immediately available for further work. This requires coordination of the *return type* and the final `return` statement at the end of the function.

1. The return type must be a reference-to-an-object-of-the-class. In this example

```
TriangleWidget& Set(...)
```

2. The final return statement must be:

```
return *this;
```

What does the idiom `return *this;` really mean? The explanation takes several steps.

When any member function `F` of a class `C` is called for an object `x` of the class, the compiler sets up a pointer to that object `x` and stores this pointer in a *reserved variable name* called `this`. Thus:

```
X.F(...); // object X of class C calls member F with some parameters
```

leads automatically to the hidden definition of the pointer `this`:

```
C* this = &X; // store pointer to X in the reserved variable called this
```

What this accomplishes is that `*this` is another name (*reference*) for the object `x` that called the function `F`. Thus the code `return *this;` means:

Return a *reference* to the object `x` that made the member function call to `F`

This return mechanism permits you to call additional member functions immediately. Thus, in the `TriangleWidget` case, you can write one line

```
T.Set(xc, yc, x1, y1, x2, y1).Draw();
```

that both sets the triangle data in `t` and tells the widget to draw itself.

Generally, our design philosophy for `set` functions is that they should return a reference to the object (via `return *this;`) unless there is an important reason to return some other value. The most important exception is when the `set` function may fail in which case the return type is usually `bool` with `true` indicating success and `false` indicating failure.

Our design philosophy for `get` functions is that the caller is likely to use the information that is extracted in some other function so that it is unlikely that the caller will immediately call a member function on the object. Thus, `get` functions return nothing (`void`).

A few words need to be said about the constructors. If no constructors are defined in a class, then the compiler will automatically generate a *default constructor* of the form:

```
TriangleWidget() {};
```

This constructor tells an object simply to call the default constructors of its member data fields and do nothing else.

Here's the tricky part. *If you want to define non-trivial constructors, then the compiler will no longer define the above default constructor automatically and so if you also want a default constructor you must insert the above definition explicitly.* In our case, we want to be able to introduce a `TriangleWidget` directly via:

```
TriangleWidget T;
```

Since `t` has no parameters, this requires the default constructor whose definition must be given explicitly since we also wish to define other constructors.

The definition of the explicit constructors is fairly trivial since they can use the `set` functions. For example:

```
TriangleWidget(const PointData& P, const PointData& Q, const PointData& R)
    { Set(P, Q, R); };
```

The mechanism of using the `set` functions to define constructors is fairly common. Be aware, however, that *if dynamic allocation is involved in construction then the constructor code and the set code must be distinct.*

The final public member function to define in the `TriangleWidget` class is the `Draw` function which can have the header

```
void Draw() const
```

The work of `Draw` is outlined in the six steps listed at the beginning of this exercise and we refer you to that discussion for the details.

After `TriangleWidget` is defined, replace the code in `TwoTriangleBlockA`, `TwoTriangleBlockB`, and `FourTriangleBlock` and then test that the random quilts draw successfully.

Remarks:

1. To learn about the classes used in this exercise (`PointData` and `PolygonWidget`), see the files `GeoTypes.h`, `GeoTypes.cpp`, `GraphicsWidget.h`, and `GraphicsWidget.cpp`.

2. You do not actually use all of the member functions defined in the `TriangleWidget` class in the random quilts program. Nevertheless, these member functions are typical of so many classes that we felt it was important that you learn how to define them.

3. There are other options that could be included in the `TriangleWidget` class that we decided for simplicity to omit in this exercise.

A. In **Core Tools**, widgets that can be filled with color provide three options: frame the object (draw its boundary only), fill the object, or framefill the object (fill the object and then frame it). We decided to omit this option since in the quilts program we only need to fill the triangles.

B. In **Core Tools**, widgets are designed to inherit either from `GraphicsWidget` (things that directly affect the graphics) or `IndirectWidget` (things that use other things to draw). In a more complete definition of the `TriangleWidget` class, the class would inherit from `IndirectWidget` and follow the conventions defined in `GraphicsWidget.h`. We decided to omit this option since the inheritance issues are not essential for the exercise.