Oy! Just b'cuz me digits be oight
It don't mean me name be boit

meg the manitoed octopus

Shapes

In this quest, you get to play with inheritance and polymorphism in C++.

What is polymorphism? Take a break to look up your reference material now. TL;DR? Here is the abstract scoop (followed by the more concrete example which you will implement in this quest):

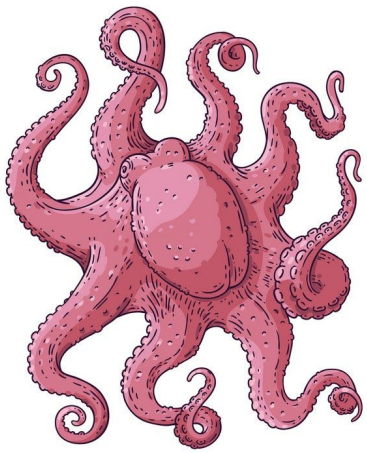
Suppose you had many different classes, X_1, X_2, ..., that all inherited from the same base class `A` and that there is some method of `A`, say `foo()`, that is overridden (possibly differently) by each of its descendants.

Let's say you have an array of pointers to objects of type `A`.[1] Each object pointed to by this array's elements may be a different X_i and each X_i may implement its own overriding `foo()` according to its own specific requirements.

Polymorphism lets you write code in which you can invoke `foo()` via a pointer to an object of type `A`, only to have it execute the `foo()` code that is attached to its derived class X_j rather than the *fallback-code* that may be supplied with class `A`.

I bet you're still confused. Cuz I almost done confusin meself writin that. This will help: Here is the concrete example (which you will also implement in this quest).

Suppose I have a generic `class` called `Shape` (a geometric shape). There may be many different subclasses of `Shape`: `Circle`, `Square`, `Triangle`, etc. Each of these subclasses must have its own `draw()` method (enforced by the compiler). 'Cuz I can't draw a triangle from vertices in the same way I draw a square, yes?



Polymorphism is a way that lets us invoke `draw()` blindly on a set of `Shape` objects when we don't know the precise type of each object. The idea is that the system will automatically invoke the correct `draw()` on each object depending on the specific subclass it belongs to (Get yer eyes off the dancing octopus and look at Figure 1 already!)

[1] Why pointers? Can we not have an array of the objects themselves?

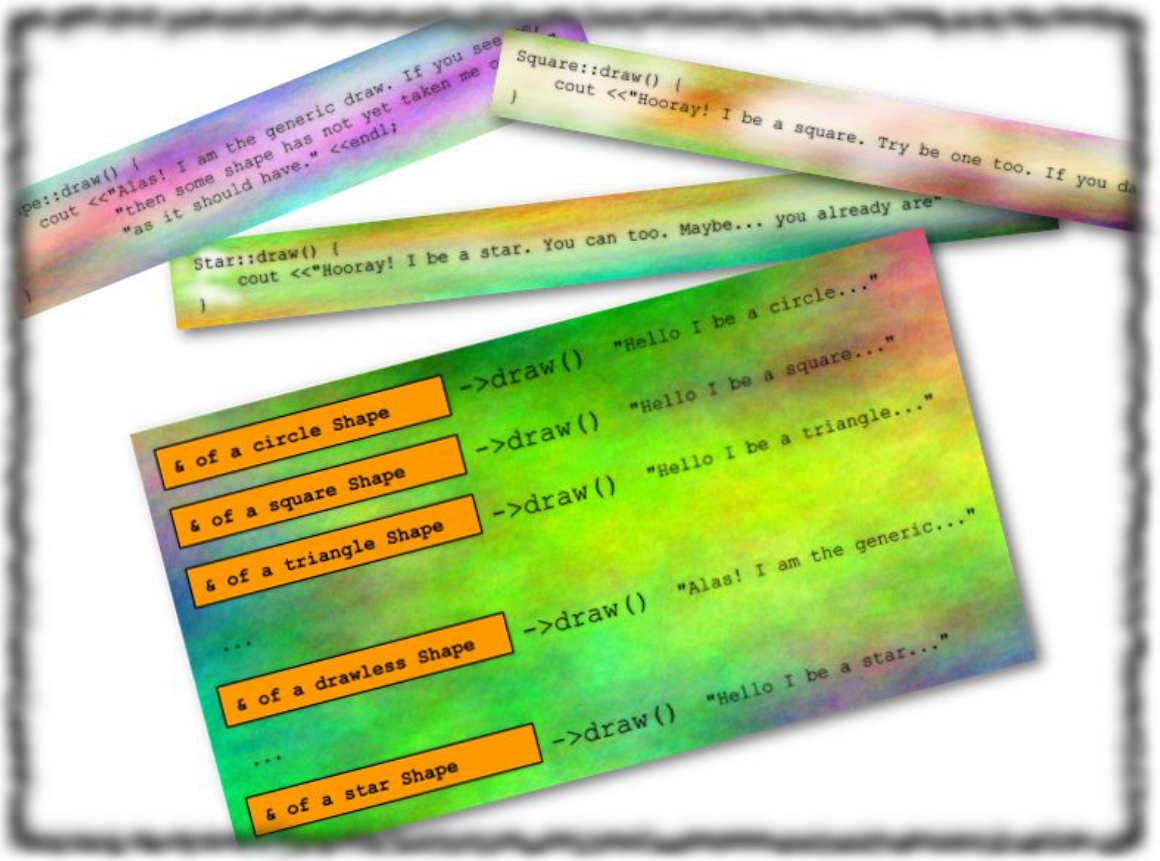


Figure 1: `draw()` is a virtual function that resolves to the appropriate derived class member function automatically. Imagine the orange boxes to be cells in an array of pointers to `Shape` objects.

In this quest, you will implement many little classes. Some of these will interact with each other in fun ways (See Figure 2).

- A `Screen` class - Your canvas, on which you can paint using ASCII characters (That is, each pixel on this canvas is an ASCII character).
- A `Shape` class - The base class for all the shapes you're gonna create. In fact, we'll make it an *abstract* class by assigning the value 0 (the null pointer) to the method we want all subclasses to implement: `draw()`. This *empty implementation assignment* tells the compiler that it is what we call a *pure virtual function*. Any class that subclasses (derives from) `Shape` MUST implement `draw()` or the compiler will consider that derived class also an abstract class. An abstract class cannot be instantiated because it has not yet been completely defined. The missing method implementations complete it. Then it becomes a *concrete* class. You can instantiate objects of a concrete class.
- You will also implement the following concrete subclasses of `Shape`: `Point`, `Line`, `Quadrilateral`, `Upright_Rectangle` and `Stick_Man`

- Each subclass should know how to draw itself (and not other Shapes) on a given screen using a given character. Thus each should have its own implementation of the `draw()` method.

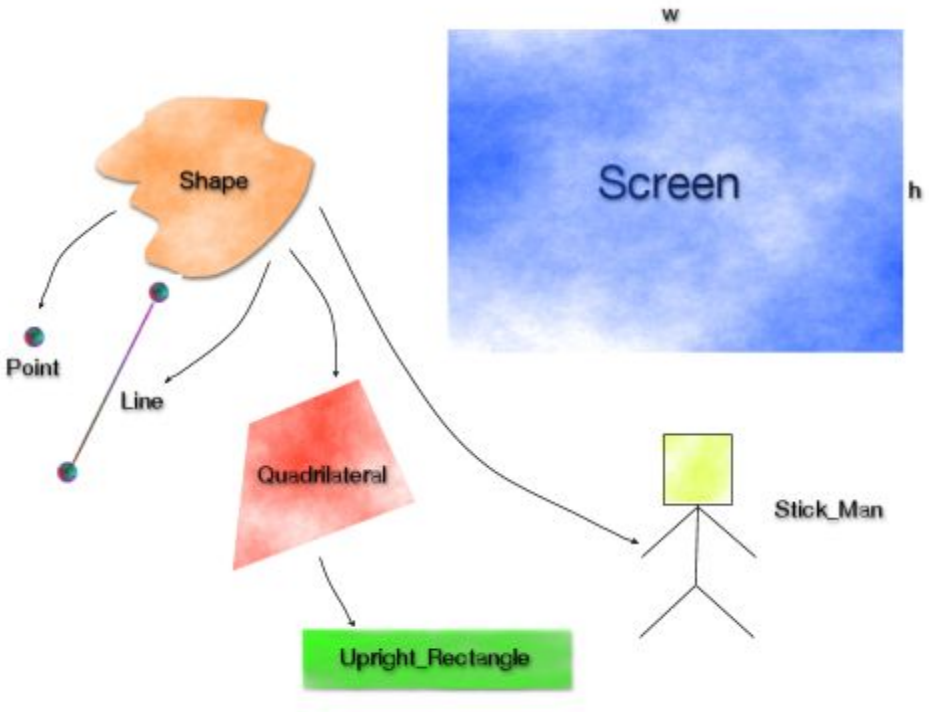


Figure 2: Classes you must implement. Arrows represent inheritance. `Shape` is an abstract class. Its subclasses must implement `draw()`

There's a lot you can do in this quest. Most of it is easy, but some miniquests are non-trivial. So I would suggest this:

Work on this quest until you get the password to move on. Then keep coming back to this quest to revisit skipped miniquests until the second great freeze.

Your first miniquest - Screen

Implement the Screen constructor:

```
Screen::Screen(size_t w, size_t h)
```

It should correctly set the size of its `_pix` vector to have `h` rows and `w` columns. This means `_pix[]` should end up with `h` `vector<char>`s, each of which is `w` chars big.

Points to keep in mind about the `Screen` class (not necessarily to do with the constructor):

- Vectors at `_pix[0...h-1]` represent screen pixel rows `0` to `h-1` from bottom to top.

- The origin, `(0,0)`, visually located at the bottom left of screens, is now the top left of the `_pix` vector if you imagine the vector growing downwards with row 0 at the top. So you have to get used to flipping the image vertically as you process it (like in the eye?). Or do you need to?

Your second miniquest - Fill

Implement:

```
void Screen::fill(char c)
```

Just fill up the screen (all elements of all rows of `_pix`) with the supplied character, `c`.

Your third miniquest - Clear

Implement:

```
void Screen::clear()
```

Just fill up the screen (all elements of all rows of `_pix`) with the background character, Screen::BG. You can simply invoke `fill()` from here. Hey look! I alreddy gave you this function line! Well, I guess I can't take it back now.

Your fourth miniquest - To string

Well, there's always a tricky one in every quest. I guess. But this doesn't have to be tricky if you remember that the bottom row of the vector is the top row of your screen and vice-versa.

This is also where you might wonder if you'd have been better off mapping bottom to bottom and top to top, and then removing the corresponding adjustments from the draw methods.

Implement:

```
string Screen::to_string() const
```

It should return a string made of `h` newline delimited strings, each of which represents the corresponding row on a screen (line 0 is the top of the screen). That is, I should be able to say something like:

```
cout << my_screen.to_string() <<endl;
```

and have it print out a rectangular frame of characters in the same order I would see if the frame was stored in a flat file and dumped to the console. For example, if `my_screen` had a width of 11 and a height of 6, and it contained the following characters:

```
_pix[5] = '  -------  '
_pix[4] = ' | /\_/\ | '
_pix[3] = ' |( o o )| '
_pix[2] = ' | > ^ < | '
_pix[1] = '  -------  '
_pix[0] = 'Schrodinger'
```

it's output should be what miniquest 1 of Quest 2 requires.

Oh wait! Wrong Quest. But still... you get the idea.

If not, bring it up in our subreddit.

Your fifth miniquest - Output

Implement:

```
friend std::osream &operator<<(std::osream &os, const Screen &scr)
```

Hey! It looks like I gave you that one too. I'm gonna have to think up some harder miniquests.

Your sixth miniquest - Point

We start our Shapes with `Point`, seeing as I've already given you the `Shape` class in the starter code.

Complete the implementation of the concrete class, `Point`. Remember that the word *concrete* means that the class is not abstract. It implies that the class is instantiable - that there are no inherited pure virtual methods left without implementations.

Essentially, what this miniquest boils down to is just that you implement:

```
bool Point::draw(Screen &screen, char c)
```

If the point falls within the given screen's boundaries, place the character `c` at the corresponding location within the screen's `_pix` array and return true. If not, return false.

How can `Point` access the private members of `Screen` (`_h`, `_w`, and `_pix`)? Although you can do this through friendship (experiment, discover and share your findings), it's just as easy to keep the classes opaque to each other and use the getters provided by `Screen`. Discuss the pros and cons in the subreddit if you have time.

Your seventh miniquest - Line by ...

Implement:

```
bool Line::draw_by_x(Screen &screen, char c)

bool Line::draw_by_y(Screen &screen, char c)
```

A `Line` is represented by 4 unsigned integers (not two `Points`): x1, y1, x2 and y2. It is defined as the set of points connecting the points (x1,y1) and (x2,y2).

When it comes to slanting lines, you'll find that simple strategies don't always work. The problem is that either x or y will need to be incremented by a fractional quantity for every unit-increment of the other. This will make more sense after you read the following general strategy to draw a line:

- Always draw the line from left to right. If it is vertical, draw it from bottom to top.
- If the line is wider than it is tall, then starting at the leftmost pixel, increment x by one each time and y by the fraction determined by the slope of the line (See Figure 4).
- If the line is taller than it is wide, then starting at the bottommost pixel, increment y by one each time, and x by the reciprocal of the slope of the line.
- Finally, return true if the line was *entirely* contained within the screen and false if even one of the points of the line falls outside screen bounds.

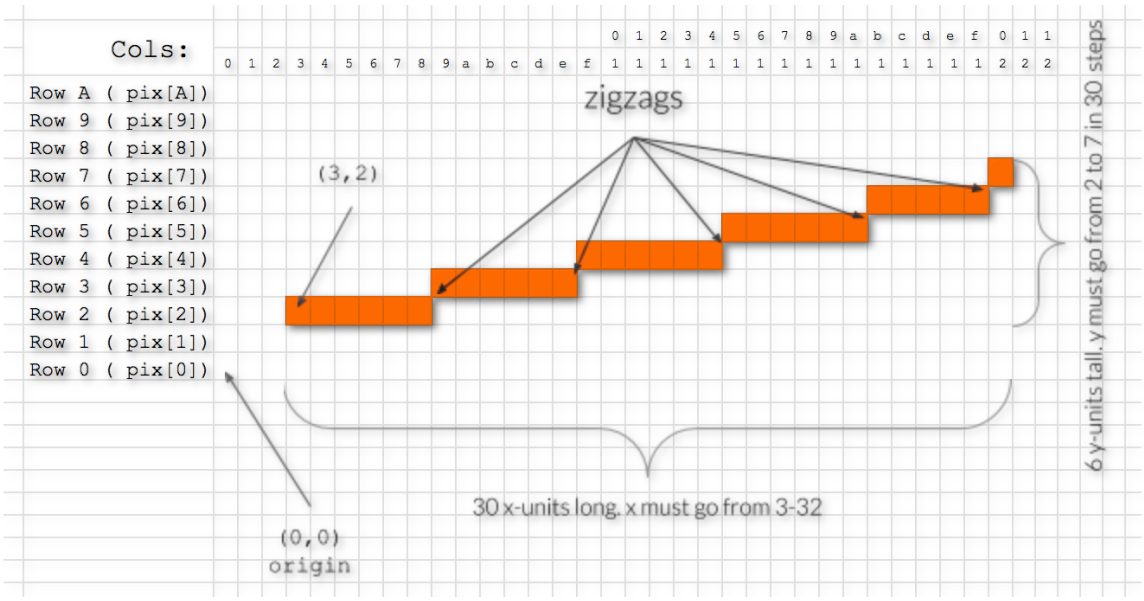


Figure 4: How to draw a slanting line

In Figure 4, you can see how to select the pixels to turn on for a line that goes from (3,2) to 32,7). Since the line is wider than it is tall, our strategy would draw the line along the x-direction. What this means is that you will calculate N coordinates where N is the horizontal length of the line (30). Your x would be steadily incremented by 1 during each of N iterations. The corresponding y-coordinate for each x will be calculated by adding a fraction, which I call dy in the code, to the previous value of y.

The basic idea is to try and draw the slanting lines such that the number of zigzags is maximized and they don't occur at aesthetically jarring locations.[2] In this miniquest, our chosen strategy inserts `h-1` zigzags (where `h` is the vertical length of the line and distributes the zigzags to be equally spaced between the ends of the line.

In the example, y needs to go from 2 to 7 in 30 iterations (x from 3 to 32). This means your dy = 5/29.0, which is the quantity by which you would increment y for each increment in x. If done properly, you will find that y rolls over from one coordinate to the next once every 6 iterations (when x would have moved by 6 steps) on average. (Why on average?)

Try the technique on a piece of paper and convince yourself it works. This method is called `draw_by_x()` in the spec.

Because this method involves floating point calculations, it's possible to calculate quantities slightly differently and get a correct answer that is a little different from my reference answer. Rounding and adjusting will catch most of these drifts. But in rare cases, even small mismatches

[2] However, this is not entirely true in this quest. Our strategy allows some aesthetically sub-optimal zig-zag placements to squeak through. Specifically, which are these and what might be a good way to handle them also?

accumulate over time and manifest at the ends of long lines (mainly at the end points). To avoid this confusion, try to model your calculations to mirror mine as much as possible.

Here is a photo of my `draw_by_x()` method. I suggest you implement your own first, and then AFTER you have your version working, use the photo below as a reference to troubleshoot when your lines are off-by-one from mine. Once you have your `draw_by_x()` working you must create a corresponding `draw_by_y()` method.

```
// Static helper
// Draw pixels on the screen along the X direction, using the supplied
// char (ch) as the pixel. The number of segments will be abs(y2-y1).
// Each segment will be abs(x2-x1)/abs(y2-y1) pixels long.
//
bool Line::draw_by_x(Screen& scr, char ch, size_t x1, size_t y1, size_t x2, size_t y2) {
    if (x1 > x2)
        return draw_by_x(scr, ch, x2, y2, x1, y1);     // reorder

    double dy = ((double) y2-y1)/((double) x2-x1);
    bool contained = true;
    double x = x1, y = y1;
    while (x <= x2) {
        contained &= Point((size_t) x, (size_t) y).draw(scr, ch);
        x += 1; y += dy;
    }

    return contained;
}
```

Figure 5: `draw_by_x()`

Note that my `draw_by_x()` always draws from left to right. If my two points are ordered differently, then rather than futz around with swapping values within the code, I simply make a tail-recursive invocation of the same method with swapped points. You don't have to do it this way, of course. But if you do, note that the recursion overhead here is quite small and capped at one stack frame.[3]

Your eighth miniquest - Line

Implement:

```
bool Line::draw(Screen &screen, char c)
```

[3] Discuss this in our subreddit.

Once you have the two `draw_by_...()` methods done, this one is eazy peazy. All this method has to do is to determine if the line is short and fat or tall and thin, and invoke the corresponding `draw_by_...()` method.

Remember that it must also return a boolean indicating non-overflow. You can blindly relay what your subordinate method returns to your caller.

Your ninth miniquest - Quadrilateral

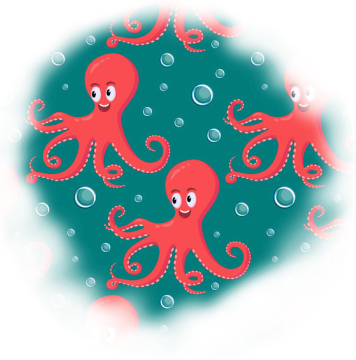
Implement:

```
bool Quadrilateral::draw(Screen &screen, char c)
```

You just have to determine the correct end points of the four lines that make this quadrilateral and draw them.

In the past, I've noticed that some students put in a great deal of effort to *avoid* redrawing corner pixels (where the edges overlap and intersect).

Let me just wish you good luck if you get sucked into that rabbit hole. Once you've established that overwriting an existing pixel with the same value is *safe*, you'll discover that it is a wiser choice to overwrite it than to build elaborate checks and guards. These kinds of educated tradeoffs are what come in handy as you gain more experience in programming.



The Upright Rectangle

A quarryshmaterell uses a lot of letters and it's hard to spell right. It also requires 4 points (and thus 8 numerical coordinates) to specify. Sure looks messy and if you're ever in a situation where the only quads you need are upright rectangles, you can simplify your life a great deal by using just 4 numbers to encode your shape - the bottom left and top right.

Thus was born the `Upright_Rectangle`.

This is not a miniquest. You'll notice that the `Upright_Rectangle`'s implementation is already completely fleshed out in the provided starter code. But you do need to know how to instantiate and use it because it is the head of the next class you need to create.

Nonetheless, FWIW, you get a freebie reward just for reading this section. Hooray!

Your tenth miniquest The Stick Man constructor

Implement:

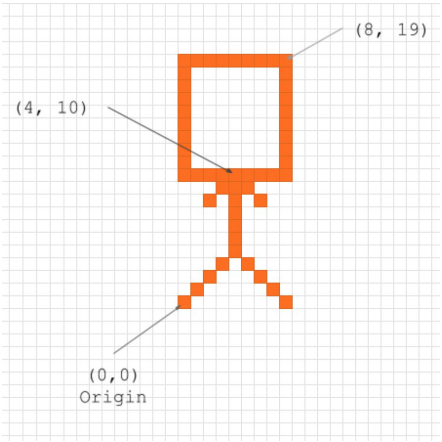
```
Stick_Man::Stick_Man(size_t x, size_t y, size_t w, size_t h)
```

Imagine that your stick man is contained within a rectangular portion of a screen. Then the coordinates of the bottom left of this portion is `(x,y)` and its width and height are given by `w` and `h` respectively.

- First clear out the `_parts` vector and set any required member variables.
- Then create and push pointers to following `Shapes` into this vector:
    - The stick man's head: An upright rectangle with bottom left at `(x+0,y+h/2)` and top right at `(x+w-1, y+h-1)`.
    - His torso: A line from `(x+w/2,y+h/2)` to `(x+w/2,y+h/4)`
    - His left arm: A line from `(x+w/2,y+h/2)` to `(x+w/4,y+h/4)`
    - His right arm: A line from `(x+w/2,y+h/2) to (x+3*w/4,y+h/4))`
    - His left leg: A line from `(x+w/2,y+h/4)` to `(x,y)`
    - His right leg: A line from `(x+w/2,y+h/4)` to `(x+w-1,y)`

That is your stickman. Your default height is 40 and default width is 20. If the constructor is passed values of 0 or 1 for `h`, you should silently set it to this default height (don't worry about throwing exceptions). Likewise for `w`.

Here is a picture of a nice little 9x20 stick man. Of course, not all stick men are nice or little, never mind both!



Note: If the shapes you create are local to the constructor (i.e. on the stack), you'll find that they're gone when the constructor exits. To make sure the shapes persist beyond the lifecycle of the constructor, they must be on the heap. This means you must delete them in your destructor or suffer memory leaks.

Your eleventh miniquest - Draw a Stick Man

Implement:

```
bool Stick_Man::draw(Screen &screen, char c)
```

Just like the other inherited draw methods, this method should also draw its shape (in this case a `Stick_Man`) on the given screen using the character `c`.

Simply invoke the `draw()` method on each of the objects whose pointers you can find in your `_parts` vector. You don't have to worry about the exact type of each object because you know that every one of those objects is a *drawable* object. That is, it supports a `draw()` method whose signature is exactly as given in the base class.

Feel free to gush prosaic in our subreddit at this nice solution to the problem of invoking distinct methods (with the same signature) on many objects sharing the same parent.

Yippee! Problem solved!

Your twelfth and final miniquest in this quest

Yeah, I know an octopus only has 8 feet. But we're not mapping the miniquests to creature feet in our quests. Otherwise you'd never be able to complete my millipede quest.

Implement:

```
Stick_Man::~Stick_Man()
```

You must take care to explicitly delete every shape you created in your constructor before the Stick_Man destructor forgets the (presumably only) pointers to them in your code. When would that happen?

Starter code

Here are the (sometimes incomplete) class definitions you can use to flesh out your classes. I suggest you implement this quest in two files: A `Shapes.h` file which contains all the class definitions (skeleton below), and a `Shapes.cpp` file that contains implementations for all of the non-inline methods you declared in your classes.

Before you decide to copy/paste the following code, remember that this is a PDF file. Some IDEs (esp. Xcode) are notoriously bad at making certain control characters visible. So your code may fail to compile with no obvious syntax issues. I think your best bet is to read the code and type in your own version of it.

```
// -------------- Screen, friend of Shape -------------------
// A virtual screen with pixels x: 0-(w-1) and y: 0-(h-1)
// NOTE: (0,0) is the bottom left - Pixels can be any character, determined
// by each Point.
//
class Screen {
    friend class Shape;

private:
    size_t _w, _h;
    std::vector<std::vector<char>> _pix;

public:
    static const char FG = '*', BG = '.';

    Screen(size_t w, size_t h);

    size_t get_w() const { return _w; }
    size_t get_h() const { return _h; }
    std::vector<std::vector<char>>& get_pix() { return _pix; }

    void set_w(size_t w) { _w = w; }
    void set_h(size_t h) { _h = h; }

    void clear() { fill(BG);}
    void fill(char c); // TODO - implement in the cpp file
    std::string to_string() const;

    friend std::osream &operator<<(std::osream &os, const Screen &scr) {
        return os << scr.to_string();
    };

    friend class Tests; // Don't remove this line
};

// --------- Shape ---------------------------------
// Anstract base class for circle, rectangle, line, point, triangle, polygon, etc.
//
class Shape {
public:
    virtual ~Shape() {}

    virtual bool draw(Screen &scr, char ch = Screen::FG) = 0;

    friend class Tests; // Don't remove this line
};
```

```
// --------- Point ---------------------------------

class Point : public Shape {
private:
    size_t _x, _y;

public:
    Point(size_t x, size_t y) : _x(x), _y(y) {}
    virtual ~Point() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

// ----------- Line in two point notation ---------------------

class Line : public Shape {
private:
    size_t _x1, _y1, _x2, _y2;

    // Helpers
    static bool draw_by_x(Screen &scr, char ch,
                          size_t x1, size_t y1, size_t x2, size_t y2);
    static bool draw_by_y(Screen &scr, char ch,
                          size_t x1, size_t y1, size_t x2, size_t y2);

public:
    Line(size_t a, size_t b, size_t c, size_t d) : _x1(a), _y1(b), _x2(c), _y2(d) {}
    virtual ~Line() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

// ----------- Quadrilateral ------------------------------------------------

// A general quadrilateral with points (x1,y1) ... (x4,y4), clockwise
// from bottom left. For the special case when x1==x2, y2==y3, x3==x4
// and y4==y1, we'd use an Upright_Rectangle.
//
class Quadrilateral : public Shape {
private:
    size_t _x1, _y1, _x2, _y2, _x3, _y3, _x4, _y4;

public:
    Quadrilateral(size_t a, size_t b, size_t c, size_t d,
                  size_t e, size_t f, size_t g, size_t h) :
      _x1(a), _y1(b), _x2(c), _y2(d), _x3(e), _y3(f), _x4(g), _y4(h) {}
    virtual ~Quadrilateral() {}

    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

// ----------- UprightRectangle, a special Quadrilateral -----------------

// A Rectangle is a special upright Quadrilateral so we don't have to
// parameterize the constructor with a ton of numbers
//
class Upright_Rectangle : public Quadrilateral {
public:
```

```
    Upright_Rectangle(size_t x1,size_t y1, size_t x2, size_t y2) :
      Quadrilateral(x1,y1, x1,y2, x2,y2, x2,y1) {}
    virtual ~Upright_Rectangle() {}
};

// ----------- StickMan, a composite Shape ------------------------------

class Stick_Man : public Shape {
    static const size_t DEFAULT_W = 20, DEFAULT_H = 40;

private:
    size_t _x, _y, _w, _h;
    std::vector<Shape *> _parts;

public:
    Stick_Man(size_t x = 0, size_t y = 0, size_t w = DEFAULT_W, size_t h = DEFAULT_H);
    virtual ~Stick_Man(); // Needed to deallocate parts

    const std::vector<Shape *>& get_parts() const { return _parts; }
    bool draw(Screen &scr, char ch = Screen::FG);

    friend class Tests; // Don't remove
};

#endif /* Shapes_h */
```

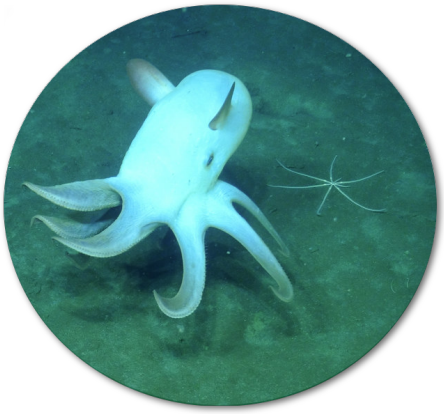
How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in under about 350 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.



1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files (`Shapes.*`) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well-thought out and helpful discussions.

Happy hacking,

&