Stacks/Queues

Recall stacks behave in FILO (first in, last out). Picture an actual **stack**. Any declared stack class would have to include the functions to

--create an empty stack

--push an item onto the stack

--pop an item from the stack

--look at top item on the stack

--return if the stack is empty

--(optional) ask how many items are in the stack

This library already exists in c++ but we will pretend it doesn’t to help us understand how it works. If it didn’t exist, how would you implement a stack? You could use an array, a linked list, or something totally different.

If you used an array you would have an array of items (the actual stack) and a pointer to point to the top item on the stack/an integer to track the index of the top of the stack. You would push items to the back of the array and increment the integer/pointer; to pop the stack, you would run the item in the current index/integer and decrement the integer/pointer. Looking at the top would be simple; call array[0]. To get how many items are in the stack, just look at the index integer; to check if it is empty, check if the integer is 0.

One possible limitation of this implementation is that the stack could actually be full – the array can reach its max size. You could go for a dynamically resizable array, but then you would have to introduce an integer to track its capacity and various destructors.

Another possible implementation of the stack is a linked list. You would probably need to keep a separate integer to keep track of how many items are in the list, making it easy to return whether the stack is empty/how many items are in the stack. If you need to push an item, create a new node and make the previous node point to it; likewise if you need to pop, just erase the final node and set the previous node’s pointer to null. The unappealing aspect of this implementation is that it can get very expensive if the list gets very long. However, fundamentally it makes sense to use a list because all the addition/removal takes place at one end. Why not make the *front* of the stack, rather than the back, the active end?

Now if you push an item on the stack you create a new node for that item and set the top of the stack head pointer point to the new item. Now the new node points to the previous head. Popping the stack simply moves the head pointer down the list one element and deletes the previous top. This is a much more efficient method of implementing a stack function using a linked list because you are not required to follow the list all the way down every time you want to pop/push.

Now we are looking at a different data structure from stack called a **queue**. This time it follows the FIFO model (first in, first out) – that is, it takes items off the back rather than the front.

Basic operations on a queue are the same as those of a stack, but they use different terminology:

--create an empty queue

--*enqueue* an item (add)

--*dequeue* an item (remove)

--look at the head item on the queue (“first in line”)

--ask if the queue is empty

-- (optional) ask how many items are in the queue

If you include the header <queue>, the standard library uses some of the same terminology as a stack (push, pop.) To look at the front you call front(), and the back is back() (not so commonly used.)

However there are a variety of ways to implement a queue (ignore the library function.) You could use an array, and enqueuing an item would be trivial, but this would require “sliding” everything down every time you dequeued an item, which could be fairly expensive. You could use a “circular buffer” array design – keep adding items until you reach the end of the array, then start adding at the beginning again, and use 2 pointers to keep track of the head and the tail.

Of course you can always use a linked list implementation. Use a head pointer and a tail pointer, and a singly linked list; enqueueing items moves the tail pointer back one, while dequeueing items moves the head pointer back and deletes the first node. However there are plenty of special cases inherent to the linked list structure that require failsafes, such as a dummy node. Eventually we come to the conclusion that a circular doubly linked list, with a dummy node at the “front”/”back”, gets rid of almost all the special cases.

Class Rectangle  
{  
 public:  
 void move(double xnew, double ynew); //moves a rectangle to a new coordinate  
 void draw() const; //draws the rectangle  
 public:  
 double m\_x;  
 double m\_y;  
 double m\_height;  
 double m\_width;  
};

Class Circle  
{  
 public:  
 void move(double xnew, double ynew);  
 void draw() const;  
 public:  
 double m\_x;  
 double m\_y;  
 double m\_r;  
};

What if you wanted to make an array with both Circles and Rectangles? This is called the problem of *heterogenous collection* – you can’t put them both into the same array or data structure. How do we solve this?

One possibility is to just have a bunch of collections--  
Circle c[10];  
Rectangle r[10];

For (int k = 0; k < ~; k++)  
 c[k].draw();  
for (int j = 0; j < ~; j++)  
 r[j].draw();

This works but is obnoxious and inflexible to changing requirements. Introducing a new class or new code somewhere will be tedious and probably introduce bugs, simply because it does not scale well at all.

In English, what would we call a group of these items (circles and rectangles)? Shapes – the term shape is sort of a generalization, while the terms circle, rectangle, square, etc. are more specialized indicators. C++ calls them, respectively, *base classes* and *derived classes.* The relationship between a derived class and a base class is that a derived class is a kind of a base class.

So in this example, our picture would be an array of shape pointers that would point to different kinds of shapes. First, though, we need to introduce a class *shape*.

Class Shape  
{  
}  
Class Rectangle : public Shape { … }  
Class Circle : public Shape { … }

This is called public inheritance, because you are declaring Rectangle and Class to be derived classes of shape.   
**Derived\* -> Base\*  
Derived& -> Base&**  
Now you can declare an array of shapes and fill it with as many rectangles and circles, in any order, as you want.

Shape\* pic[100];  
pic[0] = new Circle;  
pic[1] = new Rectangle;  
etc.  
  
What if you want to call the draw function in this array? You first need to implement a draw function into the shape class so the compiler knows it is ok.  
Class Shape  
{  
 public:  
 void move(double xnew, double ynew); //this tells the compiler that every shape can move()  
 void draw() const; //every shape can call draw()  
 private:  
 double m\_x; //every shape object has this embedded in it  
 double m\_y; //same  
}  
  
Data / functions that are common to all shapes should be promoted to the Shape class so you can call it as a shape rather than a rectangle/circle; it gives you much more flexibility.

For example, to implement Rectangle, we need to write:  
Shape::move  
Shape::draw  
Rectangle::move  
Rectangle::draw

If you are happy with the shape version of a function, you don’t need to declare or implement it in the derived class – instead it is inherited automatically. Only declare and implement it in the class if the overall Shape function is not acceptable for that specific derived class.  
So in this case, we only really need  
Shape::move  
Shape::draw  
because they are both acceptable for the rectangle class – it doesn’t require any specific instructions.