1. This is a good example of *recursion*. If you could compute the early Fibonnaci numbers, then you could compute the later ones as well. We have, roughly,

```
long Fib(short n){
return( Fib(n-1)+Fib(n-2));
}
```

The only reason this fails is that we never exit, so we need to provide an alternate branch to exit. A wrinkle here is that we have to compute both `Fib(n-1)` and `Fib(n-2)` so we need two stops:

```
long Fib(short n){
if(n>=2) {return( Fib(n-1)+Fib(n-2)); }
else if(n==1) {return 1; }
else {return 1; }
}
```

A few remarks. The last `else` is not required because of the `return`'s but it is always a good idea to write code which mimics your logic. Equally, one could replace the last `else` with `else if(n==0)` but this would not be as robust. If you make the change and then somewhere write `Fib(-2);` it's back to an infinite loop. The version presented returns 1. In a production version of `Fib` you would want to handle such problems more gracefully, either by defining Fibonacci numbers with negative subscripts and returning that value or by checking for input which is out of range and returning some sort of indicator.

2. First of all, the values of `index` indicated in the problem are wrong - they were meant to be 0,$\frac{1}{3}$, $\frac{2}{3}$. This does not really effect the point of the question which deals with the finite accuracy of computers. You can see the problem just using decimal notation. The first value of `index` is 0:0, but the second is some *finite* decimal approximation to $\frac{1}{3}$, say 0:3333. Then the third value of `index` is 0:6666. So far nothing bad has happened, but look what happens next: `index` becomes 0:9999 and we check if this is 1:0. It is not so we loop again and it is clear that we always get `index!=1.0`. The fact that the computer is using some finite *binary* approximation to $\frac{1}{3}$ does not affect the point of the discussion.

To fix the loop requires some care: changing the test to `index<=1.0` will kill the infinite loop but we will do one more pass through the loop than we wanted to do. Something like `index>0.9` does the trick (any number greater than the value after two iterations but less than the value after three iterations can be used for the `0.9`).

3. A. and B. are two ways of writing the same thing and both compile just fine. C. is an illegal declaration because when the program is *compiled* the compiler does not know how much space to set aside for the `v` array. Remember that *declarations* tell the compiler how much space to set aside for each declared variable.

4. This is an arithmetic exercise: after line two,

```
i=1; j=3; k=2;
After the first command on line three,
i=2; j=3; k=2;
After the second command,
i=2; j=5; k=2;
After the first command on line three,
i=2; j=5; k=10;
Finally, we have
i=2; j=5; k=9;
```

5. One can just grind through the steps in the code, but a bit of forethought makes life easier (as it so often does). The `x` box contains the value the function `ARD(1.3,2.4,6)` returns, so let's start by looking for the `return` statements in the code. Life is easy since there is only one, `return (n);`. Hence we need only figure out what the value of `n` is - `x` and `y` are irrelevant. When the function starts up, `n=6`. After the declaration and the first line of code, the value of `n` is still `6`. After the second line of code, `n` is `6+6` or `12` and this is the value returned. Hence `x=12`. In particular, all the thrashing around with `x`, `y` and `temp` are irrelevant.

6. 6. uses a feature of C we have not yet discussed: we may make one statement out of several potential statements by tying them together with commas rather than semi-colons. For example, the initialization part of the `for` loop has two pieces: `i` gets set to `0` and `y` get set to `1.2`. Similarly, the iteration step has two pieces. The `for` loop has an initialization, a body, an iteration and a test; the `while` loop only has a test, so we have to do the other parts explicitly. Here is one possible solution

```
   i=0; y=1.2; /* initialization */
   while(i<33){/* test */
      Some Code
      i+=2; x=x*x+1; /* iteration */
      }
```