The “let” block; a simple way to store values in functions
We often want to store the “internal state” of some function without having to pass that state as an argument
For example, rand() can be called without any argument - but clearly there must be some internal state that is somehow saved
References to data (pointers) can be permanently saved in “let” blocks - functions defined inside a let block can access these pointers
Example, part of letblock.jl (random number generator, inside a module)

```julia
let
    r = Ref(convert(UInt64,1))
    global function ran64()
        r[] = r[]*a+c
end
end
```

r is a reference (pointer) to an unsigned integer - the value at r is accessed by r[] - would be r[i] for element i of a 1-dim array
Why not just use r declared in the global scope? - for efficiency, avoid using global variables

The function must be declared global to make it accessible outside let-end - global function objects are treated as constants, not slowing things down - the integers a and c are declared as constants before let
The let block is a local hard scope, many other uses (see Julia doc)
**Romberg integration**

Idea: Use two or more trapezoidal integral estimates, extrapolate
- step sizes (decreasing order) \(h_0, h_1, \ldots, h_m\), integral estimates \(I_0, I_1, \ldots, I_m\)
- use polynomial of order \(n\) to fit and extrapolate to \(h=0\)
- error for given \(h\) scales as \(h^2\) (+ higher even powers only)
- use polynomial \(P(x)\) with \(x=h^2\)

Simplest case: 2 points (\(m=1\)), using \(h_0= (b-a)/n_0\) and \(h_1= h_0/2\) (\(x_1=x_0/4\))

\[
\begin{align*}
0 & \quad 1 & \quad 2 & \quad n_1 = 2n_0 \\
0 & \quad 1 & \quad n_0
\end{align*}
\]

Function evaluation once only for each point needed

\[
I_0 = I_\infty + \epsilon x_0, \quad I_1 = I_\infty + \epsilon x_0/4
\]

\[
\Rightarrow \quad I_\infty = \frac{4}{3} I_1 - \frac{1}{3} I_0 + O(h_0^4) \quad [O(x_0^2)]
\]

reducing \(h\) by 50%  
- error should be 1/4 of previous 
- \(\epsilon\) is unknown factor, eliminated

Computation cost doubled, error reduced by two powers of \(h_0\)!

Generalizes easily to the case of \(m\) estimates (Friday)
General case; \( h_0, h_1, \ldots, h_m \rightarrow I_1, I_2, \ldots, I_m \)

For each \( i \), let \( h_{i+1} = h_i/2 \) \((x_{i+1} = x_i/4)\)
- save old sum, add new points

\[
\begin{array}{cccc}
0 & 1 & 2 & \ldots & n_0 \rightarrow n_1 = 2n_0
\end{array}
\]

How to construct a polynomial of order \( n \) going through \( n+1 \) point pairs \( (x_i, y_i) \)

\[
P(x) = \sum_{i=0}^{m} y_i \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} \quad x_i = h_i^2
\]

Evaluate (this is the extrapolation) at \( x=0 \) \((h=0)\)

\[
I_\infty = \sum_{i=0}^{m} I_i \prod_{k \neq i} \frac{-h_0^2 2^{-2k}}{h_0^2 (2^{-2i} - 2^{-2k})} = (-1)^m \sum_{i=0}^{m} I_i \prod_{k \neq i} \frac{1}{2^{2(k-i)} - 1}
\]

Error decreases very rapidly: \( O(h^{2(m+1)}) \)

Implemented in romberg.jl (both closed and open cases)