```
44
0x3fffcc4, 44
50
50 was not found.
99
0x3fffcd8, 99
90
90 was not found.
0
0 was not found.
```

7.3 We use a `for` loop to traverse the array until `p` points to the `target`:

```
float* duplicate(float* p[], int n)
{ float* const b = new float[n];
  for (int i = 0; i < n; i++)
    b[i] = *p[i];
  return b;
}

void print(float [], int);
void print(float* [], int);

int main()
{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i];  // p[i] points to a[i]
  print(p, 8);
  float* const b = duplicate(p, 8);
  print(b, 8);
}
```

```
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
   44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
```

7.4 This function, named `riemann()`, is similar to the `sum()` function in Example 7.18. Its first argument is a pointer to a function that has one `double` argument and returns a `double`. In this test run, we pass it (a pointer to) the `cube()` function. The other three arguments are the boundaries `a` and `b` of the interval [*a*, *b*] over which the integration is being performed and the number `n` of subintervals to be used in the sum. The actual Riemann sum is the sum of the areas of the `n` rectangles based on these subintervals whose heights are given by the function being integrated:

```
double riemann(double (*)(double), double, double, int);
double cube(double);

int main()
{ cout << riemann(cube,0,2,10) << endl;
  cout << riemann(cube,0,2,100) << endl;
  cout << riemann(cube,0,2,1000) << endl;
  cout << riemann(cube,0,2,10000) << endl;
}

//  Returns [f(a)*h + f(a+h)*h + f(a+2h)*h + . . . + f(b-h)*h],
//  where h = (b-a)/n:
```

```
double riemann(double (*pf)(double t), double a, double b, int n)
{ double s = 0, h = (b-a)/n, x;
  int i;
  for (x = a, i = 0; i < n; x += h, i++)
    s += (*pf)(x);
  return s*h;
}

double cube(double t)
{ return t*t*t;
}
```

```
   3.24
   3.9204
   3.992
   3.9992
```

In this test run, we are integrating the function $y = x^3$ over the interval [0, 2]. By elementary calculus, the value of this integral is 4.0. The call `riemann(cube,0,2,10)` approximates this integral using 10 subintervals, obtaining 3.24. The call `riemann(cube,0,2,100)` approximates the integral using 100 subintervals, obtaining 3.9204. These sums get closer to their limit 4.0 as `n` increases. With 10,000 subintervals, the Riemann sum is 3.9992. Note that the only significant difference between this `riemann()` function and the `sum()` function in Example 7.18 is that the sum is multiplied by the subinterval width `h` before being returned.

7.5 This `derivative()` function is similar to the `sum()` function in Example 7.18, except that it implements the formula for the numerical derivative instead. It has three arguments: a pointer to the function *f*, the *x* value, and the tolerance *h*. In this test run, we pass it (pointers to) the `cube()` function and the `sqrt()` function.

```
#include <iostream>
#include <cmath>
using namespace std;
double derivative(double (*)(double), double, double);
double cube(double);

int main()
{ cout << derivative(cube, 1, 0.1) << endl;
  cout << derivative(cube, 1, 0.01) << endl;
  cout << derivative(cube, 1, 0.001) << endl;
  cout << derivative(sqrt, 1, 0.1) << endl;
  cout << derivative(sqrt, 1, 0.01) << endl;
  cout << derivative(sqrt, 1, 0.001) << endl;
}

//  Returns an approximation to the derivative f'(x):
double derivative(double (*pf)(double t), double x, double h)
{ return ((*pf)(x+h) - (*pf)(x-h))/(2*h);
}

double cube(double t)
{ return t*t*t;
}
```

```
    3.01
    3.0001
    3
    0.500628
    0.500006
    0.5
```

The derivative of the `cube()` function x^3 is $3x^2$, and its value at $x = 1$ is 3, so the numerical derivative should be close to 3.0 for small h. Similarly, the derivative of the `sqrt()` function $\sqrt{x}$ is $1/(2\sqrt{x})$, and its value at $x = 1$ is 1/2, so its numerical derivative should be close to 0.5 for small h.

7.6 The pointer `pmax` is used to locate the maximum `float`. It is initialized to have the same value as `p[0]` which points to the first `float`. Then inside the `for` loop, the `float` to which `p[i]` points is compared to the `float` to which `pmax` points, and `pmax` is updated to point to the larger `float` when it is detected. So when the loop terminates, `pmax` points to the largest `float`:

```
float* max(float* p[], int n)
{ float* pmax = p[0];
  for (int i = 1; i < n; i++)
  if (*p[i] > *pmax) pmax = p[i];
    return pmax;
}

void print(float [], int);
void print(float* [], int);

int main()
{ float a[8] = {44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5};
  print(a, 8);
  float* p[8];
  for (int i = 0; i < 8; i++)
    p[i] = &a[i];  // p[i] points to a[i]
  print(p, 8);
  float* m = max(p, 8);
  cout << m << ", " << *m << endl;
}
```

```
    44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
    44.4, 77.7, 22.2, 88.8, 66.6, 33.3, 99.9, 55.5
    0x3fffcd4, 99.9
```

Here we have two (overloaded) `print()` functions: one to print the array of pointers, and one to print the `float`s to which they point. After initializing and printing the array `a`, we define the array `p` and initialize its elements to point to the elements of `a`. The call `print(p, 8)` verifies that `p` provides *indirect access* to `a`. Finally, the pointer `m` is declared and initialized with the address returned by the `max()` function. The last output verifies that `m` does indeed point to the largest `float` among those accessed by `p`.

Solutions to Problems 7.7-7.24 are available on-line at `projectEuclid.net`.