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

```cpp
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;
}
```

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

```cpp
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);
}
```

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:

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

```cpp
double cube(double);
```

```cpp
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 \[ f(a)h + f(a+h)h + f(a+2h)h + \ldots + f(b-h)h \],
// 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;
}

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 \( \text{riemann}(\text{cube}, 0, 2, 10) \) approximates this integral using 10 subintervals, obtaining 3.24. The call \( \text{riemann}(\text{cube}, 0, 2, 100) \) 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 \text{riemann()} function and the \text{sum()} function in Example 7.18 is that the sum is multiplied by the subinterval width \( h \) before being returned.

7.5 This \text{derivative()} function is similar to the \text{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 \text{cube()} function and the \text{sqrt()} function.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

double derivative(double (*)(double), double, double);
double cube(double t)
{ return t*t*t;
}

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

\begin{verbatim}
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;
}
\end{verbatim}

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;
}

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

\textbf{Solutions to Problems 7.7-7.24 are available on-line at projectEuclid.net.}