```
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}
```

Here is the output of the program in Listing 7.17:

```
|                                                               |
|                               |                               |
|               |               |               |               |
|       |       |       |       |       |       |       |       |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Program Notes

The `subdivide()` function in Listing 7.17 uses the variable `level` to control the recursion level. When the function calls itself, it reduces `level` by one, and the function with a `level` of `0` terminates. Note that `subdivide()` calls itself twice, once for the left subdivision and once for the right subdivision. The original midpoint becomes the right end for one call and the left end for the other call. Notice that the number of calls grows geometrically. That is, one call generates two, which generate four calls, which generate eight, and so on. That's why the level 6 call is able to fill in 64 elements ($2^6 = 64$). This continued doubling of the number of function calls (and hence of the number of variables stored) make this form of recursion a poor choice if many levels of recursion are required. But it is an elegant and simple choice if the necessary levels of recursion are few.

Pointers to Functions

No discussion of C or C++ functions would be complete without mention of pointers to functions. We'll take a quick look at this topic and leave the full exposition of the possibilities to more advanced texts.

Functions, like data items, have addresses. A function's address is the memory address at which the stored machine language code for the function begins. Normally, it's neither important nor useful for you or the user to know that address, but it can be useful to a program. For example, it's possible to write a function that takes the address of another function as an argument. That enables the first function to find the second function and run it. This approach is more awkward than simply having the first function call the second one directly, but it leaves open the possibility of passing different function addresses at different times. That means the first function can use different functions at different times.

Function Pointer Basics

Let's clarify this process with an example. Suppose you want to design an `estimate()` function that estimates the amount of time necessary to write a given number of lines of code, and you want different programmers to use the function. Part of the code for `estimate()` will be the same for all users, but the function will allow each programmer to provide his or her own algorithm for estimating time. The mechanism for that will be to pass to `estimate()` the address of the particular algorithm function the programmer wants to use. To implement this plan, you need to be able to do the following:

- Obtain the address of a function.
- Declare a pointer to a function.
- Use a pointer to a function to invoke the function.

Obtaining the Address of a Function

Obtaining the address of a function is simple: You just use the function name without trailing parentheses. That is, if `think()` is a function, then `think` is the address of the function. To pass a function as an argument, you pass the function name. Be sure you distinguish between passing the *address* of a function and passing the *return value* of a function:

```
process(think);    // passes address of think() to process()
thought(think());  // passes return value of think() to thought()
```

The `process()` call enables the `process()` function to invoke the `think()` function from within `process()`. The `thought()` call first invokes the `think()` function and then passes the return value of `think()` to the `thought()` function.

Declaring a Pointer to a Function

To declare pointers to a data type, the declaration has had to specify exactly to what type the pointer points. Similarly, a pointer to a function has to specify to what type of function the pointer points. This means the declaration should identify the function's return type and the function's signature (its argument list). That is, the declaration should provide the same information about a function that a function prototype does. For example, suppose Pam LeCoder has written a time-estimating function with the following prototype:

```
double pam(int);  // prototype
```

Here's what a declaration of an appropriate pointer type looks like:

```
double (*pf)(int);   // pf points to a function that takes
                     // one int argument and that
                     // returns type double
```

> **Tip**
>
> In general, to declare a pointer to a particular kind of function, you can first write a prototype for a regular function of the desired kind and then replace the function name with an expression in the form `(*pf)`. In this case, `pf` is a pointer to a function of that type.

The declaration requires the parentheses around `*pf` to provide the proper operator precedence. Parentheses have a higher precedence than the `*` operator, so `*pf(int)` means `pf()` is a function that returns a pointer, whereas `(*pf)(int)` means `pf` is a pointer to a function:

```
double (*pf)(int); // pf points to a function that returns double
double *pf(int);   // pf() a function that returns a pointer-to-double
```

After you declare `pf` properly, you can assign to it the address of a matching function:

```
double pam(int);
double (*pf)(int);
pf = pam;           // pf now points to the pam() function
```

Note that `pam()` has to match `pf` in both signature and return type. The compiler rejects nonmatching assignments:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;         // invalid -- mismatched signature
pf = ted;         // invalid -- mismatched return types
```

Let's return to the `estimate()` function mentioned earlier. Suppose you want to pass to it the number of lines of code to be written and the address of an estimating algorithm, such as the `pam()` function. It could have the following prototype:

```
void estimate(int lines, double (*pf)(int));
```

This declaration says the second argument is a pointer to a function that has an `int` argument and a `double` return value. To have `estimate()` use the `pam()` function, you pass `pam()`'s address to it:

```
estimate(50, pam); // function call telling estimate() to use pam()
```

Clearly, the tricky part about using pointers to functions is writing the prototypes, whereas passing the address is very simple.

Using a Pointer to Invoke a Function

Now we get to the final part of the technique, which is using a pointer to call the pointed-to function. The clue comes in the pointer declaration. There, recall, `(*pf)` plays the same role as a function name. Thus, all you have to do is use `(*pf)` as if it were a function name:

```
double pam(int);
double (*pf)(int);
pf = pam;            // pf now points to the pam() function
double x = pam(4);   // call pam() using the function name
double y = (*pf)(5); // call pam() using the pointer pf
```

Actually, C++ also allows you to use `pf` as if it were a function name:

```
double y = pf(5);    // also call pam() using the pointer pf
```

Using the first form is uglier, but it provides a strong visual reminder that the code is using a function pointer.

> ### History Versus Logic
>
> Holy syntax! How can `pf` and `(*pf)` be equivalent? One school of thought maintains that because `pf` is a pointer to a function, `*pf` is a function; hence, you should use `(*pf)()` as a function call. A second school maintains that because the name of a function is a pointer to that function, a pointer to that function should act like the name of a function; hence you should use `pf()` as a function call. C++ takes the compromise view that both forms are correct, or at least can be allowed, even though they are logically inconsistent with each other. Before you judge that compromise too harshly, reflect that the ability to hold views that are not logically self-consistent is a hallmark of the human mental process.

A Function Pointer Example

Listing 7.18 demonstrates using function pointers in a program. It calls the `estimate()` function twice, once passing the `betsy()` function address and once passing the `pam()` function address. In the first case, `estimate()` uses `betsy()` to calculate the number of hours necessary, and in the second case, `estimate()` uses `pam()` for the calculation. This design facilitates future program development. When Ralph develops his own algorithm for estimating time, he doesn't have to rewrite `estimate()`. Instead, he merely needs to supply his own `ralph()` function, making sure it has the correct signature and return type. Of course, rewriting `estimate()` isn't a difficult task, but the same principle applies to more complex code. Also the function pointer method allows Ralph to modify the behavior of `estimate()`, even if he doesn't have access to the source code for `estimate()`.

Listing 7.18 **`fun_ptr.cpp`**

```
// fun_ptr.cpp -- pointers to functions
#include <iostream>
double betsy(int);
double pam(int);

// second argument is pointer to a type double function that
// takes a type int argument
void estimate(int lines, double (*pf)(int));

int main()
{
    using namespace std;
    int code;
```

```
    cout << "How many lines of code do you need? ";
    cin >> code;
    cout << "Here's Betsy's estimate:\n";
    estimate(code, betsy);
    cout << "Here's Pam's estimate:\n";
    estimate(code, pam);
    return 0;
}

double betsy(int lns)
{
    return 0.05 * lns;
}

double pam(int lns)
{
    return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate(int lines, double (*pf)(int))
{
    using namespace std;
    cout << lines << " lines will take ";
    cout << (*pf)(lines) << " hour(s)\n";
}
```

Here is a sample run of the program in Listing 7.18:

```
How many lines of code do you need? 30
Here's Betsy's estimate:
30 lines will take 1.5 hour(s)
Here's Pam's estimate:
30 lines will take 1.26 hour(s)
```

Here is a second sample run of the program:

```
How many lines of code do you need? 100
Here's Betsy's estimate:
100 lines will take 5 hour(s)
Here's Pam's estimate:
100 lines will take 7 hour(s)
```

Variations on the Theme of Function Pointers

With function pointers, the notation can get intimidating. Let's look at an example that illustrates some of the challenges of function pointers and ways of dealing with them. To begin, here are prototypes for some functions that share the same signature and return type:

```
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);
```

The signatures might look different, but they are the same. First, recall that in a function prototype parameter list `const double ar[]` and `const double * ar` have exactly the same meaning. Second, recall that in a prototype you can omit identifiers. Therefore, `const double ar[]` can be reduced to `const double []`, and `const double * ar` can be reduced to `const double *`. So all the function signatures shown previously have the same meaning. Function definitions, on the other hand, do provide identifiers, so either `const double ar[]` or `const double * ar` will serve in that context.

Next, suppose you wish to declare a pointer that can point to one of these three functions. The technique, you'll recall, is if `pa` is the desired pointer, take the prototype for a target function and replace the function name with `(*pa)`:

```
const double * (*p1)(const double *, int);
```

This can be combined with initialization:

```
const double * (*p1)(const double *, int) = f1;
```

With the C++11 automatic type deduction feature, you can simplify this a bit:

```
auto p2 = f2;  // C++11 automatic type deduction
```

Now consider the following statements:

```
cout <<  (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
cout << p2(av,3) << ": " << *p2(av,3) << endl;
```

Both `(*p1)(av,3)` and `p2(av,3)`, recall, represent calling the pointed-to functions (`f1()` and `f2()`, in this case) with `av` and `3` as arguments. Therefore, what should print are the return values of these two functions. The return values are type `const double *` (that is, address of `double` values). So the first part of each `cout` expression should print the address of a `double` value. To see the actual value stored at the addresses, we need to apply the `*` operator to these addresses, and that's what the expressions `*(*p1)(av,3)` and `*p2(av,3)` do.

With three functions to work with, it could be handy to have an array of function pointers. Then one can use a `for` loop to call each function, via its pointer, in turn. What would that look like? Clearly, it should look something like the declaration of a single pointer, but there should be a `[3]` somewhere to indicate an array of three pointers. The question is where. And here's the answer (including initialization):

```
const double * (*pa[3])(const double *, int) = {f1,f2,f3};
```

Why put the `[3]` there? Well, `pa` is an array of three things, and the starting point for declaring an array of three things is this: `pa[3]`. The rest of the declaration is about what kind of thing is to be placed in the array. Operator precedence ranks `[]` higher than `*`, so `*pa[3]` says `pa` is an array of three pointers. The rest of the declaration indicates what each pointer points to: a function with a signature of `const double *, int` and a return type of `const double *`. Hence, `pa` is an array of three pointers, each of which is a pointer to a function that takes a `const double *` and `int` as arguments and returns a `const double *`.

Can we use `auto` here? No. Automatic type deduction works with a single initializer value, not an initialization list. But now that we have the array `pa`, it is simple to declare a pointer of the matching type:

```
auto pb = pa;
```

The name of an array, as you'll recall, is a pointer to its first element, so both `pa` and `pb` are pointers to a pointer to a function.

How can we use them to call a function? Both `pa[i]` and `pb[i]` represent pointers in the array, so you can use either of the function call notations with either of them:

```
const double * px = pa[0](av,3);
const double * py = (*pb[1])(av,3);
```

And you can apply the `*` operator to get the pointed-to `double` value:

```
double x = *pa[0](av,3);
double y = *(*pb[1])(av,3);
```

Something else you can do (and who wouldn't want to?) is create a pointer to the whole array. Because the array name `pa` already is a pointer to a function pointer, a pointer to the array would be a pointer to a pointer to a pointer. This sounds intimidating, but because the result can be initialed with a single value, you can use `auto`:

```
auto pc = &pa;  // C++11 automatic type deduction
```

What if you prefer to do it yourself? Clearly, the declaration should resemble the declaration for `pa`, but because there is one more level of indirection, we'll need one more `*` stuck somewhere. In particular, if we call the new pointer `pd`, we need to indicate that it is pointer, not an array name. This suggests the heart of the declaration should be `(*pd)[3]`. The parentheses bind the `pd` identifier to the `*`:

```
*pd[3]    // an array of 3 pointers
(*pd)[3]  // a pointer to an array of 3 elements
```

In other words, `pd` is a pointer, and it points to an array of three things. What those things are is described by the rest of the original declaration of `pa`. This approach yields the following:

```
const double *(*(*pd)[3])(const double *, int) = &pa;
```

To call a function, realize that if `pd` points to an array, then `*pd` is the array and `(*pd)[i]` is an array element, which is a pointer to a function. The simpler notation, then, for the function call is `(*pd)[i](av,3)`, and `*(*pd)[i](av,3)` would be the value that the returned pointer points to. Alternatively, you could use second syntax for invoking a function with a pointer and use `(*(*pd)[i])(av,3)` for the call and `*(*(*pd)[i])(av,3)` for the pointed-to `double` value.

Be aware of the difference between `pa`, which as an array name is an address, and `&pa`. As you've seen before, in most contexts `pa` is the address of the first element of the array—that is, `&pa[0]`. Therefore, it is the address of a single pointer. But `&pa` is the address of the entire array (that is, of a block of three pointers). Numerically, `pa` and `&pa` may have the same value, but they are of different types. One practical difference is that `pa+1` is the address of the next element in the array, whereas `&pa+1` is the address of the next block of 12 bytes (assuming addresses are 4 bytes) following the `pa` array. Another difference is that you dereference `pa` once to get the value of the first element and you deference `&pa` twice to get the same value:

```
**&pa == *pa == pa[0]
```

Listing 7.19 puts this discussion to use. For illustrative purposes, the functions `f1()`, and so on, have been kept embarrassingly simple. The program shows, as comments, the C++98 alternatives to using `auto`.

Listing 7.19 **arfupt.cpp**

```
// arfupt.cpp -- an array of function pointers
#include <iostream>
// various notations, same signatures
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);

int main()
{
    using namespace std;
    double av[3] = {1112.3, 1542.6, 2227.9};

    // pointer to a function
    const double *(*p1)(const double *, int) = f1;
    auto p2 = f2;  // C++11 automatic type deduction
    // pre-C++11 can use the following code instead
    // const double *(*p2)(const double *, int) = f2;
    cout << "Using pointers to functions:\n";
    cout << " Address  Value\n";
    cout <<  (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
    cout << p2(av,3) << ": " << *p2(av,3) << endl;

    // pa an array of pointers
    // auto doesn't work with list initialization
    const double *(*pa[3])(const double *, int) = {f1,f2,f3};
    // but it does work for initializing to a single value
    // pb a pointer to first element of pa
    auto pb = pa;
    // pre-C++11 can use the following code instead
```

```
    // const double *(**pb)(const double *, int) = pa;
    cout << "\nUsing an array of pointers to functions:\n";
    cout << " Address  Value\n";
    for (int i = 0; i < 3; i++)
        cout << pa[i](av,3) << ": " << *pa[i](av,3) << endl;
    cout << "\nUsing a pointer to a pointer to a function:\n";
    cout << " Address  Value\n";
    for (int i = 0; i < 3; i++)
        cout << pb[i](av,3) << ": " << *pb[i](av,3) << endl;

    // what about a pointer to an array of function pointers
    cout << "\nUsing pointers to an array of pointers:\n";
    cout << " Address  Value\n";
    // easy way to declare pc
    auto pc = &pa;
    // pre-C++11 can use the following code instead
    // const double *(*(*pc)[3])(const double *, int) = &pa;
    cout << (*pc)[0](av,3) << ": " << *(*pc)[0](av,3) << endl;
    // hard way to declare pd
    const double *(*(*pd)[3])(const double *, int) = &pa;
    // store return value in pdb
    const double * pdb = (*pd)[1](av,3);
    cout << pdb << ": " << *pdb << endl;
    // alternative notation
    cout << (*(*pd)[2])(av,3) << ": " << *(*(*pd)[2])(av,3) << endl;
    // cin.get();
    return 0;
}

// some rather dull functions

const double * f1(const double * ar, int n)
{
    return ar;
}
const double * f2(const double ar[], int n)
{
    return ar+1;
}
const double * f3(const double ar[], int n)
{
    return ar+2;
}
```

And here is the output:

```
Using pointers to functions:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6

Using an array of pointers to functions:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9

Using a pointer to a pointer to a function:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9

Using pointers to an array of pointers:
 Address  Value
002AF9E0: 1112.3
002AF9E8: 1542.6
002AF9F0: 2227.9
```

The addresses shown are the locations of the `double` values in the `av` array.

This example may seem esoteric, but pointers to arrays of pointers to functions are not unheard of. Indeed, the usual implementation of virtual class methods (see Chapter 13, "Class Inheritance") uses this technique. Fortunately, the compiler handles the details.

Appreciating `auto`

One of the goals of C++11 is to make C++ easier to use, letting the programmer concentrate more on design and less on details. Listing 7.19 surely illustrates this point:

```
auto pc = &pa;                                   // C++11 automatic type deduction
const double *(*(*pd)[3])(const double *, int) = &pa; // C++98, do it yourself
```

The automatic type deduction feature reflects a philosophical shift in the role of the compiler. In C++98, the compiler uses its knowledge to tell you when you are wrong. In C++11, at least with this feature, it uses its knowledge to help you get the right declaration.

There is a potential drawback. Automatic type deduction ensures that the type of the variable matches the type of the initializer, but it still is possible that you might provide the wrong type of initializer:

```
auto pc = *pa;   // oops! used *pa instead of &pa
```

This declaration would make `pc` match the type of `*pa`, and that would result in a compile-time error when Listing 7.19 later uses `pc`, assuming that it is of the same type as `&pa`.

Simplifying with `typedef`

C++ does provide tools other than `auto` for simplifying declarations. You may recall from Chapter 5, "Loops and Relational Expressions," that the `typedef` keyword allows you to create a type alias:

```
typedef double real; // makes real another name for double
```

The technique is to declare the alias as if it were an identifier and to insert the keyword `typedef` at the beginning. So you can do this to make `p_fun` an alias for the function pointer type used in Listing 7.19:

```
typedef const double *(*p_fun)(const double *, int);  // p_fun now a type name
p_fun p1 = f1;  // p1 points to the f1() function
```

You then can use this type to build elaborations:

```
p_fun pa[3] = {f1,f2,f3}; // pa an array of 3 function pointers
p_fun (*pd)[3] = &pa;     // pd points to an array of 3 function pointers
```

Not only does `typedef` save you some typing, it makes writing the code less error prone, and it makes the program easier to understand.

Summary

Functions are the C++ programming modules. To use a function, you need to provide a definition and a prototype, and you have to use a function call. The function definition is the code that implements what the function does. The function prototype describes the function interface: how many and what kinds of values to pass to the function and what sort of return type, if any, to get from it. The function call causes the program to pass the function arguments to the function and to transfer program execution to the function code.

By default, C++ functions pass arguments by value. This means that the formal parameters in the function definition are new variables that are initialized to the values provided by the function call. Thus, C++ functions protect the integrity of the original data by working with copies.

C++ treats an array name argument as the address of the first element of the array. Technically, this is still passing by value because the pointer is a copy of the original address, but the function uses the pointer to access the contents of the original array. When you declare formal parameters for a function (and only then), the following two declarations are equivalent:

```
typeName arr[];
typeName * arr;
```

Both of these mean that `arr` is a pointer to *`typeName`*. When you write the function code, however, you can use `arr` as if it were an array name in order to access elements: `arr[i]`. Even when passing pointers, you can preserve the integrity of the original data by declaring the formal argument to be a pointer to a `const` type. Because passing the