CHAPTER 6

FUNCTIONS

CONTENTS

This chapter describes how to define and declare functions. We'll cover how arguments are passed to and values returned from functions. In C++, functions can be overloaded, which means that we can use the same name for several different functions. We'll cover both how to overload functions and how the compiler selects the matching version for a particular call from several overloaded functions. The chapter closes by describing pointers to functions.

A function is a block of code with a name. We execute the code by calling the function. A function may take zero or more arguments and (usually) yields a result. Functions can be overloaded, meaning that the same name may refer to several different functions.

6.1 Function Basics

A ***function*** definition typically consists of a ***return type***, a name, a list of zero or more ***parameters***, and a body. The parameters are specified in a comma-separated list enclosed in parentheses. The actions that the function performs are specified in a statement block (§ 5.1, p. 173), referred to as the ***function body***.

We execute a function through the **call operator**, which is a pair of parentheses. The call operator takes an expression that is a function or points to a function. Inside the parentheses is a comma-separated list of ***arguments***. The arguments are used to initialize the function's parameters. The type of a call expression is the return type of the function.

Writing a Function

As an example, we'll write a function to determine the factorial of a given number. The factorial of a number `n` is the product of the numbers from 1 through `n`. The factorial of 5, for example, is 120.

```
1 * 2 * 3 * 4 * 5 = 120
```

We might define this function as follows:

```
// factorial of val is val * (val - 1) * (val - 2) ... * ((val - (val - 1)) * 1)
int fact(int val)
{
    int ret = 1; // local variable to hold the result as we calculate it
    while (val > 1)
        ret *= val--;  // assign ret * val to ret and decrement val
    return ret;        // return the result
}
```

Our function is named `fact`. It takes one `int` parameter and returns an `int` value. Inside the `while` loop, we compute the factorial using the postfix decrement operator (§ 4.5, p. 147) to reduce the value of `val` by 1 on each iteration. The `return` statement ends execution of `fact` and returns the value of `ret`.

Calling a Function

To call `fact`, we must supply an `int` value. The result of the call is also an `int`:

```
int main()
{
    int j = fact(5);  // j equals 120, i.e., the result of fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
```

A function call does two things: It initializes the function's parameters from the corresponding arguments, and it transfers control to that function. Execution of the *calling* function is suspended and execution of the *called* function begins.

Execution of a function begins with the (implicit) definition and initialization of its parameters. Thus, when we call `fact`, the first thing that happens is that an `int` variable named `val` is created. This variable is initialized by the argument in the call to `fact`, which in this case is `5`.

Execution of a function ends when a `return` statement is encountered. Like a function call, the `return` statement does two things: It returns the value (if any) in the `return`, and it transfers control out of the *called* function back to the *calling* function. The value returned by the function is used to initialize the result of the call expression. Execution continues with whatever remains of the expression in which the call appeared. Thus, our call to `fact` is equivalent to the following:

```
int val = 5;        // initialize val from the literal 5
int ret = 1;        // code from the body of fact
while (val > 1)
    ret *= val--;
int j = ret;        // initialize j as a copy of ret
```

Parameters and Arguments

Arguments are the initializers for a function's parameters. The first argument initializes the first parameter, the second argument initializes the second parameter, and so on. Although we know which argument initializes which parameter, we have no guarantees about the order in which arguments are evaluated (§ 4.1.3, p. 137). The compiler is free to evaluate the arguments in whatever order it prefers.

The type of each argument must match the corresponding parameter in the same way that the type of any initializer must match the type of the object it initializes. We must pass exactly the same number of arguments as the function has parameters. Because every call is guaranteed to pass as many arguments as the function has parameters, parameters are always initialized.

Because `fact` has a single parameter of type `int`, every time we call it we must supply a single argument that can be converted (§ 4.11, p. 159) to `int`:

```
fact("hello");        // error: wrong argument type
fact();               // error: too few arguments
fact(42, 10, 0);      // error: too many arguments
fact(3.14);           // ok: argument is converted to int
```

The first call fails because there is no conversion from `const char*` to `int`. The second and third calls pass the wrong number of arguments. The `fact` function must be called with one argument; it is an error to call it with any other number. The last call is legal because there is a conversion from `double` to `int`. In this call, the argument is implicitly converted to `int` (through truncation). After the conversion, this call is equivalent to

```
fact(3);
```

Function Parameter List

A function's parameter list can be empty but cannot be omitted. Typically we define a function with no parameters by writing an empty parameter list. For compatibility with C, we also can use the keyword `void` to indicate that there are no parameters:

```
void f1(){ /* ... */ }      // implicit void parameter list
void f2(void){ /* ... */ } // explicit void parameter list
```

A parameter list typically consists of a comma-separated list of parameters, each of which looks like a declaration with a single declarator. Even when the types of two parameters are the same, the type must be repeated:

```
int f3(int v1, v2) { /* ... */ }      // error
int f4(int v1, int v2) { /* ... */ } // ok
```

No two parameters can have the same name. Moreover, local variables at the outermost scope of the function may not use the same name as any parameter.

Parameter names are optional. However, there is no way to use an unnamed parameter. Therefore, parameters ordinarily have names. Occasionally a function has a parameter that is not used. Such parameters are often left unnamed, to indicate that they aren't used. Leaving a parameter unnamed doesn't change the number of arguments that a call must supply. A call must supply an argument for every parameter, even if that parameter isn't used.

Function Return Type

Most types can be used as the return type of a function. In particular, the return type can be `void`, which means that the function does not return a value. However, the return type may not be an array type (§ 3.5, p. 113) or a function type. However, a function may return a pointer to an array or a function. We'll see how to define functions that return pointers (or references) to arrays in § 6.3.3 (p. 228) and how to return pointers to functions in § 6.7 (p. 247).

6.1.1 Local Objects

In C++, names have scope (§ 2.2.4, p. 48), and objects have **lifetimes**. It is important to understand both of these concepts.

- The scope of a name is *the part of the program's text* in which that name is visible.
- The lifetime of an object is *the time during the program's execution* that the object exists.

As we've seen, the body of a function is a statement block. As usual, the block forms a new scope in which we can define variables. Parameters and variables defined inside a function body are referred to as **local variables**. They are "local" to that function and **hide** declarations of the same name made in an outer scope.

EXERCISES SECTION 6.1

Exercise 6.1: What is the difference between a parameter and an argument?

Exercise 6.2: Indicate which of the following functions are in error and why. Suggest how you might correct the problems.

(a)
```
int f() {
        string s;
        // ...
        return s;
    }
```
(b)
```
f2(int i) { /* ... */ }
```
(c)
```
int calc(int v1, int v1) /* ... */ }
```
(d)
```
double square(double x) return x * x;
```

Exercise 6.3: Write and test your own version of `fact`.

Exercise 6.4: Write a function that interacts with the user, asking for a number and generating the factorial of that number. Call this function from `main`.

Exercise 6.5: Write a function to return the absolute value of its argument.

Objects defined outside any function exist throughout the program's execution. Such objects are created when the program starts and are not destroyed until the program ends. The lifetime of a local variable depends on how it is defined.

Automatic Objects

The objects that correspond to ordinary local variables are created when the function's control path passes through the variable's definition. They are destroyed when control passes through the end of the block in which the variable is defined. Objects that exist only while a block is executing are known as **automatic objects**. After execution exits a block, the values of the automatic objects created in that block are undefined.

Parameters are automatic objects. Storage for the parameters is allocated when the function begins. Parameters are defined in the scope of the function body. Hence they are destroyed when the function terminates.

Automatic objects corresponding to the function's parameters are initialized by the arguments passed to the function. Automatic objects corresponding to local variables are initialized if their definition contains an initializer. Otherwise, they are default initialized (§ 2.2.1, p. 43), which means that uninitialized local variables of built-in type have undefined values.

Local `static` Objects

It can be useful to have a local variable whose lifetime continues across calls to the function. We obtain such objects by defining a local variable as `static`. Each **local `static` object** is initialized before the *first* time execution passes through the

object's definition. Local `static`s are not destroyed when a function ends; they are destroyed when the program terminates.

As a trivial example, here is a function that counts how many times it is called:

```
size_t count_calls()
{
    static size_t ctr = 0;   // value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

This program will print the numbers from 1 through 10 inclusive.

Before control flows through the definition of `ctr` for the first time, `ctr` is created and given an initial value of `0`. Each call increments `ctr` and returns its new value. Whenever `count_calls` is executed, the variable `ctr` already exists and has whatever value was in that variable the last time the function exited. Thus, on the second invocation, the value of `ctr` is `1`, on the third it is `2`, and so on.

If a local `static` has no explicit initializer, it is value initialized (§ 3.3.1, p. 98), meaning that local `static`s of built-in type are initialized to zero.

Exercises Section 6.1.1

Exercise 6.6: Explain the differences between a parameter, a local variable, and a local `static` variable. Give an example of a function in which each might be useful.

Exercise 6.7: Write a function that returns 0 when it is first called and then generates numbers in sequence each time it is called again.

6.1.2 Function Declarations

Like any other name, the name of a function must be declared before we can use it. As with variables (§ 2.2.2, p. 45), a function may be defined only once but may be declared multiple times. With one exception that we'll cover in § 15.3 (p. 603), we can declare a function that is not defined so long as we never use that function.

A function declaration is just like a function definition except that a declaration has no function body. In a declaration, a semicolon replaces the function body.

Because a function declaration has no body, there is no need for parameter names. Hence, parameter names are often omitted in a declaration. Although parameter names are not required, they can be used to help users of the function understand what the function does:

```
// parameter names chosen to indicate that the iterators denote a range of values to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
```

These three elements—the return type, function name, and parameter types—describe the function's interface. They specify all the information we need to call the function. Function declarations are also known as the **function prototype**.

Function Declarations Go in Header Files

Recall that variables are declared in header files (§ 2.6.3, p. 76) and defined in source files. For the same reasons, functions should be declared in header files and defined in source files.

It may be tempting—and would be legal—to put a function declaration directly in each source file that uses the function. However, doing so is tedious and error-prone. When we use header files for our function declarations, we can ensure that all the declarations for a given function agree. Moreover, if the interface to the function changes, only one declaration has to be changed.

The source file that defines a function should include the header that contains that function's declaration. That way the compiler will verify that the definition and declaration are consistent.

The header that *declares* a function should be included in the source file that *defines* that function.

EXERCISES SECTION 6.1.2

Exercise 6.8: Write a header file named `Chapter6.h` that contains declarations for the functions you wrote for the exercises in § 6.1 (p. 205).

6.1.3 Separate Compilation

As our programs get more complicated, we'll want to store the various parts of the program in separate files. For example, we might store the functions we wrote for the exercises in § 6.1 (p. 205) in one file and store code that uses these functions in other source files. To allow programs to be written in logical parts, C++ supports what is commonly known as ***separate compilation***. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

Compiling and Linking Multiple Source Files

As an example, assume that the definition of our `fact` function is in a file named `fact.cc` and its declaration is in a header file named `Chapter6.h`. Our `fact.cc` file, like any file that uses these functions, will include the `Chapter6.h` header. We'll store a `main` function that calls `fact` in a second file named `factMain.cc`.

To produce an ***executable file***, we must tell the compiler where to find all of the code we use. We might compile these files as follows:

```
$ CC factMain.cc fact.cc   # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

Here `CC` is the name of our compiler, `$` is our system prompt, and `#` begins a command-line comment. We can now run the executable file, which will run our `main` function.

If we have changed only one of our source files, we'd like to recompile only the file that actually changed. Most compilers provide a way to separately compile each file. This process usually yields a file with the `.obj` (Windows) or `.o` (UNIX) file extension, indicating that the file contains ***object code***.

The compiler lets us ***link*** object files together to form an executable. On the system we use, we would separately compile our program as follows:

```
$ CC -c factMain.cc     # generates factMain.o
$ CC -c fact.cc         # generates fact.o
$ CC factMain.o fact.o  # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main # generates main or main.exe
```

You'll need to check with your compiler's user's guide to understand how to compile and execute programs made up of multiple source files.

Exercises Section 6.1.3

Exercise 6.9: Write your own versions of the `fact.cc` and `factMain.cc` files. These files should include your `Chapter6.h` from the exercises in the previous section. Use these files to understand how your compiler supports separate compilation.

6.2 Argument Passing

As we've seen, each time we call a function, its parameters are created and initialized by the arguments passed in the call.

Parameter initialization works the same way as variable initialization.

As with any other variable, the type of a parameter determines the interaction between the parameter and its argument. If the parameter is a reference (§ 2.3.1, p. 50), then the parameter is bound to its argument. Otherwise, the argument's value is copied.

When a parameter is a reference, we say that its corresponding argument is **"passed by reference"** or that the function is **"called by reference."** As with any other reference, a reference parameter is an alias for the object to which it is bound; that is, the parameter is an alias for its corresponding argument.

When the argument value is copied, the parameter and argument are independent objects. We say such arguments are **"passed by value"** or alternatively that the function is **"called by value."**

6.2.1 Passing Arguments by Value

When we initialize a nonreference type variable, the value of the initializer is copied. Changes made to the variable have no effect on the initializer:

```
int n = 0;              // ordinary variable of type int
int i = n;              // i is a copy of the value in n
i = 42;                 // value in i is changed; n is unchanged
```

Passing an argument by value works exactly the same way; nothing the function does to the parameter can affect the argument. For example, inside `fact` (§ 6.1, p. 202) the parameter `val` is decremented:

```
ret *= val--;  // decrements the value of val
```

Although `fact` changes the value of `val`, that change has no effect on the argument passed to `fact`. Calling `fact(i)` does not change the value of `i`.

Pointer Parameters

Pointers (§ 2.3.2, p. 52) behave like any other nonreference type. When we copy a pointer, the value of the pointer is copied. After the copy, the two pointers are distinct. However, a pointer also gives us indirect access to the object to which that pointer points. We can change the value of that object by assigning through the pointer (§ 2.3.2, p. 55):

```
int n = 0, i = 42;
int *p = &n, *q = &i; // p points to n; q points to i
*p = 42;              // value in n is changed; p is unchanged
p = q;                // p now points to i; values in i and n are unchanged
```

The same behavior applies to pointer parameters:

```
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0;  // changes the value of the object to which ip points
    ip = 0;   // changes only the local copy of ip; the argument is unchanged
}
```

After a call to `reset`, the object to which the argument points will be `0`, but the pointer argument itself is unchanged:

```
int i = 42;
reset(&i);                      // changes i but not the address of i
cout << "i = "  << i << endl;   // prints i = 0
```

Programmers accustomed to programming in C often use pointer parameters to access objects outside a function. In C++, programmers generally use reference parameters instead.

EXERCISES SECTION 6.2.1

Exercise 6.10: Using pointers, write a function to swap the values of two `int`s. Test the function by calling it and printing the swapped values.

6.2.2 Passing Arguments by Reference

Recall that operations on a reference are actually operations on the object to which the reference refers (§ 2.3.1, p. 50):

```
int n = 0, i = 42;
int &r = n;              //  r is bound to n (i.e., r is another name for n)
r = 42;                  //  n is now 42
r = i;                   //  n now has the same value as i
i = r;                   //  i has the same value as n
```

Reference parameters exploit this behavior. They are often used to allow a function to change the value of one or more of its arguments.

As one example, we can rewrite our `reset` program from the previous section to take a reference instead of a pointer:

```
//  function that takes a reference to an int and sets the given object to zero
void reset(int &i)   //  i is just another name for the object passed to reset
{
    i = 0;   //  changes the value of the object to which i refers
}
```

As with any other reference, a reference parameter is bound directly to the object from which it is initialized. When we call this version of `reset`, `i` will be bound to whatever `int` object we pass. As with any reference, changes made to `i` are made to the object to which `i` refers. In this case, that object is the argument to `reset`.

When we call this version of `reset`, we pass an object directly; there is no need to pass its address:

```
int j = 42;
reset(j);  //  j is passed by reference; the value in j is changed
cout << "j = " << j  << endl;  //  prints j = 0
```

In this call, the parameter `i` is just another name for `j`. Any use of `i` inside `reset` is a use of `j`.

Using References to Avoid Copies

It can be inefficient to copy objects of large class types or large containers. Moreover, some class types (including the IO types) cannot be copied. Functions must use reference parameters to operate on objects of a type that cannot be copied.

As an example, we'll write a function to compare the length of two `string`s. Because `string`s can be long, we'd like to avoid copying them, so we'll make our parameters references. Because comparing two `string`s does not involve changing the `string`s, we'll make the parameters references to `const` (§ 2.4.1, p. 61):

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

As we'll see in § 6.2.3 (p. 213), functions should use references to `const` for reference parameters they do not need to change.

Reference parameters that are not changed inside a function should be references to `const`.

Using Reference Parameters to Return Additional Information

A function can return only a single value. However, sometimes a function has more than one value to return. Reference parameters let us effectively return multiple results. As an example, we'll define a function named `find_char` that will return the position of the first occurrence of a given character in a `string`. We'd also like the function to return a count of how many times that character occurs.

How can we define a function that returns a position and an occurrence count? We could define a new type that contains the position and the count. An easier solution is to pass an additional reference argument to hold the occurrence count:

```
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size();    // position of the first occurrence, if any
    occurs = 0;             // set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;    // remember the first occurrence of c
            ++occurs;       // increment the occurrence count
        }
    }
    return ret;             // count is returned implicitly in occurs
}
```

When we call `find_char`, we have to pass three arguments: a `string` in which to look, the character to look for, and a `size_type` (§ 3.2.2, p. 88) object to hold the occurrence count. Assuming `s` is a `string`, and `ctr` is a `size_type` object, we can call `find_char` as follows:

```
auto index = find_char(s, 'o', ctr);
```

After the call, the value of `ctr` will be the number of times `o` occurs, and `index` will refer to the first occurrence if there is one. Otherwise, `index` will be equal to `s.size()` and `ctr` will be zero.

EXERCISES SECTION 6.2.2

Exercise 6.11: Write and test your own version of `reset` that takes a reference.

Exercise 6.12: Rewrite the program from exercise 6.10 in § 6.2.1 (p. 210) to use references instead of pointers to swap the value of two `int`s. Which version do you think would be easier to use and why?

Exercise 6.13: Assuming `T` is the name of a type, explain the difference between a function declared as `void f(T)` and `void f(T&)`.

Exercise 6.14: Give an example of when a parameter should be a reference type. Give an example of when a parameter should not be a reference.

Exercise 6.15: Explain the rationale for the type of each of `find_char`'s parameters In particular, why is `s` a reference to `const` but `occurs` is a plain reference? Why are these parameters references, but the `char` parameter `c` is not? What would happen if we made `s` a plain reference? What if we made `occurs` a reference to `const`?

6.2.3 `const` Parameters and Arguments

When we use parameters that are `const`, it is important to remember the discussion of top-level `const` from § 2.4.3 (p. 63). As we saw in that section, a top-level `const` is one that applies to the object itself:

```
const int ci = 42;      // we cannot change ci; const is top-level
int i = ci;             // ok: when we copy ci, its top-level const is ignored
int * const p = &i;     // const is top-level; we can't assign to p
*p = 0;                 // ok: changes through p are allowed; i is now 0
```

Just as in any other initialization, when we copy an argument to initialize a parameter, top-level `const`s are ignored. As a result, top-level `const` on parameters are ignored. We can pass either a `const` or a nonconst object to a parameter that has a top-level `const`:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
```

We can call `fcn` passing it either a `const int` or a plain `int`. The fact that top-level `const`s are ignored on a parameter has one possibly surprising implication:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ }  // error: redefines fcn(int)
```

In C++, we can define several different functions that have the same name. However, we can do so only if their parameter lists are sufficiently different. Because top-level consts are ignored, we can pass exactly the same types to either version of fcn. The second version of fcn is an error. Despite appearances, its parameter list doesn't differ from the list in the first version of fcn.

Pointer or Reference Parameters and const

Because parameters are initialized in the same way that variables are initialized, it can be helpful to remember the general initialization rules. We can initialize an object with a low-level const from a nonconst object but not vice versa, and a plain reference must be initialized from an object of the same type.

```
int i = 42;
const int *cp = &i; // ok: but cp can't change i (§ 2.4.2 (p. 62))
const int &r = i;   // ok: but r can't change i (§ 2.4.1 (p. 61))
const int &r2 = 42; // ok: (§ 2.4.1 (p. 61))
int *p = cp;  // error: types of p and cp don't match (§ 2.4.2 (p. 62))
int &r3 = r;  // error: types of r3 and r don't match (§ 2.4.1 (p. 61))
int &r4 = 42; // error: can't initialize a plain reference from a literal (§ 2.3.1 (p. 50))
```

Exactly the same initialization rules apply to parameter passing:

```
int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i);    // calls the version of reset that has an int* parameter
reset(&ci);   // error: can't initialize an int* from a pointer to a const int object
reset(i);     // calls the version of reset that has an int& parameter
reset(ci);    // error: can't bind a plain reference to the const object ci
reset(42);    // error: can't bind a plain reference to a literal
reset(ctr);   // error: types don't match; ctr has an unsigned type
// ok: find_char's first parameter is a reference to const
find_char("Hello World!", 'o', ctr);
```

We can call the reference version of reset (§ 6.2.2, p. 210) only on int objects. We cannot pass a literal, an expression that evaluates to an int, an object that requires conversion, or a const int object. Similarly, we may pass only an int* to the pointer version of reset (§ 6.2.1, p. 209). On the other hand, we can pass a string literal as the first argument to find_char (§ 6.2.2, p. 211). That function's reference parameter is a reference to const, and we can initialize references to const from literals.

Use Reference to const When Possible

It is a somewhat common mistake to define parameters that a function does not change as (plain) references. Doing so gives the function's caller the misleading impression that the function might change its argument's value. Moreover, using a

reference instead of a reference to `const` unduly limits the type of arguments that can be used with the function. As we've just seen, we cannot pass a `const` object, or a literal, or an object that requires conversion to a plain reference parameter.

The effect of this mistake can be surprisingly pervasive. As an example, consider our `find_char` function from § 6.2.2 (p. 211). That function (correctly) made its `string` parameter a reference to `const`. Had we defined that parameter as a plain `string&`:

```
// bad design: the first parameter should be a const string&
string::size_type find_char(string &s, char c,
                            string::size_type &occurs);
```

we could call `find_char` only on a `string` object. A call such as

```
find_char("Hello World", 'o', ctr);
```

would fail at compile time.

More subtly, we could not use this version of `find_char` from other functions that (correctly) define their parameters as references to `const`. For example, we might want to use `find_char` inside a function that determines whether a `string` represents a sentence:

```
bool is_sentence(const string &s)
{
    // if there's a single period at the end of s, then s is a sentence
    string::size_type ctr = 0;
    return find_char(s, '.', ctr) == s.size() - 1 && ctr == 1;
}
```

If `find_char` took a plain `string&`, then this call to `find_char` would be a compile-time error. The problem is that `s` is a reference to a `const string`, but `find_char` was (incorrectly) defined to take a plain reference.

It might be tempting to try to fix this problem by changing the type of the parameter in `is_sentence`. But that fix only propagates the error—callers of `is_sentence` could pass only nonconst `string`s.

The right way to fix this problem is to fix the parameter in `find_char`. If it's not possible to change `find_char`, then define a local `string` copy of `s` inside `is_sentence` and pass that `string` to `find_char`.

6.2.4 Array Parameters

Arrays have two special properties that affect how we define and use functions that operate on arrays: We cannot copy an array (§ 3.5.1, p. 114), and when we use an array it is (usually) converted to a pointer (§ 3.5.3, p. 117). Because we cannot copy an array, we cannot pass an array by value. Because arrays are converted to pointers, when we pass an array to a function, we are actually passing a pointer to the array's first element.

Even though we cannot pass an array by value, we can write a parameter that looks like an array:

EXERCISES SECTION 6.2.3

Exercise 6.16: The following function, although legal, is less useful than it might be. Identify and correct the limitation on this function:

```
bool is_empty(string& s) { return s.empty(); }
```

Exercise 6.17: Write a function to determine whether a `string` contains any capital letters. Write a function to change a `string` to all lowercase. Do the parameters you used in these functions have the same type? If so, why? If not, why not?

Exercise 6.18: Write declarations for each of the following functions. When you write these declarations, use the name of the function to indicate what the function does.

(a) A function named `compare` that returns a `bool` and has two parameters that are references to a class named `matrix`.

(b) A function named `change_val` that returns a `vector<int>` iterator and takes two parameters: One is an `int` and the other is an iterator for a `vector<int>`.

Exercise 6.19: Given the following declarations, determine which calls are legal and which are illegal. For those that are illegal, explain why.

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1); (b) count("abcda", 'a');
(c) calc(66);         (d) sum(vec.begin(), vec.end(), 3.8);
```

Exercise 6.20: When should reference parameters be references to `const`? What happens if we make a parameter a plain reference when it could be a reference to `const`?

```
// despite appearances, these three declarations of print are equivalent
// each function has a single parameter of type const int*
void print(const int*);
void print(const int[]);    // shows the intent that the function takes an array
void print(const int[10]); // dimension for documentation purposes (at best)
```

Regardless of appearances, these declarations are equivalent: Each declares a function with a single parameter of type `const int*`. When the compiler checks a call to `print`, it checks only that the argument has type `const int*`:

```
int i = 0, j[2] = {0, 1};
print(&i); // ok: &i is int*
print(j);  // ok: j is converted to an int* that points to j[0]
```

If we pass an array to `print`, that argument is automatically converted to a pointer to the first element in the array; the size of the array is irrelevant.

As with any code that uses arrays, functions that take array parameters must ensure that all uses of the array stay within the array bounds.

Because arrays are passed as pointers, functions ordinarily don't know the size of the array they are given. They must rely on additional information provided by the caller. There are three common techniques used to manage pointer parameters.

Using a Marker to Specify the Extent of an Array

The first approach to managing array arguments requires the array itself to contain an end marker. C-style character strings (§ 3.5.4, p. 122) are an example of this approach. C-style strings are stored in character arrays in which the last character of the string is followed by a null character. Functions that deal with C-style strings stop processing the array when they see a null character:

```
void print(const char *cp)
{
    if (cp)           //  if cp is not a null pointer
        while (*cp)   //  so long as the character it points to is not a null character
            cout << *cp++;   //  print the character and advance the pointer
}
```

This convention works well for data where there is an obvious end-marker value (like the null character) that does not appear in ordinary data. It works less well with data, such as `int`s, where every value in the range is a legitimate value.

Using the Standard Library Conventions

A second technique used to manage array arguments is to pass pointers to the first and one past the last element in the array. This approach is inspired by techniques used in the standard library. We'll learn more about this style of programming in Part II. Using this approach, we'll print the elements in an array as follows:

```
void print(const int *beg, const int *end)
{
    //  print every element starting at beg up to but not including end
    while (beg != end)
        cout << *beg++ << endl; //  print the current element
                                //  and advance the pointer
}
```

The `while` uses the dereference and postfix increment operators (§ 4.5, p. 148) to print the current element and advance `beg` one element at a time through the array. The loop stops when `beg` is equal to `end`.

To call this function, we pass two pointers—one to the first element we want to print and one just past the last element:

```
int j[2] = {0, 1};
//  j is converted to a pointer to the first element in j
//  the second argument is a pointer to one past the end of j
print(begin(j), end(j));   //  begin and end functions, see § 3.5.3 (p. 118)
```

This function is safe, as long as the caller correctly calculates the pointers. Here we let the library `begin` and `end` functions (§ 3.5.3, p. 118) provide those pointers.

Explicitly Passing a Size Parameter

A third approach for array arguments, which is common in C programs and older C++ programs, is to define a second parameter that indicates the size of the array. Using this approach, we'll rewrite `print` as follows:

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void print(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
```

This version uses the `size` parameter to determine how many elements there are to print. When we call `print`, we must pass this additional parameter:

```
int j[] = { 0, 1 };  // int array of size 2
print(j, end(j) - begin(j));
```

The function executes safely as long as the size passed is no greater than the actual size of the array.

Array Parameters and `const`

Note that all three versions of our `print` function defined their array parameters as pointers to `const`. The discussion in § 6.2.3 (p. 213) applies equally to pointers as to references. When a function does not need write access to the array elements, the array parameter should be a pointer to `const` (§ 2.4.2, p. 62). A parameter should be a plain pointer to a nonconst type only if the function needs to change element values.

Array Reference Parameters

Just as we can define a variable that is a reference to an array (§ 3.5.1, p. 114), we can define a parameter that is a reference to an array. As usual, the reference parameter is bound to the corresponding argument, which in this case is an array:

```
// ok: parameter is a reference to an array; the dimension is part of the type
void print(int (&arr)[10])
{
    for (auto elem : arr)
        cout << elem << endl;
}
```

The parentheses around `&arr` are necessary (§ 3.5.1, p. 114):

```
f(int &arr[10])   // error: declares arr as an array of references
f(int (&arr)[10]) // ok: arr is a reference to an array of ten ints
```

Because the size of an array is part of its type, it is safe to rely on the dimension in the body of the function. However, the fact that the size is part of the type limits the usefulness of this version of `print`. We may call this function only for an array of exactly ten `int`s:

```
int i = 0, j[2] = {0, 1};
int k[10] = {0,1,2,3,4,5,6,7,8,9};
print(&i);    // error: argument is not an array of ten int
print(j);     // error: argument is not an array of ten int
print(k);     // ok: argument is an array of ten int
```

We'll see in § 16.1.1 (p. 654) how we might write this function in a way that would allow us to pass a reference parameter to an array of any size.

Passing a Multidimensional Array

Recall that there are no multidimensional arrays in C++ (§ 3.6, p. 125). Instead, what appears to be a multidimensional array is an array of arrays.

As with any array, a multidimensional array is passed as a pointer to its first element (§ 3.6, p. 128). Because we are dealing with an array of arrays, that element is an array, so the pointer is a pointer to an array. The size of the second (and any subsequent) dimension is part of the element type and must be specified:

```
// matrix points to the first element in an array whose elements are arrays of ten int
void print(int (*matrix)[10], int rowSize) { /* ... */ }
```

declares `matrix` as a pointer to an array of ten `int`s.

Again, the parentheses around `*matrix` are necessary:

```
int *matrix[10];    // array of ten pointers
int (*matrix)[10]; // pointer to an array of ten int
```

We can also define our function using array syntax. As usual, the compiler ignores the first dimension, so it is best not to include it:

```
// equivalent definition
void print(int matrix[][10], int rowSize) { /* ... */ }
```

declares `matrix` to be what looks like a two-dimensional array. In fact, the parameter is a pointer to an array of ten `int`s.

6.2.5 `main`: Handling Command-Line Options

It turns out that `main` is a good example of how C++ programs pass arrays to functions. Up to now, we have defined `main` with an empty parameter list:

```
int main() { ... }
```

However, we sometimes need to pass arguments to `main`. The most common use of arguments to `main` is to let the user specify a set of options to guide the operation of the program. For example, assuming our `main` program is in an executable file named `prog`, we might pass options to the program as follows:

EXERCISES SECTION 6.2.4

Exercise 6.21: Write a function that takes an `int` and a pointer to an `int` and returns the larger of the `int` value or the value to which the pointer points. What type should you use for the pointer?

Exercise 6.22: Write a function to swap two `int` pointers.

Exercise 6.23: Write your own versions of each of the `print` functions presented in this section. Call each of these functions to print `i` and `j` defined as follows:

```
int i = 0, j[2] = {0, 1};
```

Exercise 6.24: Explain the behavior of the following function. If there are problems in the code, explain what they are and how you might fix them.

```
void print(const int ia[10])
{
    for (size_t i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

```
prog -d -o ofile data0
```

Such command-line options are passed to `main` in two (optional) parameters:

```
int main(int argc, char *argv[]) { ... }
```

The second parameter, `argv`, is an array of pointers to C-style character strings. The first parameter, `argc`, passes the number of strings in that array. Because the second parameter is an array, we might alternatively define `main` as

```
int main(int argc, char **argv) { ... }
```

indicating that `argv` points to a `char*`.

When arguments are passed to `main`, the first element in `argv` points either to the name of the program or to the empty string. Subsequent elements pass the arguments provided on the command line. The element just past the last pointer is guaranteed to be 0.

Given the previous command line, `argc` would be 5, and `argv` would hold the following C-style character strings:

```
argv[0] = "prog";   // or argv[0] might point to an empty string
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```

When you use the arguments in `argv`, remember that the optional arguments begin in `argv[1]`; `argv[0]` contains the program's name, not user input.

Exercises Section 6.2.5

Exercise 6.25: Write a `main` function that takes two arguments. Concatenate the supplied arguments and print the resulting `string`.

Exercise 6.26: Write a program that accepts the options presented in this section. Print the values of the arguments passed to `main`.

6.2.6 Functions with Varying Parameters

Sometimes we do not know in advance how many arguments we need to pass to a function. For example, we might want to write a routine to print error messages generated from our program. We'd like to use a single function to print these error messages in order to handle them in a uniform way. However, different calls to our error-printing function might pass different arguments, corresponding to different kinds of error messages.

The new standard provides two primary ways to write a function that takes a varying number of arguments: If all the arguments have the same type, we can pass a library type named `initializer_list`. If the argument types vary, we can write a special kind of function, known as a variadic template, which we'll cover in § 16.4 (p. 699).

C++ also has a special parameter type, ellipsis, that can be used to pass a varying number of arguments. We'll look briefly at ellipsis parameters in this section. However, it is worth noting that this facility ordinarily should be used only in programs that need to interface to C functions.

`initializer_list` Parameters

C++ 11

We can write a function that takes an unknown number of arguments of a single type by using an **`initializer_list`** parameter. An `initializer_list` is a library type that represents an array (§ 3.5, p. 113) of values of the specified type. This type is defined in the `initializer_list` header. The operations that `initializer_list` provides are listed in Table 6.1.

Table 6.1: Operations on `initializer_lists`

`initializer_list<T> lst;`	Default initialization; an empty list of elements of type `T`.
`initializer_list<T> lst{a,b,c...};`	`lst` has as many elements as there are initializers; elements are copies of the corresponding initializers. Elements in the list are `const`.
`lst2(lst)` `lst2 = lst`	Copying or assigning an `initializer_list` does not copy the elements in the list. After the copy, the original and the copy share the elements.
`lst.size()`	Number of elements in the list.
`lst.begin()` `lst.end()`	Returns a pointer to the first and one past the last element in `lst`.

Like a vector, initializer_list is a template type (§ 3.3, p. 96). When we define an initializer_list, we must specify the type of the elements that the list will contain:

```
initializer_list<string> ls; // initializer_list of strings
initializer_list<int> li;    // initializer_list of ints
```

Unlike vector, the elements in an initializer_list are always const values; there is no way to change the value of an element in an initializer_list.

We can write our function to produce error messages from a varying number of arguments as follows:

```
void error_msg(initializer_list<string> il)
{
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " " ;
    cout << endl;
}
```

The begin and end operations on initializer_list objects are analogous to the corresponding vector members (§ 3.4.1, p. 106). The begin() member gives us a pointer to the first element in the list, and end() is an off-the-end pointer one past the last element. Our function initializes beg to denote the first element and iterates through each element in the initializer_list. In the body of the loop we dereference beg in order to access the current element and print its value.

When we pass a sequence of values to an initializer_list parameter, we must enclose the sequence in curly braces:

```
// expected, actual are strings
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error_msg({"functionX", "okay"});
```

Here we're calling the same function, error_msg, passing three values in the first call and two values in the second.

A function with an initializer_list parameter can have other parameters as well. For example, our debugging system might have a class, named ErrCode, that represents various kinds of errors. We can revise our program to take an ErrCode in addition to an initializer_list as follows:

```
void error_msg(ErrCode e, initializer_list<string> il)
{
    cout << e.msg() << ": ";
    for (const auto &elem : il)
        cout << elem << " " ;
    cout << endl;
}
```

Because initializer_list has begin and end members, we can use a range for (§ 5.4.3, p. 187) to process the elements. This program, like our previous version, iterates an element at a time through the braced list of values passed to the il parameter.

To call this version, we need to revise our calls to pass an `ErrCode` argument:

```
if (expected != actual)
    error_msg(ErrCode(42), {"functionX", expected, actual});
else
    error_msg(ErrCode(0), {"functionX", "okay"});
```

Ellipsis Parameters

Ellipsis parameters are in C++ to allow programs to interface to C code that uses a C library facility named `varargs`. Generally an ellipsis parameter should not be used for other purposes. Your C compiler documentation will describe how to use `varargs`.

Ellipsis parameters should be used only for types that are common to both C and C++. In particular, objects of most class types are not copied properly when passed to an ellipsis parameter.

An ellipsis parameter may appear only as the last element in a parameter list and may take either of two forms:

```
void foo(parm_list, ...);
void foo(...);
```

The first form specifies the type(s) for some of `foo`'s parameters. Arguments that correspond to the specified parameters are type checked as usual. No type checking is done for the arguments that correspond to the ellipsis parameter. In this first form, the comma following the parameter declarations is optional.

EXERCISES SECTION 6.2.6

Exercise 6.27: Write a function that takes an `initializer_list<int>` and produces the sum of the elements in the list.

Exercise 6.28: In the second version of `error_msg` that has an `ErrCode` parameter, what is the type of `elem` in the `for` loop?

Exercise 6.29: When you use an `initializer_list` in a range `for` would you ever use a reference as the loop control variable? If so, why? If not, why not?

6.3 Return Types and the `return` Statement

A `return` statement terminates the function that is currently executing and returns control to the point from which the function was called. There are two forms of `return` statements:

```
return;
return expression;
```

6.3.1 Functions with No Return Value

A `return` with no value may be used only in a function that has a return type of `void`. Functions that return `void` are not required to contain a `return`. In a `void` function, an implicit `return` takes place after the function's last statement.

Typically, `void` functions use a `return` to exit the function at an intermediate point. This use of `return` is analogous to the use of a `break` statement (§ 5.5.1, p. 190) to exit a loop. For example, we can write a `swap` function that does no work if the values are identical:

```
void swap(int &v1, int &v2)
{
    // if the values are already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // if we're here, there's work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
```

This function first checks if the values are equal and, if so, exits the function. If the values are unequal, the function swaps them. An implicit return occurs after the last assignment statement.

A function with a `void` return type may use the second form of the `return` statement only to return the result of calling another function that returns `void`. Returning any other expression from a `void` function is a compile-time error.

6.3.2 Functions That Return a Value

The second form of the `return` statement provides the function's result. Every return in a function with a return type other than `void` must return a value. The value returned must have the same type as the function return type, or it must have a type that can be implicitly converted (§ 4.11, p. 159) to that type.

Although C++ cannot guarantee the correctness of a result, it can guarantee that every `return` includes a result of the appropriate type. Although it cannot do so in all cases, the compiler attempts to ensure that functions that return a value are exited only through a valid `return` statement. For example:

```
// incorrect return values, this code will not compile
bool str_subrange(const string &str1, const string &str2)
{
    // same sizes: return normal equality test
    if (str1.size() == str2.size())
        return str1 == str2;    // ok: == returns bool
    // find the size of the smaller string; conditional operator, see § 4.7 (p. 151)
    auto size = (str1.size() < str2.size())
                ? str1.size() : str2.size();
```

```
        // look at each element up to the size of the smaller string
        for (decltype(size) i = 0; i != size; ++i) {
            if (str1[i] != str2[i])
                return; // error #1: no return value; compiler should detect this error
        }
        // error #2: control might flow off the end of the function without a return
        // the compiler might not detect this error
}
```

The `return` from within the `for` loop is an error because it fails to return a value. The compiler should detect this error.

The second error occurs because the function fails to provide a `return` after the loop. If we call this function with one `string` that is a subset of the other, execution would fall out of the `for`. There should be a return to handle this case. The compiler may or may not detect this error. If it does not detect the error, what happens at run time is undefined.

Failing to provide a `return` after a loop that contains a `return` is an error. However, many compilers will not detect such errors.

How Values Are Returned

Values are returned in exactly the same way as variables and parameters are initialized: The return value is used to initialize a temporary at the call site, and that temporary is the result of the function call.

It is important to keep in mind the initialization rules in functions that return local variables. As an example, we might write a function that, given a counter, a word, and an ending, gives us back the plural version of the word if the counter is greater than 1:

```
// return the plural version of word if ctr is greater than 1
string make_plural(size_t ctr, const string &word,
                                const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

The return type of this function is `string`, which means the return value is copied to the call site. This function returns a copy of `word`, or it returns an unnamed temporary `string` that results from adding `word` and `ending`.

As with any other reference, when a function returns a reference, that reference is just another name for the object to which it refers. As an example, consider a function that returns a reference to the shorter of its two `string` parameters:

```
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

The parameters and return type are references to `const string`. The `strings` are not copied when the function is called or when the result is returned.

Never Return a Reference or Pointer to a Local Object

When a function completes, its storage is freed (§ 6.1.1, p. 204). After a function terminates, references to local objects refer to memory that is no longer valid:

```
// disaster: this function returns a reference to a local object
const string &manip()
{
    string ret;
    // transform ret in some way
    if (!ret.empty())
        return ret;      // WRONG: returning a reference to a local object!
    else
        return "Empty"; // WRONG: "Empty" is a local temporary string
}
```

Both of these `return` statements return an undefined value—what happens if we try to use the value returned from `manip` is undefined. In the first `return`, it should be obvious that the function returns a reference to a local object. In the second case, the string literal is converted to a local temporary `string` object. That object, like the `string` named `ret`, is local to `manip`. The storage in which the temporary resides is freed when the function ends. Both `returns` refer to memory that is no longer available.

One good way to ensure that the return is safe is to ask: To what *preexisting* object is the reference referring?

For the same reasons that it is wrong to return a reference to a local object, it is also wrong to return a pointer to a local object. Once the function completes, the local objects are freed. The pointer would point to a nonexistent object.

Functions That Return Class Types and the Call Operator

Like any operator the call operator has associativity and precedence (§ 4.1.2, p. 136). The call operator has the same precedence as the dot and arrow operators (§ 4.6, p. 150). Like those operators, the call operator is left associative. As a result, if a function returns a pointer, reference or object of class type, we can use the result of a call to call a member of the resulting object.

For example, we can determine the size of the shorter `string` as follows:

```
// call the size member of the string returned by shorterString
auto sz = shorterString(s1, s2).size();
```

Because these operators are left associative, the result of `shorterString` is the left-hand operand of the dot operator. That operator fetches the `size` member of that `string`. That member is the left-hand operand of the second call operator.

Reference Returns Are Lvalues

Whether a function call is an lvalue (§ 4.1.1, p. 135) depends on the return type of the function. Calls to functions that return references are lvalues; other return types yield rvalues. A call to a function that returns a reference can be used in the same ways as any other lvalue. In particular, we can assign to the result of a function that returns a reference to nonconst:

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}
int main()
{
    string s("a value");
    cout << s << endl;   // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A
    cout << s << endl;   // prints A value
    return 0;
}
```

It may be surprising to see a function call on the left-hand side of an assignment. However, nothing special is involved. The return value is a reference, so the call is an lvalue. Like any other lvalue, it may appear as the left-hand operand of the assignment operator.

If the return type is a reference to const, then (as usual) we may not assign to the result of the call:

```
shorterString("hi", "bye") = "X"; // error: return value is const
```

List Initializing the Return Value

C++ 11

Under the new standard, functions can return a braced list of values. As in any other return, the list is used to initialize the temporary that represents the function's return. If the list is empty, that temporary is value initialized (§ 3.3.1, p. 98). Otherwise, the value of the return depends on the function's return type.

As an example, recall the error_msg function from § 6.2.6 (p. 220). That function took a varying number of string arguments and printed an error message composed from the given strings. Rather than calling error_msg, in this function we'll return a vector that holds the error-message strings:

```
vector<string> process()
{
    // ...
    // expected and actual are strings
    if (expected.empty())
        return {};  // return an empty vector
    else if (expected == actual)
        return {"functionX", "okay"}; // return list-initialized vector
    else
        return {"functionX", expected, actual};
}
```

In the first return statement, we return an empty list. In this case, the vector that process returns will be empty. Otherwise, we return a vector initialized with two or three elements depending on whether expected and actual are equal.

In a function that returns a built-in type, a braced list may contain at most one value, and that value must not require a narrowing conversion (§ 2.2.1, p. 43). If the function returns a class type, then the class itself defines how the intiailizers are used (§ 3.3.1, p. 99).

Return from main

There is one exception to the rule that a function with a return type other than void must return a value: The main function is allowed to terminate without a return. If control reaches the end of main and there is no return, then the compiler implicitly inserts a return of 0.

As we saw in § 1.1 (p. 2), the value returned from main is treated as a status indicator. A zero return indicates success; most other values indicate failure. A nonzero value has a machine-dependent meaning. To make return values machine independent, the cstdlib header defines two preprocessor variables (§ 2.3.2, p. 54) that we can use to indicate success or failure:

```
int main()
{
    if (some_failure)
        return EXIT_FAILURE;  // defined in cstdlib
    else
        return EXIT_SUCCESS;  // defined in cstdlib
}
```

Because these are preprocessor variables, we must not precede them with std::, nor may we mention them in using declarations.

Recursion

A function that calls itself, either directly or indirectly, is a ***recursive function***. As an example, we can rewrite our factorial function to use recursion:

```
// calculate val!, which is 1 * 2 * 3 ... * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

In this implementation, we recursively call factorial to compute the factorial of the numbers counting down from the original value in val. Once we have reduced val to 1, we stop the recursion by returning 1.

There must always be a path through a recursive function that does not involve a recursive call; otherwise, the function will recurse "forever," meaning that the function will continue to call itself until the program stack is exhausted. Such

functions are sometimes described as containing a **recursion loop**. In the case of `factorial`, the stopping condition occurs when `val` is `1`.

The following table traces the execution of `factorial` when passed the value 5.

Trace of `factorial(5)`

Call	Returns	Value
`factorial(5)`	`factorial(4) * 5`	`120`
`factorial(4)`	`factorial(3) * 4`	`24`
`factorial(3)`	`factorial(2) * 3`	`6`
`factorial(2)`	`factorial(1) * 2`	`2`
`factorial(1)`	`1`	`1`

The `main` function may *not* call itself.

EXERCISES SECTION 6.3.2

Exercise 6.30: Compile the version of `str_subrange` as presented on page 223 to see what your compiler does with the indicated errors.

Exercise 6.31: When is it valid to return a reference? A reference to `const`?

Exercise 6.32: Indicate whether the following function is legal. If so, explain what it does; if not, correct any errors and then explain it.

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = i;
}
```

Exercise 6.33: Write a recursive function to print the contents of a `vector`.

Exercise 6.34: What would happen if the stopping condition in `factorial` were

```
if (val != 0)
```

Exercise 6.35: In the call to `factorial`, why did we pass `val - 1` rather than `val--`?

6.3.3 Returning a Pointer to an Array

Because we cannot copy an array, a function cannot return an array. However, a function can return a pointer or a reference to an array (§ 3.5.1, p. 114). Unfortunately, the syntax used to define functions that return pointers or references to

arrays can be intimidating. Fortunately, there are ways to simplify such declarations. The most straightforward way is to use a type alias (§ 2.5.1, p. 67):

```
typedef int arrT[10]; // arrT is a synonym for the type array of ten ints
using arrT = int[10]; // equivalent declaration of arrT; see § 2.5.1 (p. 68)
arrT* func(int i);    // func returns a pointer to an array of ten ints
```

Here `arrT` is a synonym for an array of ten `int`s. Because we cannot return an array, we define the return type as a pointer to this type. Thus, `func` is a function that takes a single `int` argument and returns a pointer to an array of ten `int`s.

Declaring a Function That Returns a Pointer to an Array

To declare `func` without using a type alias, we must remember that the dimension of an array follows the name being defined:

```
int arr[10];           // arr is an array of ten ints
int *p1[10];           // p1 is an array of ten pointers
int (*p2)[10] = &arr; // p2 points to an array of ten ints
```

As with these declarations, if we want to define a function that returns a pointer to an array, the dimension must follow the function's name. However, a function includes a parameter list, which also follows the name. The parameter list precedes the dimension. Hence, the form of a function that returns a pointer to an array is:

Type `(*`*function*`(`*parameter_list*`))[`*dimension*`]`

As in any other array declaration, *Type* is the type of the elements and *dimension* is the size of the array. The parentheses around `(*`*function*`(`*parameter_list*`))` are necessary for the same reason that they were required when we defined `p2`. Without them, we would be defining a function that returns an array of pointers.

As a concrete example, the following declares `func` without using a type alias:

```
int (*func(int i))[10];
```

To understand this declaration, it can be helpful to think about it as follows:

- `func(int)` says that we can call `func` with an `int` argument.
- `(*func(int))` says we can dereference the result of that call.
- `(*func(int))[10]` says that dereferencing the result of a call to `func` yields an array of size ten.
- `int (*func(int))[10]` says the element type in that array is `int`.

Using a Trailing Return Type

C++ 11

Under the new standard, another way to simplify the declaration of `func` is by using a **trailing return type**. Trailing returns can be defined for any function, but are most useful for functions with complicated return types, such as pointers (or references) to arrays. A trailing return type follows the parameter list and is preceded by `->`. To signal that the return follows the parameter list, we use `auto` where the return type ordinarily appears:

```
// fcn takes an int argument and returns a pointer to an array of ten ints
auto func(int i) -> int(*)[10];
```

Because the return type comes after the parameter list, it is easier to see that `func` returns a pointer and that that pointer points to an array of ten `int`s.

Using `decltype`

As another alternative, if we know the array(s) to which our function can return a pointer, we can use `decltype` to declare the return type. For example, the following function returns a pointer to one of two arrays, depending on the value of its parameter:

```
int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
// returns a pointer to an array of five int elements
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even; // returns a pointer to the array
}
```

The return type for `arrPtr` uses `decltype` to say that the function returns a pointer to whatever type `odd` has. That object is an array, so `arrPtr` returns a pointer to an array of five `int`s. The only tricky part is that we must remember that `decltype` does not automatically convert an array to its corresponding pointer type. The type returned by `decltype` is an array type, to which we must add a `*` to indicate that `arrPtr` returns a pointer.

Exercises Section 6.3.3

Exercise 6.36: Write the declaration for a function that returns a reference to an array of ten `string`s, without using either a trailing return, `decltype`, or a type alias.

Exercise 6.37: Write three additional declarations for the function in the previous exercise. One should use a type alias, one should use a trailing return, and the third should use `decltype`. Which form do you prefer and why?

Exercise 6.38: Revise the `arrPtr` function on to return a reference to the array.

6.4 Overloaded Functions

Functions that have the same name but different parameter lists and that appear in the same scope are **overloaded**. For example, in § 6.2.4 (p. 214) we defined several functions named `print`:

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

These functions perform the same general action but apply to different parameter types. When we call these functions, the compiler can deduce which function we want based on the argument type we pass:

```
int j[2] = {0,1};
print("Hello World");         // calls print(const char*)
print(j, end(j) - begin(j)); // calls print(const int*, size_t)
print(begin(j), end(j));      // calls print(const int*, const int*)
```

Function overloading eliminates the need to invent—and remember—names that exist only to help the compiler figure out which function to call.

The `main` function may *not* be overloaded.

Defining Overloaded Functions

Consider a database application with several functions to find a record based on name, phone number, account number, and so on. Function overloading lets us define a collection of functions, each named `lookup`, that differ in terms of how they do the search. We can call `lookup` passing a value of any of several types:

```
Record lookup(const Account&);  // find by Account
Record lookup(const Phone&);    // find by Phone
Record lookup(const Name&);     // find by Name

Account acct;
Phone phone;
Record r1 = lookup(acct);  // call version that takes an Account
Record r2 = lookup(phone); // call version that takes a Phone
```

Here, all three functions share the same name, yet they are three distinct functions. The compiler uses the argument type(s) to figure out which function to call.

Overloaded functions must differ in the number or the type(s) of their parameters. Each of the functions above takes a single parameter, but the parameters have different types.

It is an error for two functions to differ only in terms of their return types. If the parameter lists of two functions match but the return types differ, then the second declaration is an error:

```
Record lookup(const Account&);
bool lookup(const Account&);    // error: only the return type is different
```

Determining Whether Two Parameter Types Differ

Two parameter lists can be identical, even if they don't look the same:

```
// each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); // parameter names are ignored

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&); // Telno and Phone are the same type
```

In the first pair, the first declaration names its parameter. Parameter names are only a documentation aid. They do not change the parameter list.

In the second pair, it looks like the types are different, but `Telno` is not a new type; it is a synonym for `Phone`. A type alias (§ 2.5.1, p. 67) provides an alternative name for an existing type; it does not create a new type. Therefore, two parameters that differ only in that one uses an alias and the other uses the type to which the alias corresponds are not different.

Overloading and `const` Parameters

As we saw in § 6.2.3 (p. 212), top-level `const` (§ 2.4.3, p. 63) has no effect on the objects that can be passed to the function. A parameter that has a top-level `const` is indistinguishable from one without a top-level `const`:

```
Record lookup(Phone);
Record lookup(const Phone);   // redeclares Record lookup(Phone)
Record lookup(Phone*);
Record lookup(Phone* const); // redeclares Record lookup(Phone*)
```

In these declarations, the second declaration declares the same function as the first.

On the other hand, we can overload based on whether the parameter is a reference (or pointer) to the `const` or nonconst version of a given type; such `const`s are low-level:

```
// functions taking const and nonconst references or pointers have different parameters
// declarations for four independent, overloaded functions
Record lookup(Account&);        // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
Record lookup(Account*);        // new function, takes a pointer to Account
Record lookup(const Account*); // new function, takes a pointer to const
```

In these cases, the compiler can use the `const`ness of the argument to distinguish which function to call. Because there is no conversion (§ 4.11.2, p. 162) *from* `const`, we can pass a `const` object (or a pointer to `const`) only to the version with a `const` parameter. Because there is a conversion *to* `const`, we can call either function on a nonconst object or a pointer to nonconst. However, as we'll see in § 6.6.1 (p. 246), the compiler will prefer the nonconst versions when we pass a nonconst object or pointer to nonconst.

`const_cast` and Overloading

In § 4.11.3 (p. 163) we noted that `const_cast`s are most useful in the context of overloaded functions. As one example, recall our `shorterString` function from § 6.3.2 (p. 224):

```
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

ADVICE: WHEN NOT TO OVERLOAD A FUNCTION NAME

Although overloading lets us avoid having to invent (and remember) names for common operations, we should only overload operations that actually do similar things. There are some cases where providing different function names adds information that makes the program easier to understand. Consider a set of functions that move the cursor on a `Screen`.

```
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, string direction);
```

It might at first seem better to overload this set of functions under the name `move`:

```
Screen& move();
Screen& move(int, int);
Screen& move(int, int, string direction);
```

However, by overloading these functions, we've lost information that was inherent in the function names. Although cursor movement is a general operation shared by all these functions, the specific nature of that movement is unique to each of these functions. `moveHome`, for example, represents a special instance of cursor movement. Whether to overload these functions depends on which of these two calls is easier to understand:

```
// which is easier to understand?
myScreen.moveHome(); // we think this one!
myScreen.move();
```

This function takes and returns references to `const string`. We can call the function on a pair of nonconst `string` arguments, but we'll get a reference to a `const string` as the result. We might want to have a version of `shorterString` that, when given nonconst arguments, would yield a plain reference. We can write this version of our function using a `const_cast`:

```
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                            const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

This version calls the `const` version of `shorterString` by casting its arguments to references to `const`. That function returns a reference to a `const string`, which we know is bound to one of our original, nonconst arguments. Therefore, we know it is safe to cast that `string` back to a plain `string&` in the return.

Calling an Overloaded Function

Once we have defined a set of overloaded functions, we need to be able to call them with appropriate arguments. **Function matching** (also known as **overload resolution**) is the process by which a particular function call is associated with

a specific function from a set of overloaded functions. The compiler determines which function to call by comparing the arguments in the call with the parameters offered by each function in the overload set.

In many—probably most—cases, it is straightforward for a programmer to determine whether a particular call is legal and, if so, which function will be called. Often the functions in the overload set differ in terms of the number of arguments, or the types of the arguments are unrelated. In such cases, it is easy to determine which function is called. Determining which function is called when the overloaded functions have the same number of parameters and those parameters are related by conversions (§ 4.11, p. 159) can be less obvious. We'll look at how the compiler resolves calls involving conversions in § 6.6 (p. 242).

For now, what's important to realize is that for any given call to an overloaded function, there are three possible outcomes:

- The compiler finds exactly one function that is a **best match** for the actual arguments and generates code to call that function.
- There is no function with parameters that match the arguments in the call, in which case the compiler issues an error message that there was **no match**.
- There is more than one function that matches and none of the matches is clearly best. This case is also an error; it is an **ambiguous call**.

EXERCISES SECTION 6.4

Exercise 6.39: Explain the effect of the second declaration in each one of the following sets of declarations. Indicate which, if any, are illegal.

(a)
```
int calc(int, int);
int calc(const int, const int);
```
(b)
```
int get();
double get();
```
(c)
```
int *reset(int *);
double *reset(double *);
```

6.4.1 Overloading and Scope

Ordinarily, it is a bad idea to declare a function locally. However, to explain how scope interacts with overloading, we will violate this practice and use local function declarations.

Programmers new to C++ are often confused about the interaction between scope and overloading. However, overloading has no special properties with respect to scope: As usual, if we declare a name in an inner scope, that name *hides* uses of that name declared in an outer scope. Names do not overload across scopes:

```
string read();
void print(const string &);
void print(double);    // overloads the print function
void fooBar(int ival)
{
    bool read = false; // new scope: hides the outer declaration of read
    string s = read(); // error: read is a bool variable, not a function
    // bad practice: usually it's a bad idea to declare functions at local scope
    void print(int);   // new scope: hides previous instances of print
    print("Value: "); // error: print(const string &) is hidden
    print(ival);       // ok: print(int) is visible
    print(3.14);       // ok: calls print(int); print(double) is hidden
}
```

Most readers will not be surprised that the call to `read` is in error. When the compiler processes the call to `read`, it finds the local definition of `read`. That name is a `bool` variable, and we cannot call a `bool`. Hence, the call is illegal.

Exactly the same process is used to resolve the calls to `print`. The declaration of `print(int)` in `fooBar` hides the earlier declarations of `print`. It is as if there is only one `print` function available: the one that takes a single `int` parameter.

When we call `print`, the compiler first looks for a declaration of that name. It finds the local declaration for `print` that takes an `int`. Once a name is found, the compiler ignores uses of that name in any outer scope. Instead, the compiler assumes that the declaration it found is the one for the name we are using. What remains is to see if the use of the name is valid.

In C++, name lookup happens before type checking.

The first call passes a string literal, but the only declaration for `print` that is in scope has a parameter that is an `int`. A string literal cannot be converted to an `int`, so this call is an error. The `print(const string&)` function, which would have matched this call, is hidden and is not considered.

When we call `print` passing a `double`, the process is repeated. The compiler finds the local definition of `print(int)`. The `double` argument can be converted to an `int`, so the call is legal.

Had we declared `print(int)` in the same scope as the other `print` functions, then it would be another overloaded version of `print`. In that case, these calls would be resolved differently, because the compiler will see all three functions:

```
void print(const string &);
void print(double); // overloads the print function
void print(int);    // another overloaded instance
void fooBar2(int ival)
{
    print("Value: "); // calls print(const string &)
    print(ival);      // calls print(int)
    print(3.14);      // calls print(double)
}
```

6.5 Features for Specialized Uses

In this section we'll cover three function-related features that are useful in many, but not all, programs: default arguments, inline and `constexpr` functions, and some facilities that are often used during debugging.

6.5.1 Default Arguments

Some functions have parameters that are given a particular value in most, but not all, calls. In such cases, we can declare that common value as a **default argument** for the function. Functions with default arguments can be called with or without that argument.

For example, we might use a `string` to represent the contents of a window. By default, we might want the window to have a particular height, width, and background character. However, we might also want to allow users to pass values other than the defaults. To accommodate both default and specified values we would declare our function to define the window as follows:

```
typedef string::size_type sz;  // typedef see § 2.5.1 (p. 67)
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

Here we've provided a default for each parameter. A default argument is specified as an initializer for a parameter in the parameter list. We may define defaults for one or more parameters. However, if a parameter has a default argument, all the parameters that follow it must also have default arguments.

Calling Functions with Default Arguments

If we want to use the default argument, we omit that argument when we call the function. Because `screen` provides defaults for all of its parameters, we can call `screen` with zero, one, two, or three arguments:

```
string window;
window = screen();  // equivalent to screen(24,80,' ')
window = screen(66);// equivalent to screen(66,80,' ')
window = screen(66, 256);       // screen(66,256,' ')
window = screen(66, 256, '#'); // screen(66,256,'#')
```

Arguments in the call are resolved by position. The default arguments are used for the trailing (right-most) arguments of a call. For example, to override the default for `backgrnd`, we must also supply arguments for `ht` and `wid`:

```
window = screen(, , '?'); // error: can omit only trailing arguments
window = screen('?');     // calls screen('?',80,' ')
```

Note that the second call, which passes a single character value, is legal. Although legal, it is unlikely to be what was intended. The call is legal because `'?'` is a `char`, and a `char` can be converted (§ 4.11.1, p. 160) to the type of the left-most parameter. That parameter is `string::size_type`, which is an `unsigned` integral type. In this call, the `char` argument is implicitly converted to `string::size_type`,

and is passed as the argument to `height`. On our machine, `'?'` has the hexadecimal value `0x3F`, which is decimal `63`. Thus, this call passes `63` to the `height` parameter.

Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

Default Argument Declarations

Although it is normal practice to declare a function once inside a header, it is legal to redeclare a function multiple times. However, each parameter can have its default specified only once in a given scope. Thus, any subsequent declaration can add a default only for a parameter that has not previously had a default specified. As usual, defaults can be specified only if all parameters to the right already have defaults. For example, given

```
// no default for the height or width parameters
string screen(sz, sz, char = ' ');
```

we cannot change an already declared default value:

```
string screen(sz, sz, char = '*'); // error: redeclaration
```

but we can add a default argument as follows:

```
string screen(sz = 24, sz = 80, char);  // ok: adds default arguments
```

Default arguments ordinarily should be specified with the function declaration in an appropriate header.

Default Argument Initializers

Local variables may not be used as a default argument. Excepting that restriction, a default argument can be any expression that has a type that is convertible to the type of the parameter:

```
// the declarations of wd, def, and ht must appear outside a function
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
string window = screen(); // calls screen(ht(), 80, ' ')
```

Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call:

```
void f2()
{
    def = '*';    // changes the value of a default argument
    sz wd = 100; // hides the outer definition of wd but does not change the default
    window = screen(); // calls screen(ht(), 80, '*')
}
```

Inside `f2`, we changed the value of `def`. The call to `screen` passes this updated value. Our function also declared a local variable that hides the outer `wd`. However, the local named `wd` is unrelated to the default argument passed to `screen`.

EXERCISES SECTION 6.5.1

Exercise 6.40: Which, if either, of the following declarations are errors? Why?

(a)
```
int ff(int a, int b = 0, int c = 0);
```
(b)
```
char *init(int ht = 24, int wd, char bckgrnd);
```

Exercise 6.41: Which, if any, of the following calls are illegal? Why? Which, if any, are legal but unlikely to match the programmer's intent? Why?

```
char *init(int ht, int wd = 80, char bckgrnd = ' ');
```

(a) `init();` (b) `init(24,10);` (c) `init(14, '*');`

Exercise 6.42: Give the second parameter of `make_plural` (§ 6.3.2, p. 224) a default argument of `'s'`. Test your program by printing singular and plural versions of the words `success` and `failure`.

6.5.2 Inline and `constexpr` Functions

In § 6.3.2 (p. 224) we wrote a small function that returned a reference to the shorter of its two `string` parameters. The benefits of defining a function for such a small operation include the following:

- It is easier to read and understand a call to `shorterString` than it would be to read and understand the equivalent conditional expression.
- Using a function ensures uniform behavior. Each test is guaranteed to be done the same way.
- If we need to change the computation, it is easier to change the function than to find and change every occurrence of the equivalent expression.
- The function can be reused rather than rewritten for other applications.

There is, however, one potential drawback to making `shorterString` a function: Calling a function is apt to be slower than evaluating the equivalent expression. On most machines, a function call does a lot of work: Registers are saved before the call and restored after the return; arguments may be copied; and the program branches to a new location.

`inline` Functions Avoid Function Call Overhead

A function specified as **`inline`** (usually) is expanded "in line" at each call. If `shorterString` were defined as `inline`, then this call

```
cout << shorterString(s1, s2) << endl;
```

(probably) would be expanded during compilation into something like

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

The run-time overhead of making `shorterString` a function is thus removed.

We can define `shorterString` as an inline function by putting the keyword `inline` before the function's return type:

```
// inline version: find the shorter of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
        return s1.size() <= s2.size() ? s1 : s2;
}
```

The `inline` specification is only a *request* to the compiler. The compiler may choose to ignore this request.

In general, the `inline` mechanism is meant to optimize small, straight-line functions that are called frequently. Many compilers will not inline a recursive function. A 75-line function will almost surely not be expanded inline.

`constexpr` Functions

A **`constexpr` function** is a function that can be used in a constant expression (§ 2.4.4, p. 65). A `constexpr` function is defined like any other function but must meet certain restrictions: The `return` type and the type of each parameter in a must be a literal type (§ 2.4.4, p. 66), and the function body must contain exactly one `return` statement:

C++ 11

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz();  // ok: foo is a constant expression
```

Here we defined `new_sz` as a `constexpr` that takes no arguments. The compiler can verify—at compile time—that a call to `new_sz` returns a constant expression, so we can use `new_sz` to initialize our `constexpr` variable, `foo`.

When it can do so, the compiler will replace a call to a `constexpr` function with its resulting value. In order to be able to expand the function immediately, `constexpr` functions are implicitly `inline`.

A `constexpr` function body may contain other statements so long as those statements generate no actions at run time. For example, a `constexpr` function may contain null statements, type aliases (§ 2.5.1, p. 67), and `using` declarations.

A `constexpr` function is permitted to return a value that is not a constant:

```
// scale(arg) is a constant expression if arg is a constant expression
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
```

The `scale` function will return a constant expression if its argument is a constant expression but not otherwise:

```
int arr[scale(2)]; //  ok: scale(2) is a constant expression
int i = 2;         //  i is not a constant expression
int a2[scale(i)];  //  error: scale(i) is not a constant expression
```

When we pass a constant expression—such as the literal `2`—then the return is a constant expression. In this case, the compiler will replace the call to `scale` with the resulting value.

If we call `scale` with an expression that is not a constant expression—such as on the `int` object `i`—then the return is not a constant expression. If we use `scale` in a context that requires a constant expression, the compiler checks that the result is a constant expression. If it is not, the compiler will produce an error message.

A `constexpr` function is not required to return a constant expression.

Put `inline` and `constexpr` Functions in Header Files

Unlike other functions, `inline` and `constexpr` functions may be defined multiple times in the program. After all, the compiler needs the definition, not just the declaration, in order to expand the code. However, all of the definitions of a given `inline` or `constexpr` must match exactly. As a result, `inline` and `constexpr` functions normally are defined in headers.

Exercises Section 6.5.2

Exercise 6.43: Which one of the following declarations and definitions would you put in a header? In a source file? Explain why.

(a) `inline bool eq(const BigInt&, const BigInt&) {...}`
(b) `void putValues(int *arr, int size);`

Exercise 6.44: Rewrite the `isShorter` function from § 6.2.2 (p. 211) to be `inline`.

Exercise 6.45: Review the programs you've written for the earlier exercises and decide whether they should be defined as `inline`. If so, do so. If not, explain why they should not be `inline`.

Exercise 6.46: Would it be possible to define `isShorter` as a `constexpr`? If so, do so. If not, explain why not.

6.5.3 Aids for Debugging

C++ programmers sometimes use a technique similar to header guards (§ 2.6.3, p. 77) to conditionally execute debugging code. The idea is that the program will contain debugging code that is executed only while the program is being developed. When the application is completed and ready to ship, the debugging code is turned off. This approach uses two preprocessor facilities: `assert` and `NDEBUG`.

The `assert` Preprocessor Macro

`assert` is a **preprocessor macro**. A preprocessor macro is a preprocessor variable that acts somewhat like an inline function. The `assert` macro takes a single expression, which it uses as a condition:

```
assert(expr);
```

evaluates *expr* and if the expression is false (i.e., zero), then `assert` writes a message and terminates the program. If the expression is true (i.e., is nonzero), then `assert` does nothing.

The `assert` macro is defined in the `cassert` header. As we've seen, preprocessor names are managed by the preprocessor not the compiler (§ 2.3.2, p. 54). As a result, we use preprocessor names directly and do not provide a `using` declaration for them. That is, we refer to `assert`, not `std::assert`, and provide no `using` declaration for `assert`.

As with preprocessor variables, macro names must be unique within the program. Programs that include the `cassert` header may not define a variable, function, or other entity named `assert`. In practice, it is a good idea to avoid using the name `assert` for our own purposes even if we don't include `cassert`. Many headers include the `cassert` header, which means that even if you don't directly include that file, your programs are likely to have it included anyway.

The `assert` macro is often used to check for conditions that "cannot happen." For example, a program that does some manipulation of input text might know that all words it is given are always longer than a threshold. That program might contain a statement such as

```
assert(word.size() > threshold);
```

The `NDEBUG` Preprocessor Variable

The behavior of `assert` depends on the status of a preprocessor variable named `NDEBUG`. If `NDEBUG` is defined, `assert` does nothing. By default, `NDEBUG` is *not* defined, so, by default, `assert` performs a run-time check.

We can "turn off" debugging by providing a `#define` to define `NDEBUG`. Alternatively, most compilers provide a command-line option that lets us define preprocessor variables:

```
$ CC -D NDEBUG main.C  # use /D with the Microsoft compiler
```

has the same effect as writing `#define NDEBUG` at the beginning of `main.C`.

If `NDEBUG` is defined, we avoid the potential run-time overhead involved in checking various conditions. Of course, there is also no run-time check. Therefore, `assert` should be used only to verify things that truly should not be possible. It can be useful as an aid in getting a program debugged but should not be used to substitute for run-time logic checks or error checking that the program should do.

In addition to using `assert`, we can write our own conditional debugging code using `NDEBUG`. If `NDEBUG` is *not* defined, the code between the `#ifndef` and the `#endif` is executed. If `NDEBUG` *is* defined, that code is ignored:

```
void print(const int ia[], size_t size)
{
#ifndef NDEBUG
// __func__ is a local static defined by the compiler that holds the function's name
cerr << __func__ << ": array size is " << size << endl;
#endif
// ...
```

Here we use a variable named `__func__` to print the name of the function we are debugging. The compiler defines `__func__` in every function. It is a local `static` array of `const char` that holds the name of the function.

In addition to `__func__`, which the C++ compiler defines, the preprocessor defines four other names that can be useful in debugging:

`__FILE__` string literal containing the name of the file

`__LINE__` integer literal containing the current line number

`__TIME__` string literal containing the time the file was compiled

`__DATE__` string literal containing the date the file was compiled

We might use these constants to report additional information in error messages:

```
if (word.size() < threshold)
    cerr << "Error: " << __FILE__
         << " : in function " << __func__
         << " at line " << __LINE__ << endl
         << "       Compiled on " << __DATE__
         << " at " << __TIME__ << endl
         << "       Word read was \"" << word
         << "\":  Length too short" << endl;
```

If we give this program a `string` that is shorter than the `threshold`, then the following error message will be generated:

```
Error: wdebug.cc : in function main at line 27
       Compiled on Jul 11 2012 at 20:50:03
       Word read was "foo":  Length too short
```

6.6 Function Matching

In many (if not most) cases, it is easy to figure out which overloaded function matches a given call. However, it is not so simple when the overloaded functions have the same number of parameters and when one or more of the parameters have types that are related by conversions. As an example, consider the following set of functions and function call:

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);  // calls void f(double, double)
```

EXERCISES SECTION 6.5.3

Exercise 6.47: Revise the program you wrote in the exercises in § 6.3.2 (p. 228) that used recursion to print the contents of a `vector` to conditionally print information about its execution. For example, you might print the size of the `vector` on each call. Compile and run the program with debugging turned on and again with it turned off.

Exercise 6.48: Explain what this loop does and whether it is a good use of `assert`:

```
string s;
while (cin >> s && s != sought) { }  // empty body
assert(cin);
```

Determining the Candidate and Viable Functions

The first step of function matching identifies the set of overloaded functions considered for the call. The functions in this set are the **candidate functions**. A candidate function is a function with the same name as the called function and for which a declaration is visible at the point of the call. In this example, there are four candidate functions named `f`.

The second step selects from the set of candidate functions those functions that can be called with the arguments in the given call. The selected functions are the **viable functions**. To be viable, a function must have the same number of parameters as there are arguments in the call, and the type of each argument must match—or be convertible to—the type of its corresponding parameter.

We can eliminate two of our candidate functions based on the number of arguments. The function that has no parameters and the one that has two `int` parameters are not viable for this call. Our call has only one argument, and these functions have zero and two parameters, respectively.

The function that takes a single `int` and the function that takes two `doubles` might be viable. Either of these functions can be called with a single argument. The function taking two `doubles` has a default argument, which means it can be called with a single argument.

When a function has default arguments (§ 6.5.1, p. 236), a call may appear to have fewer arguments than it actually does.

Having used the number of arguments to winnow the candidate functions, we next look at whether the argument types match those of the parameters. As with any call, an argument might match its parameter either because the types match exactly or because there is a conversion from the argument type to the type of the parameter. In this example, both of our remaining functions are viable:

- `f(int)` is viable because a conversion exists that can convert the argument of type `double` to the parameter of type `int`.
- `f(double, double)` is viable because a default argument is provided for the function's second parameter and its first parameter is of type `double`, which exactly matches the type of the argument in the call.

If there are no viable functions, the compiler will complain that there is no matching function.

Finding the Best Match, If Any

The third step of function matching determines which viable function provides the best match for the call. This process looks at each argument in the call and selects the viable function (or functions) for which the corresponding parameter best matches the argument. We'll explain the details of "best" in the next section, but the idea is that the closer the types of the argument and parameter are to each other, the better the match.

In our case, there is only one (explicit) argument in the call. That argument has type `double`. To call `f(int)`, the argument would have to be converted from `double` to `int`. The other viable function, `f(double, double)`, is an exact match for this argument. An exact match is better than a match that requires a conversion. Therefore, the compiler will resolve the call `f(5.6)` as a call to the function that has two `double` parameters. The compiler will add the default argument for the second, missing argument.

Function Matching with Multiple Parameters

Function matching is more complicated if there are two or more arguments. Given the same functions named `f`, let's analyze the following call:

```
f(42, 2.56);
```

The set of viable functions is selected in the same way as when there is only one parameter. The compiler selects those functions that have the required number of parameters and for which the argument types match the parameter types. In this case, the viable functions are `f(int, int)` and `f(double, double)`. The compiler then determines, argument by argument, which function is (or functions are) the best match. There is an overall best match if there is one and only one function for which

- The match for each argument is no worse than the match required by any other viable function
- There is at least one argument for which the match is better than the match provided by any other viable function

If after looking at each argument there is no single function that is preferable, then the call is in error. The compiler will complain that the call is ambiguous.

In this call, when we look only at the first argument, we find that the function `f(int, int)` is an exact match. To match the second function, the `int` argument `42` must be converted to `double`. A match through a built-in conversion is "less good" than one that is exact. Considering only the first argument, `f(int, int)` is a better match than `f(double, double)`.

When we look at the second argument, `f(double, double)` is an exact match to the argument `2.56`. Calling `f(int, int)` would require that `2.56` be converted from `double` to `int`. When we consider only the second parameter, the function `f(double, double)` is a better match.

The compiler will reject this call because it is ambiguous: Each viable function is a better match than the other on one of the arguments to the call. It might be tempting to force a match by explicitly casting (§ 4.11.3, p. 162) one of our arguments. However, in well-designed systems, argument casts should not be necessary.

Casts should not be needed to call an overloaded function. The need for a cast suggests that the parameter sets are designed poorly.

Exercises Section 6.6

Exercise 6.49: What is a candidate function? What is a viable function?

Exercise 6.50: Given the declarations for `f` from page 242, list the viable functions, if any for each of the following calls. Indicate which function is the best match, or if the call is illegal whether there is no match or why the call is ambiguous.

(a) `f(2.56, 42)` (b) `f(42)` (c) `f(42, 0)` (d) `f(2.56, 3.14)`

Exercise 6.51: Write all four versions of `f`. Each function should print a distinguishing message. Check your answers for the previous exercise. If your answers were incorrect, study this section until you understand why your answers were wrong.

6.6.1 Argument Type Conversions

In order to determine the best match, the compiler ranks the conversions that could be used to convert each argument to the type of its corresponding parameter. Conversions are ranked as follows:

1. An exact match. An exact match happens when:
   - The argument and parameter types are identical.
   - The argument is converted from an array or function type to the corresponding pointer type. (§ 6.7 (p. 247) covers function pointers.)
   - A top-level `const` is added to or discarded from the argument.
2. Match through a `const` conversion (§ 4.11.2, p. 162).
3. Match through a promotion (§ 4.11.1, p. 160).
4. Match through an arithmetic (§ 4.11.1, p. 159) or pointer conversion (§ 4.11.2, p. 161).
5. Match through a class-type conversion. (§ 14.9 (p. 579) covers these conversions.)

Matches Requiring Promotion or Arithmetic Conversion

Promotions and conversions among the built-in types can yield surprising results in the context of function matching. Fortunately, well-designed systems rarely include functions with parameters as closely related as those in the following examples.

In order to analyze a call, it is important to remember that the small integral types always promote to `int` or to a larger integral type. Given two functions, one of which takes an `int` and the other a `short`, the `short` version will be called only on values of type `short`. Even though the smaller integral values might appear to be a closer match, those values are promoted to `int`, whereas calling the `short` version would require a conversion:

```
void ff(int);
void ff(short);
ff('a');    // char promotes to int; calls f(int)
```

All the arithmetic conversions are treated as equivalent to each other. The conversion from `int` to `unsigned int`, for example, does not take precedence over the conversion from `int` to `double`. As a concrete example, consider

```
void manip(long);
void manip(float);
manip(3.14); // error: ambiguous call
```

The literal `3.14` is a `double`. That type can be converted to either `long` or `float`. Because there are two possible arithmetic conversions, the call is ambiguous.

Function Matching and `const` Arguments

When we call an overloaded function that differs on whether a reference or pointer parameter refers or points to `const`, the compiler uses the `const`ness of the argument to decide which function to call:

```
Record lookup(Account&);        // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
const Account a;
Account b;

lookup(a);    // calls lookup(const Account&)
lookup(b);    // calls lookup(Account&)
```

In the first call, we pass the `const` object `a`. We cannot bind a plain reference to a `const` object. In this case the only viable function is the version that takes a reference to `const`. Moreover, that call is an exact match to the argument `a`.

In the second call, we pass the nonconst object `b`. For this call, both functions are viable. We can use `b` to initialize a reference to either `const` or nonconst type. However, initializing a reference to `const` from a nonconst object requires a conversion. The version that takes a nonconst parameter is an exact match for `b`. Hence, the nonconst version is preferred.

Pointer parameters work in a similar way. If two functions differ only as to whether a pointer parameter points to `const` or nonconst, the compiler can distinguish which function to call based on the `const`ness of the argument: If the argument is a pointer to `const`, the call will match the function that takes a `const*`; otherwise, if the argument is a pointer to nonconst, the function taking a plain pointer is called.

EXERCISES SECTION 6.6.1

Exercise 6.52: Given the following declarations,

```
void manip(int, int);
double dobj;
```

what is the rank (§ 6.6.1, p. 245) of each conversion in the following calls?

(a) `manip('a', 'z');` (b) `manip(55.4, dobj);`

Exercise 6.53: Explain the effect of the second declaration in each one of the following sets of declarations. Indicate which, if any, are illegal.

```
(a) int calc(int&, int&);
    int calc(const int&, const int&);
(b) int calc(char*, char*);
    int calc(const char*, const char*);
(c) int calc(char*, char*);
    int calc(char* const, char* const);
```

6.7 Pointers to Functions

A function pointer is just that—a pointer that denotes a function rather than an object. Like any other pointer, a function pointer points to a particular type. A function's type is determined by its return type and the types of its parameters. The function's name is not part of its type. For example:

```
// compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

has type `bool(const string&, const string&)`. To declare a pointer that can point at this function, we declare a pointer in place of the function name:

```
// pf points to a function returning bool that takes two const string references
bool (*pf)(const string &, const string &);  // uninitialized
```

Starting from the name we are declaring, we see that `pf` is preceded by a `*`, so `pf` is a pointer. To the right is a parameter list, which means that `pf` points to a function. Looking left, we find that the type the function returns is `bool`. Thus, `pf` points to a function that has two `const string&` parameters and returns `bool`.

The parentheses around `*pf` are necessary. If we omit the parentheses, then we declare `pf` as a function that returns a pointer to `bool`:

```
// declares a function named pf that returns a bool*
bool *pf(const string &, const string &);
```

Using Function Pointers

When we use the name of a function as a value, the function is automatically converted to a pointer. For example, we can assign the address of `lengthCompare` to `pf` as follows:

```
pf = lengthCompare;  // pf now points to the function named lengthCompare
pf = &lengthCompare; // equivalent assignment: address-of operator is optional
```

Moreover, we can use a pointer to a function to call the function to which the pointer points. We can do so directly—there is no need to dereference the pointer:

```
bool b1 = pf("hello", "goodbye");     // calls lengthCompare
bool b2 = (*pf)("hello", "goodbye"); // equivalent call
bool b3 = lengthCompare("hello", "goodbye");  // equivalent call
```

There is no conversion between pointers to one function type and pointers to another function type. However, as usual, we can assign `nullptr` (§ 2.3.2, p. 53) or a zero-valued integer constant expression to a function pointer to indicate that the pointer does not point to any function:

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);

pf = 0;               // ok: pf points to no function
pf = sumLength;       // error: return type differs
pf = cstringCompare; // error: parameter types differ
pf = lengthCompare;  // ok: function and pointer types match exactly
```

Pointers to Overloaded Functions

As usual, when we use an overloaded function, the context must make it clear which version is being used. When we declare a pointer to an overloaded function

```
void ff(int*);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff;  // pf1 points to ff(unsigned)
```

the compiler uses the type of the pointer to determine which overloaded function to use. The type of the pointer must match one of the overloaded functions exactly:

```
void (*pf2)(int) = ff;     // error: no ff with a matching parameter list
double (*pf3)(int*) = ff; // error: return type of ff and pf3 don't match
```

Function Pointer Parameters

Just as with arrays (§ 6.2.4, p. 214), we cannot define parameters of function type but can have a parameter that is a pointer to function. As with arrays, we can write a parameter that looks like a function type, but it will be treated as a pointer:

```
// third parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
```

When we pass a function as an argument, we can do so directly. It will be automatically converted to a pointer:

```
// automatically converts the function lengthCompare to a pointer to function
useBigger(s1, s2, lengthCompare);
```

As we've just seen in the declaration of `useBigger`, writing function pointer types quickly gets tedious. Type aliases (§ 2.5.1, p. 67), along with `decltype` (§ 2.5.3, p. 70), let us simplify code that uses function pointers:

```
// Func and Func2 have function type
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2; // equivalent type
// FuncP and FuncP2 have pointer to function type
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2;  // equivalent type
```

Here we've used `typedef` to define our types. Both `Func` and `Func2` are function types, whereas `FuncP` and `FuncP2` are pointer types. It is important to note that `decltype` returns the function type; the automatic conversion to pointer is not done. Because `decltype` returns a function type, if we want a pointer we must add the `*` ourselves. We can redeclare `useBigger` using any of these types:

```
// equivalent declarations of useBigger using type aliases
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
```

Both declarations declare the same function. In the first case, the compiler will automatically convert the function type represented by `Func` to a pointer.

Returning a Pointer to Function

As with arrays (§ 6.3.3, p. 228), we can't return a function type but can return a pointer to a function type. Similarly, we must write the return type as a pointer type; the compiler will not automatically treat a function return type as the corresponding pointer type. Also as with array returns, by far the easiest way to declare a function that returns a pointer to function is by using a type alias:

```
using F = int(int*, int);      // F is a function type, not a pointer
using PF = int(*)(int*, int); // PF is a pointer type
```

Here we used type alias declarations (§ 2.5.1, p. 68) to define `F` as a function type and `PF` as a pointer to function type. The thing to keep in mind is that, unlike what happens to parameters that have function type, the return type is not automatically converted to a pointer type. We must explicitly specify that the return type is a pointer type:

```
PF f1(int);   // ok: PF is a pointer to function; f1 returns a pointer to function
F f1(int);    // error: F is a function type; f1 can't return a function
F *f1(int);   // ok: explicitly specify that the return type is a pointer to function
```

Of course, we can also declare `f1` directly, which we'd do as

```
int (*f1(int))(int*, int);
```

Reading this declaration from the inside out, we see that `f1` has a parameter list, so `f1` is a function. `f1` is preceded by a `*` so `f1` returns a pointer. The type of that pointer itself has a parameter list, so the pointer points to a function. That function returns an `int`.

For completeness, it's worth noting that we can simplify declarations of functions that return pointers to function by using a trailing return (§ 6.3.3, p. 229):

```
auto f1(int) -> int (*)(int*, int);
```

Using `auto` or `decltype` for Function Pointer Types

If we know which function(s) we want to return, we can use `decltype` to simplify writing a function pointer return type. For example, assume we have two functions, both of which return a `string::size_type` and have two `const string&` parameters. We can write a third function that takes a `string` parameter and returns a pointer to one of these two functions as follows:

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// depending on the value of its string parameter,
// getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string &);
```

The only tricky part in declaring `getFcn` is to remember that when we apply `decltype` to a function, it returns a function type, not a pointer to function type. We must add a `*` to indicate that we are returning a pointer, not a function.

Exercises Section 6.7

Exercise 6.54: Write a declaration for a function that takes two `int` parameters and returns an `int`, and declare a `vector` whose elements have this function pointer type.

Exercise 6.55: Write four functions that add, subtract, multiply, and divide two `int` values. Store pointers to these functions in your `vector` from the previous exercise.

Exercise 6.56: Call each element in the `vector` and print their result.

Chapter Summary

Functions are named units of computation and are essential to structuring even modest programs. Every function has a return type, a name, a (possibly empty) list of parameters, and a function body. The function body is a block that is executed when the function is called. When a function is called, the arguments passed to the function must be compatible with the types of the corresponding parameters.

In C++, functions may be overloaded: The same name may be used to define different functions as long as the number or types of the parameters in the functions differ. The compiler automatically figures out which function to call based on the arguments in a call. The process of selecting the right function from a set of overloaded functions is referred to as function matching.

Defined Terms

ambiguous call Compile-time error that results during function matching when two or more functions provide an equally good match for a call.

arguments Values supplied in a function call that are used to initialize the function's parameters.

assert Preprocessor macro that takes a single expression, which it uses as a condition. When the preprocessor variable `NDEBUG` is not defined, `assert` evaluates the condition and, if the condition is false, writes a message and terminates the program.

automatic objects Objects that exist only during the execution of a function. They are created when control passes through their definition and are destroyed at the end of the block in which they are defined.

best match Function selected from a set of overloaded functions for a call. If a best match exists, the selected function is a better match than all the other viable candidates for at least one argument in the call and is no worse on the rest of the arguments.

call by reference See pass by reference.

call by value See pass by value.

candidate functions Set of functions that are considered when resolving a function call. The candidate functions are all the functions with the name used in the call for which a declaration is in scope at the time of the call.

constexpr Function that may return a constant expression. A `constexpr` function is implicitly `inline`.

default argument Value specified to be used when an argument is omitted in a call to the function.

executable file File, which the operating system executes, that contains code corresponding to our program.

function Callable unit of computation.

function body Block that defines the actions of a function.

function matching Compiler process by which a call to an overloaded function is resolved. Arguments used in the call are compared to the parameter list of each overloaded function.

function prototype Function declaration, consisting of the name, return type, and parameter types of a function. To call a function, its prototype must have been declared before the point of call.

hidden names Names declared inside a scope hide previously declared entities with the same names declared outside that scope.

initializer_list Library class that represents a comma-separated list of objects of a single type enclosed inside curly braces.

inline function Request to the compiler to expand a function at the point of call, if possible. Inline functions avoid the normal function-calling overhead.

link Compilation step in which multiple object files are put together to form an executable program.

local static objects Local objects whose value persists across calls to the function. Local `static` objects that are created and initialized before control reaches their use and are destroyed when the program ends.

local variables Variables defined inside a block.

no match Compile-time error that results during function matching when there is no function with parameters that match the arguments in a given call.

object code Format into which the compiler transforms our source code.

object file File holding object code generated by the compiler from a given source file. An executable file is generated from one or more object files after the files are linked together.

object lifetime Every object has an associated lifetime. Nonstatic objects that are defined inside a block exist from when their definition is encountered until the end of the block in which they are defined. Global objects are created during program startup. Local `static` objects are created before the first time execution passes through the object's definition. Global objects and local `static` objects are destroyed when the `main` function ends.

overload resolution See function matching.

overloaded function Function that has the same name as at least one other function. Overloaded functions must differ in the number or type of their parameters.

parameters Local variables declared inside the function parameter list. Parameters are initialized by the arguments provided in each function call.

pass by reference Description of how arguments are passed to parameters of reference type. Reference parameters work the same way as any other use of references; the parameter is bound to its corresponding argument.

pass by value How arguments are passed to parameters of a nonreference type. A nonreference parameter is a copy of the value of its corresponding argument.

preprocessor macro Preprocessor facility that behaves like an inline function. Aside from `assert`, modern C++ programs make very little use of preprocessor macros.

recursion loop Description of a recursive function that omits a stopping condition and which calls itself until exhasuting the program stack.

recursive function Function that calls itself directly or indirectly.

return type Part of a function declaration that specifies the type of the value that the function returns.

separate compilation Ability to split a program into multiple separate source files.

trailing return type Return type specified after the parameter list.

viable functions Subset of the candidate functions that could match a given call. Viable functions have the same number of parameters as arguments to the call, and each argument type can be converted to the corresponding parameter type.

`()` operator Call operator. Executes a function. The name of a function or a function pointer precedes the parentheses, which enclose a (possibly empty) comma-separated list of arguments.