2

OBJECTS, FUNCTIONS, AND TYPES

In this chapter, you'll learn about objects, functions, and types. We'll examine how to declare variables (objects with identifiers) and functions, take the addresses of objects, and dereference those object pointers. You already learned about object types that are available to C programmers, as well as derived types. The first thing you'll learn in this chapter is one of the last things that I learned: every type in C is either an *object* type or a *function* type.

Objects, Functions, Types, and Pointers

An *object* is storage in which you can represent values. To be precise, an object is defined by the C Standard (ISO/IEC 9899:2018) as a "region of data storage in the execution environment, the contents of which can represent values," with the added note, "when referenced, an object can be interpreted as having a particular type." A variable is an example of an object.

Variables have a declared *type* that tells you the kind of object its value represents. For example, an object with type `int` contains an integer value. The type is important because the collection of bits that represents one type of object will likely have a different value if interpreted as a different type of object. For example, the number 1 is represented in IEEE 754 (the IEEE Standard for Floating-Point Arithmetic) by the bit pattern `0x3f800000` (IEEE 754–2008). But if you were to interpret this same bit pattern as an integer, you'd get the value 1,065,353,216 instead of 1.

Functions are not objects but do have types. A function type is characterized by both its return type as well as the number and types of its parameters.

The C language also has *pointers*, which can be thought of as an *address*—a location in memory where an object or function is stored. A pointer type is derived from a function or object type called the *referenced type*. A pointer type derived from the referenced type `T` is called a *pointer to* `T`.

Because objects and functions are different things, object pointers and function pointers are also different things, and should not be used interchangeably. In the following section, you'll write a simple program that attempts to swap the values of two variables to help you better understand objects, functions, pointers, and types.

Declaring Variables

When you declare a variable, you assign it a type and provide it a name, or *identifier*, by which to reference the variable.

Listing 2-1 declares two integer objects with initial values. This simple program also declares, but doesn't define, a `swap` function to swap those values.

```
#include <stdio.h>

❶ void swap(int, int); // defined in Listing 2-2

int main(void) {
  int a = 21;
  int b = 17;

❷ swap(a, b);
  printf("main: a = %d, b = %d\n", a, b);
  return 0;
}
```

Listing 2-1: Program meant to swap two integers

This example program shows a main function with a single code block between the { } characters. This kind of code block is also known as a *compound statement*. We define two variables, a and b, within the main function. We declare the variables as having the type int and initialize them to 21 and 17, respectively. Each variable must have a declaration. The main function then calls the swap function ❷ to try to swap the values of the two integers. The swap function is declared in this program ❶ but not defined. We'll look at some possible implementations of this function later in this section.

DECLARING MULTIPLE VARIABLES

You can declare multiple variables in any single declaration, but doing so can get confusing if the variables are pointers or arrays, or the variables are different types. For example, the following declarations are all correct:

```
char *src, c;
int x, y[5];
int m[12], n[15][3], o[21];
```

The first line declares two variables, src and c, which have different types. The src variable has a type of char *, and c has a type of char. The second line again declares two variables, x and y, with different types. The variable x has a type int, and y is an array of five elements of type int. The third line declares three arrays—m, n, and o—with different dimensions and numbers of elements.

These declarations are easier to understand if each is on its own line:

```
char *src;    // src has a type of char *
char c;       // c has a type of char
int x;        // x has a type int
int y[5];     // y is an array of 5 elements of type int
int m[12];    // m is an array of 12 elements of type int
int n[15][3]; // n is an array of 15 arrays of 3 elements of type int
int o[21];    // o is an array of 21 elements of type int
```

Readable and understandable code is less likely to have defects.

Swapping Values (First Attempt)

Each object has a storage duration that determines its *lifetime,* which is the time during program execution for which the object exists, has storage, has a constant address, and retains its last-stored value. Objects must not be referenced outside their lifetime.

Local variables such as a and b from Listing 2-1 have *automatic storage duration*, meaning that they exist until execution leaves the block in which they're defined. We are going to attempt to swap the values stored in these two variables.

Listing 2-2 is our first attempt to implement the swap function.

```
void swap(int a, int b) {
  int t = a;
  a = b;
  b = t;
  printf("swap: a = %d, b = %d\n", a, b);
}
```

Listing 2-2: The swap function

The swap function declares two parameters, a and b, that you use to pass arguments to this function. C distinguishes between *parameters*, which are objects declared as part of the function declaration that acquire a value on entry to the function, and *arguments*, which are comma-separated expressions you include in the function call expression. We also declare a temporary variable t of type int in the swap function and initialize it to the value of a. This variable is used to temporarily save the value stored in a so that it is not lost during the swap.

You can now compile and test the complete program by running the generated executable:

```
% ./a.out
swap: a = 17, b = 21
main: a = 21, b = 17
```

This result may be surprising. The variables a and b were initialized to 21 and 17, respectively. The first call to printf within the swap function shows these two values swapped, but the second call to printf in main shows the original values unchanged. Let's examine what happened.

C is a *call-by-value* (also called a *pass-by-value*) language, which means that when you provide an argument to a function, the value of that argument is copied into a distinct variable for use within the function. The swap function assigns the values of the objects you pass as arguments to the respective parameters. When the values of the parameters in the function are changed, the values in the caller are unaffected because they are distinct objects. Consequently, the variables a and b retain their original values in main during the second call to printf. The goal of the program was to swap the values of these two objects. By testing the program, we've discovered it has a bug, or defect.

Swapping Values (Second Attempt)

To repair this bug, you can use pointers to rewrite the swap function. We use the indirection (*) operator to both declare pointers and dereference them, as shown in Listing 2-3.

```
void swap(int *pa, int *pb) {
  int t = *pa;
  *pa = *pb;
  *pb = t;
  return;
}
```

Listing 2-3: The revised swap function using pointers

When used in a function declaration or definition, * acts as part of a pointer declarator indicating that the parameter is a pointer to an object or function of a specific type. In the rewritten swap function, we specify two parameters, pa and pb, and declare them both as type pointers to int.

When you use the unary * operator in expressions within the function, the unary * operator dereferences the pointer to the object. For example, consider the following assignment:

```
pa = pb;
```

This replaces the value of the pointer pa with the value of the pointer pb. Now consider the actual assignment in the swap function:

```
*pa = *pb;
```

This dereferences the pointer pb, reads the referenced value, dereferences the pointer pa, and then overwrites the value at the location referenced by pa with the value referenced by pb.

When you call the swap function in main, you must also place an ampersand (&) character before each variable name:

```
swap(&a, &b);
```

The unary & is the *address-of* operator, which generates a pointer to its operand. This change is necessary because the swap function now accepts pointers to objects of type int as parameters instead of simply values of type int.

Listing 2-4 shows the entire swap program with emphasis on the objects created during execution of this code and their values.

```
#include <stdio.h>
void swap(int *pa, int *pb) {   // pa → a: 21    pb → b: 17
  int t = *pa;                  // t: 21
  *pa = *pb;                    // pa → a: 17    pb → b: 17
  *pb = t;                      // pa → a: 17    pb → b: 21

}
int main(void) {
  int a = 21;                   // a: 21
  int b = 17;                   // b: 17
  swap(&a, &b);
  printf("a = %d, b = %d\n", a, b);    // a: 17    b: 21
  return 0;
}
```

Listing 2-4: Simulated call-by-reference

Upon entering the `main` block, the variables `a` and `b` are initialized to `21` and `17`, respectively. The code then takes the addresses of these objects and passes them to the `swap` function as arguments.

Within the `swap` function, the parameters `pa` and `pb` now both declared to have the type pointer to `int` and contain copies of the arguments passed to `swap` from the calling function (in this case, `main`). These address copies still refer to the exact same objects, so when the values of the objects they reference are swapped in the `swap` function, the contents of the original objects declared in `main` are accessed and also swapped. This approach simulates *call-by-reference* (also known as *pass-by-reference*) by generating object addresses, passing those by value, and then dereferencing the copied addresses to access the original objects.

Scope

Objects, functions, macros, and other C language identifiers have *scope* that delimits the contiguous region where they can be accessed. C has four types of scope: file, block, function prototype, and function.

The scope of an object or function identifier is determined by where it is declared. If the declaration is outside any block or parameter list, the identifier has *file scope*, meaning the scope is the entire text file in which it appears as well as any files included after that point.

If the declaration appears inside a block or within the list of parameters, it has *block scope*, meaning that the identifier it declares is accessible only within the block. The identifiers for `a` and `b` from Listing 2-4 have block scope and can be used to refer to only these variables within the code block in the `main` function in which they're defined.

If the declaration appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope,* which terminates at the end of the function declarator. *Function scope* is the area between the opening `{` of a function definition and its closing `}`. A label name is the only kind of identifier that has function scope. *Labels* are identifiers followed by a colon and identify a statement in a function to which control may be transferred. Chapter 5 covers labels and control transfer.

Scopes can be *nested,* with *inner* and *outer* scopes. For example, you can have a block scope inside another block scope, and every block scope is defined within a file scope. The inner scope has access to the outer scope, but not vice versa. As the name implies, any inner scope must be completely contained within the outer scopes that encompass it.

If you declare the same identifier in both the inner scope and an outer scope, the identifier declared in the outer scope is *hidden* by the identifier within the inner scope, which takes precedence. In this case, naming the identifier will refer to the object in the inner scope; the object from the outer scope is hidden and cannot be referenced by its name. The easiest way to prevent this from becoming a problem is to use different names.

Listing 2-5 demonstrates different scopes and how identifiers declared in inner scopes can hide identifiers declared in outer scopes.

```
int j;  // file scope of j begins

void f(int i) {         // block scope of i begins
  int j = 1;            // block scope of j begins; hides file-scope j
  i++;                  // i refers to the function parameter
  for (int i = 0; i < 2; i++) {  // block scope of loop-local i begins
    int j = 2;          // block scope of the inner j begin; hides outer j
    printf("%d\n", j);  // inner j is in scope, prints 2
  }                     // block scope of the inner i and j ends
  printf("%d\n", j);    // the outer j is in scope, prints 1
}  // the block scope of i and j ends

void g(int j);          // j has function prototype scope; hides file-scope j
```

Listing 2-5: Scope

There is nothing wrong with this code, provided the comments accurately describe your intent. Best practice is to use different names for different identifiers to avoid confusion, which leads to bugs. Using short names such as `i` and `j` is fine for identifiers with small scopes. Identifiers in large scopes should have longer, descriptive names that are unlikely to be hidden in nested scopes. Some compilers will warn about hidden identifiers.

Storage Duration

Objects have a storage duration that determines their lifetime. Altogether, four storage durations are available: automatic, static, thread, and allocated. You've already seen that objects of automatic storage duration are declared within a block or as a function parameter. The lifetime of these objects begins when the block in which they're declared begins execution, and ends when execution of the block ends. If the block is entered recursively, a new object is created each time, each with its own storage.

NOTE *Scope and lifetime are entirely different concepts.* Scope *applies to identifiers, whereas* lifetime *applies to objects. The scope of an identifier is the code region where the object denoted by the identifier can be accessed by its name. The lifetime of an object is the time period for which the object exists.*

Objects declared in file scope have *static* storage duration. The lifetime of these objects is the entire execution of the program, and their stored value is initialized prior to program startup. You can also declare a variable within a block scope to have static storage duration by using the storage-class specifier `static`, as shown in the counting example in Listing 2-6. These objects persist after the function has exited.

```
void increment(void) {
  static unsigned int counter = 0;
  counter++;
  printf("%d ", counter);
}

int main(void) {
  for (int i = 0; i < 5; i++) {
    increment();
  }
  return 0;
}
```

Listing 2-6: A counting example

This program outputs `1 2 3 4 5`. We initialize the static variable `counter` to 0 once at program startup, and increment it each time the `increment` function is called. The lifetime of `counter` is the entire execution of the program, and it will retain its last-stored value throughout its lifetime. You could achieve the same behavior by declaring `counter` with file scope. However, it is good software engineering practice to limit the scope of an object wherever possible.

Static objects must be initialized with a constant value and not a variable:

```
int *func(int i) {
  const int j = i; // ok
  static int k = j; // error
  return &k;
}
```

A constant value refers to literal constants (for example, `1`, `'a'`, or `0xFF`), `enum` members, and the results of operators such as `alignof` or `sizeof`; not `const`-qualified objects.

Thread storage duration is used in concurrent programming and is not covered by this book. *Allocated* storage duration deals with dynamically allocated memory and is discussed in Chapter 6.

Alignment

Object types have alignment requirements that place restrictions on the addresses at which objects of that type may be allocated. An *alignment* represents the number of bytes between successive addresses at which a given object can be allocated. CPUs may have different behavior when accessing aligned data (for example, the data address is a multiple of the data size) versus unaligned data.

Some machine instructions can perform multibyte accesses on non-word boundaries, but there may be a performance penalty. Some platforms cannot access unaligned memory. Alignment requirements may depend on the CPU word size (typically, 16, 32, or 64 bits).

Generally, C programmers need not concern themselves with alignment requirements, because the compiler chooses suitable alignments for its various types. Dynamically allocated memory from `malloc` is required to be sufficiently aligned for all standard types, including arrays and structures. However, on rare occasions, you might need to override the compiler's default choices; for example, to align data on the boundaries of the memory cache lines that must start at power-of-two address boundaries, or to meet other system-specific requirements. Traditionally, these requirements were met by linker commands, or by overallocating memory with `malloc` followed by rounding the user address upward, or similar operations involving other nonstandard facilities.

C11 introduced a simple, forward-compatible mechanism for specifying alignments. Alignments are represented as values of the type `size_t`. Every valid alignment value is a nonnegative integral power of two. An object type imposes a default alignment requirement on every object of that type: a stricter alignment (a larger power of two) can be requested using the alignment specifier (`_Alignas`). You can include an alignment specifier in the declaration specifiers of a declaration. Listing 2-7 uses the alignment specifier to ensure that `good_buff` is properly aligned (`bad_buff` may have incorrect alignment for member-access expressions).

```
struct S {
  int i; double d; char c;
};

int main(void) {
  unsigned char bad_buff[sizeof(struct S)];
  _Alignas(struct S) unsigned char good_buff[sizeof(struct S)];

  struct S *bad_s_ptr = (struct S *)bad_buff;   // wrong pointer alignment
  struct S *good_s_ptr = (struct S *)good_buff; // correct pointer alignment
}
```

Listing 2-7: Use of the `_Alignas` keyword

Alignments are ordered from weaker to stronger (also called stricter) alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any valid, weaker alignment requirement.

Object Types

This section introduces the object types in C. Specifically, we'll cover the Boolean type, character types, and numerical types (including both integer and floating-point types).

Boolean

Objects declared as `_Bool` can store only the values 0 and 1. This *Boolean type* was introduced in C99, and starts with an underscore to differentiate it in existing programs that had already declared their own identifiers named bool or boolean. Identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved. The idea is that the C Standards committee can create new keywords such as _Bool, assuming that you have avoided the use of reserved identifiers. If you haven't, as far as the C Standards committee is concerned, it is your fault for not reading the standard carefully.

If you include the header <stdbool.h>, you can also spell this type as bool and assign it the values true (which expands to the integer constant 1) and false (which expands to the integer constant 0). Here we declare two Boolean variables using both spellings of the type name:

```
#include <stdbool.h>
_Bool flag1 = 0;
bool flag2 = false;
```

Both spellings will work, but it is better to use bool, as this is the long-term direction for the language.

Character Types

The C language defines three *character types*: char, signed char, and unsigned char. Each compiler implementation will define char to have the same alignment, size, range, representation, and behavior as either signed char or unsigned char. Regardless of the choice made, char is a separate type from the other two and is incompatible with both.

The char type is commonly used to represent character data in C language programs. In particular, objects of type char must be able to represent the minimum set of characters required in the execution environment (known as the *basic execution character set*), including upper- and lowercase letters, the 10 decimal digits, the space character, and various punctuation and control characters. The char type is inappropriate for integer data; it is safer to use signed char to represent small signed integer values, and unsigned char to represent small unsigned values.

The basic execution character set suits the needs of many conventional data processing applications, but its lack of non-English letters is an obstacle to acceptance by international users. To address this need, the C Standards committee specified a new, wide type to allow large character sets. You can represent the characters of a large character set as *wide characters* by using the wchar_t type, which generally takes more space than a basic character. Typically, implementations choose 16 or 32 bits to represent a wide character. The C Standard Library provides functions that support both narrow and wide character types.

Numerical Types

C provides several *numerical types* that can be used to represent integers, enumerators, and floating-point values. Chapter 3 covers some of these in more detail, but here's a brief introduction.

Integers

Signed integer types can be used to represent negative numbers, positive numbers, and zero. The signed integer types include `signed char`, `short int`, `int`, `long int`, and `long long int`.

Except for `int` itself, the keyword `int` may be omitted in the declarations for these types, so you might, for example, declare a type by using `long long` instead of `long long int`.

For each signed integer type, there is a corresponding *unsigned integer type* that uses the same amount of storage: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int`. The unsigned types can be used to represent only positive numbers and zero.

The signed and unsigned integer types are used to represent integers of various sizes. Each platform (current or historical) determines the size for each of these types, given some constraints. Each type has a minimum representable range. The types are ordered by width, guaranteeing that *wider* types are at least as large as *narrower* types so that an object of type `long long int` can represent all values that an object of type `long int` can represent, an object of type `long int` can represent all values that can be represented by an object of type `int`, and so forth.

The `int` type usually has the natural size suggested by the architecture of the execution environment, so the size would be 16 bits wide on a 16-bit architecture, and 32 bits wide on a 32-bit architecture. You can specify actual-width integers by using type definitions from the `<stdint.h>` or `<inttypes.h>` headers, like `uint32_t`. These headers also provide type definitions for the widest available integer types: `uintmax_t` and `intmax_t`.

Chapter 3 covers integer types in excruciating detail.

Enums

An *enumeration*, or `enum`, allows you to define a type that assigns names (*enumerators*) to integer values in cases with an enumerable set of constant values. The following are examples of enumerations:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
enum cardinal_points { north = 0, east = 90, south = 180, west = 270 };
enum months { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

If you don't specify a value to the first enumerator with the `=` operator, the value of its enumeration constant is 0, and each subsequent enumerator without an `=` adds 1 to the value of the previous enumeration constant. Consequently, the value of `sun` in the `day` enumeration is 0, `mon` is 1, and so forth.

You can also assign specific values to each enumerator, as shown by the `cardinal_points` enumeration. Using `=` with enumerators may produce enumeration constants with duplicate values, which can be a problem if you incorrectly assume all the values are unique. The `months` enumeration sets the first enumerator at 1, and each subsequent enumerator that isn't specifically assigned a value will be incremented by 1.

The actual value of the enumeration constant must be representable as an `int`, but its type is implementation defined. For example, Visual C++ uses a `signed int`, and GCC uses an `unsigned int`.

Floating-Point Types

The C language supports three *floating-point types*: `float`, `double`, and `long double`. Floating-point arithmetic is similar to, and often used as a model for, the arithmetic of real numbers. The C language supports a variety of floating-point representations including, on most systems, the IEEE Standard for Floating-Point Arithmetic (IEEE 754–2008). The choice of floating-point representation is implementation dependent. Chapter 3 covers floating-point types in detail.

void

The `void` type is a rather strange type. The keyword `void` (by itself) means "cannot hold any value." For example, you can use it to indicate that a function doesn't return a value, or as the sole parameter of a function to indicate that the function takes no arguments. On the other hand, the *derived type* `void *` means that the pointer can reference *any* object. I'll discuss derived types later in this chapter.

Function Types

Function types are derived types. In this case, the type is derived from the return type and the number and types of its parameters. The return type of a function cannot be an array type.

When you declare a function, you use the *function declarator* to specify the name of the function and the return type. If the declarator includes a parameter type list and a definition, the declaration of each parameter must include an identifier, except parameter lists with only a single parameter of type `void`, which needs no identifier.

Here are a few function type declarations:

```
int f(void);
int *fip();
void g(int i, int j);
void h(int, int);
```

First, we declare a function `f` with no parameters that returns an `int`. Next, we declare a function `fip` with no specified parameters that returns a pointer to an `int`. Finally, we declare two functions, `g` and `h`, each returning `void` and taking two parameters of type `int`.

Specifying parameters with identifiers (as done here with `g`) can be problematic if an identifier is a macro. However, providing parameter names is good practice for self-documenting code, so omitting the identifiers (as done with `h`) is not typically recommended.

In a function declaration, specifying parameters is optional. However, failing to do so is occasionally problematic. If you were to write the function declaration for `fip` in C++, it would declare a function accepting no arguments and returning an `int *`. In C, `fip` declares a function accepting any number of arguments of any type and returning an `int *`. You should never declare functions with an empty parameter list in C. First, this is a deprecated feature of the language that may be removed in the future. Second, the code could be ported to C++, so explicitly list parameter types and use `void` when there are no parameters.

A function type with a parameter type list is known as a *function prototype*. A function prototype informs the compiler about the number and types of parameters a function accepts. Compilers use this information to verify that the correct number and type of parameters are used in the *function definition* and any calls to the function.

The function definition provides the actual implementation of the function. Take a look at the following function definition:

```
int max(int a, int b)
{ return a > b ? a : b; }
```

The return type specifier is `int`; the function declarator is `max(int a, int b)`; and the function body is `{ return a > b ? a : b; }`. The specification of a function type must not include any type qualifiers (see "Type Qualifiers" on page 32). The function body itself uses the condition operator (`? :`), which is explained further in Chapter 4. This expression states that if `a` is greater than `b`, return `a`; otherwise, return `b`.

Derived Types

Derived types are types that are constructed from other types. These include pointers, arrays, type definitions, structures, and unions, all of which we'll cover here.

Pointer Types

A *pointer type* is derived from the function or object type that it points to, called the *referenced* type. A pointer provides a reference to an entity of the referenced type.

The following three declarations declare a pointer to `int`, a pointer to `char`, and a pointer to `void`:

```
int *ip;
char *cp;
void *vp;
```

Earlier in the chapter, I introduced the address-of (&) and indirection (*) operators. You use the & operator to take the address of an object or function. If the object is an int, for example, the result of the operator has the type pointer to int:

```
int i = 17;
int *ip = &i;
```

We declare the variable ip as a pointer to int and assign it the address of i. You can also use the operator on the result of the operator:

```
ip = &*ip;
```

Dereferencing ip by using the indirection operator resolves to the actual object i. Taking the address of *ip by using the & operator retrieves the pointer, so these two operations cancel each other out.

The unary * operator converts a pointer to a type into a value of that type. It denotes *indirection* and operates only on pointers. If the operand points to a function, the result of using the * operator is the function designator, and if it points to an object, the result is a value of the designated object. For example, if the operand is a pointer to int, the result of the indirection operator has the type int. If the pointer is not pointing to a valid object or function, bad things may happen.

Arrays

An *array* is a contiguously allocated sequence of objects that all have the same element type. Array types are characterized by their element types and the number of elements in the array. Here we declare an array of 11 elements of type int identified by ia, and an array of 17 elements of type pointer to float identified by afp:

```
int ia[11];
float *afp[17];
```

You use square brackets ([]) to identify an element of an array. For example, the following contrived code snippet creates the string "0123456789" to demonstrate how to assign values to the elements of an array:

```
char str[11];
for (unsigned int i = 0; i < 10; ++i) {
❶ str[i] = '0' + i;
}
str[10] = '\0';
```

The first line declares an array of char with a bound of 11. This allocates sufficient storage to create a string with 10 characters plus a null character.

The for loop iterates 10 times, with the values of i ranging from 0 to 9. Each iteration assigns the result of the expression '0' + i to str[i]. Following the end of the loop, the null character is copied to the final element of the array str[10].

In the expression at ❶, str is automatically converted to a pointer to the first member of the array (an object of type char), and i has an unsigned integer type. The subscript ([]) operator and addition (+) operator are defined so that str[i] is identical to *(str + i). When str is an array object (as it is here), the expression str[i] designates the ith element of the array (counting from 0). Because arrays are indexed starting at 0, the array char str[11] is indexed from 0 to 10, with 10 being the last element, as referenced on the last line of this example.

If the operand of the unary & operator is the result of a [] operator, the result is as if the & operator were removed and the [] operator were changed to a + operator. For example, &str[10] is the same as str + 10.

You can also declare multidimensional arrays. Listing 2-8 declares arr in the function main as a two-dimensional 5 × 3 array of type int, also referred to as a *matrix*.

```
void func(int arr[5]);
int main(void) {
  unsigned int i = 0;
  unsigned int j = 0;
  int arr[3][5];
❶ func(arr[i]);
❷ int x = arr[i][j];
  return 0;
}
```

Listing 2-8: Matrix operations

More precisely, arr is an array of three elements, each of which is an array of five elements of type int. When you use the expression arr[i] at ❶ (which is equivalent to *(arr+i)), the following occurs:

1. arr is converted to a pointer to the initial array of five elements of type int starting at arr[i].
2. i is scaled to the type of arr by multiplying i by the size of one array of five int objects.
3. The results from steps 1 and 2 are added.
4. Indirection is applied to the result to produce an array of five elements of type int.

When used in the expression arr[i][j] at ❷, that array is converted to a pointer to the first element of type int, so arr[i][j] produces an object of type int.

TYPE DEFINITIONS

You use a typedef to declare an alias for an existing type; it never creates a new type. For example, each of the following declarations creates a new type alias:

```
typedef unsigned int uint_type;
typedef signed char schar_type, *schar_p, (*fp)(void);
```

On the first line, we declare uint_type as an alias for the type unsigned int. On the second line, we declare schar_type as an alias for signed char, schar_p as an alias for signed char *, and fp as an alias for signed char(*)(void). Identifiers that end in _t in the standard headers are type definitions (aliases for existing types). Generally speaking, you should not follow this convention in your own code because the C Standard reserve identifiers that match the patterns int[0-9a-z_]*_t and uint[0-9a-z_]*_t, and Portable Operating System Interface (POSIX) reserves all identifiers that end in _t. If you define identifiers that use these names, they may collide with names used by the implementation, which can cause problems that are difficult to debug.

Structures

A *structure type* (also known as a struct) contains sequentially allocated member objects. Each object has its own name and may have a distinct type—unlike arrays, which must all be of the same type. Structures are similar to record types found in other programming languages. Listing 2-9 declares an object identified by sigline that has a type of struct sigrecord and a pointer to the sigline object identified by sigline_p.

```
struct sigrecord {
  int signum;
  char signame[20];
  char sigdesc[100];
} sigline, *sigline_p;
```

Listing 2-9: struct sigrecord

The structure has three member objects: signum is an object of type int, signame is an array of type char consisting of 20 elements, and sigdesc is an array of type char consisting of 100 elements.

Structures are useful for declaring collections of related objects and may be used to represent things such as a date, customer, or personnel record. They are especially useful for grouping objects that are frequently passed together as arguments to a function, so you don't need to repeatedly pass individual objects separately.

Once you have defined a structure, you'll likely want to reference its members. You reference members of an object of the structure type by using the structure member (`.`) operator. If you have a pointer to a structure, you can reference its members with the structure pointer(`->`)operator. Listing 2-10 demonstrates the use of each operator.

```
  sigline.signum = 5;
  strcpy(sigline.signame, "SIGINT");
  strcpy(sigline.sigdesc, "Interrupt from keyboard");

❶ sigline_p = &sigline;

  sigline_p->signum = 5;
  strcpy(sigline_p->signame, "SIGINT");
  strcpy(sigline_p->sigdesc, "Interrupt from keyboard");
```

Listing 2-10: Referencing structure members

The first three lines of Listing 2-10 directly access members of the `sigline` object by using the `.` operator. At ❶, we assign the pointer to `sigline_p` to the address of the `sigline` object. In the final three lines of the program, we indirectly access the members of the `sigline` object by using the `->` operator through the `sigline_p` pointer.

Unions

Union types are similar to structures, except that the memory used by the member objects overlaps. Unions can contain an object of one type at one time, and an object of a different type at a different time, but never both objects at the same time, and are primarily used to save memory. Listing 2-11 shows the union `u` that contains three structures: `n`, `ni`, and `nf`. This union might be used in a tree, graph, or other data structure that has some nodes that contain integer values (`ni`) and other nodes that contain floating-point values (`nf`).

```
union {
  struct {
    int type;
  } n;
  struct {
    int type;
    int intnode;
  } ni;
  struct {
    int type;
    double doublenode;
  } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
```

Listing 2-11: Unions

As with structures, you can access union members via the . operator. Using a pointer to a union, you can reference its members with the -> operator. In Listing 2-11, the type member in the nf struct of the union is referenced as u.nf.type, and the doublenode member is referenced as u.nf .doublenode. Code that uses this union will typically check the type of the node by examining the value stored in u.n.type and then accessing either the intnode or doublenode struct depending on the type. If this had been implemented as a structure, each node would contain storage for both the intnode and the doublenode members. The use of a union allows the same storage to be used for both members.

Tags

Tags are a special naming mechanism for structs, unions, and enumerations. For example, the identifier s appearing in the following structure is a *tag*:

```
struct s {
  /---snip---/
};
```

By itself, a tag is not a type name and cannot be used to declare a variable (Saks 2002). Instead, you must declare variables of this type as follows:

```
struct s v;   // instance of struct s
struct s *p;  // pointer to struct s
```

The names of unions and enumerations are also tags and not types, meaning that they cannot be used alone to declare a variable. For example:

```
enum day { sun, mon, tue, wed, thu, fri, sat };
day today;  // error
enum day tomorrow;  // OK
```

The tags of structures, unions, and enumerations are defined in a separate *namespace* from ordinary identifiers. This allows a C program to have both a tag and another identifier with the same spelling in the same scope:

```
enum status { ok, fail };  // enumeration
enum status status(void);  // function
```

You can even declare an object s of type struct s:

```
struct s s;
```

This may not be good practice, but it is valid in C. You can think of struct tags as type names and define an alias for the tag by using a typedef. Here's an example:

```
typedef struct s { int x; } t;
```

This now allows you to declare variables of type `t` instead of `struct s`. The tag name in `struct`, `union`, and `enum` is optional, so you can just dispense with it entirely:

```
typedef struct { int x; } t;
```

This works fine except in the case of self-referential structures that contain pointers to themselves:

```
struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

If you omit the tag on the first line, the compiler may complain because the referenced structure on lines 3 and 4 has not yet been declared, or because the whole structure is not used anywhere. Consequently, you have no choice but to declare a tag for the structure, but you can declare a `typedef` as well:

```
typedef struct tnode {
  int count;
  struct tnode *left;
  struct tnode *right;
} tnode;
```

Most C programmers use a different name for the tag and the `typedef`, but the same name works just fine. You can also define this type before the structure so that you can use it to declare the `left` and `right` members that refer to other objects of type `tnode`:

```
typedef struct tnode tnode;
struct tnode {
  int count;
  tnode *left
  tnode *right;
} tnode;
```

Type definitions can improve code readability beyond their use with structures. For example, all three of the following declarations of the `signal` function specify the same type:

```
typedef void fv(int), (*pfv)(int);
void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Type Qualifiers

All the types examined so far have been unqualified types. Types can be *qualified* by using one or more of the following qualifiers: `const`, `volatile`, and `restrict`. Each of these qualifiers changes behaviors when accessing objects of the qualified type.

The qualified and unqualified versions of types can be used interchangeably as arguments to functions, return values from functions, and members of unions.

NOTE *The* `_Atomic` *type qualifier, available since C11, supports concurrent programs.*

const

Objects declared with the `const` qualifier (`const`-qualified types) are not modifiable. In particular, they're not assignable but can have constant initializers. This means objects with `const`-qualified types can be placed in read-only memory by the compiler, and any attempt to write to them will result in a runtime error:

```
const int i = 1; // const-qualified int
i = 2; // error: i is const-qualified
```

It's possible to accidentally convince your compiler to change a `const`-qualified object for you. In the following example, we take the address of a `const`-qualified object `i` and tell the compiler that this is actually a pointer to an `int`:

```
const int i = 1;  // object of const-qualified type
int *ip = (int *)&i;
*ip = 2;  // undefined behavior
```

C does not allow you to cast away the `const` if the original was declared as a `const`-qualified object. This code might appear to work, but it's defective and may fail later. For example, the compiler might place the `const`-qualified object in read-only memory, causing a memory fault when trying to store a value in the object at runtime.

C allows you to modify an object that is pointed to by a `const`-qualified pointer by casting the `const` away, provided that the original object was not declared `const`:

```
int i = 12;
const int j = 12;
const int *ip = &i;
const int *jp = &j;
*(int *)ip = 42; // ok
*(int *)jp = 42; // undefined behavior
```

volatile

Objects of volatile-qualified types serve a special purpose. Static volatile-qualified objects are used to model memory-mapped input/output (I/O) ports, and static constant volatile-qualified objects model memory-mapped input ports such as a real-time clock.

The values stored in these objects may change without the knowledge of the compiler. For example, every time the value from a real-time clock is read, it may change, even if the value has not been written to by the C program. Using a volatile-qualified type lets the compiler know that the value may change, and ensures that every access to the real-time clock occurs (otherwise, an access to the real-time clock may be optimized away or replaced by a previously read and cached value). In the following code, for example, the compiler must generate instructions to read the value from port and then write this value back to port:

```
volatile int port;
port = port;
```

Without the volatile qualification, the compiler would see this as a *no-op* (a programming statement that does nothing) and potentially eliminate both the read and the write.

Also, volatile-qualified types are used for communications with signal handlers and with setjmp/longjmp (refer to the C Standard for information on signal handlers and setjmp/longjmp). Unlike in Java and other programming languages, volatile-qualified types in C should not be used for synchronization between threads.

restrict

A restrict-qualified pointer is used to promote optimization. Objects indirectly accessed through a pointer frequently cannot be fully optimized because of potential *aliasing*, which occurs when more than one pointer refers to the same object. Aliasing can inhibit optimizations, because the compiler can't tell if portions of an object can change values when another apparently unrelated object is modified, for example.

The following function copies n bytes from the storage referenced by q to the storage referenced by p. The function parameters p and q are both restrict-qualified pointers:

```
void f(unsigned int n, int * restrict p, int * restrict q) {
  while (n-- > 0) {
    *p++ = *q++;
  }
}
```

Because both p and q are restrict-qualified pointers, the compiler can assume that an object accessed through one of the pointer parameters is not also accessed through the other. The compiler can make this assessment based solely on the parameter declarations without analyzing the function body. Although using `restrict`-qualified pointers can result in more efficient code, you must ensure that the pointers do not refer to overlapping memory to prevent undefined behavior.

Exercises

Try these code exercises on your own:

1. Add a `retrieve` function to the counting example from Listing 2-6 to retrieve the current value of `counter`.
2. Declare an array of three pointers to functions and invoke the appropriate function based on an index value passed in as an argument.

Summary

In this chapter, you learned about objects and functions and how they differ. You learned how to declare variables and functions, take the addresses of objects, and dereference those object pointers. You also learned about most of the object types that are available to C programmers as well as derived types.

We'll return to these types in later chapters to explore in more detail how they can be best used to implement your designs. In the next chapter, I provide detailed information about the two kinds of arithmetic types: integers and floating-point types.