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 named identifiers) and functions, take the addresses of objects, and dereference those object pointers. Each object or function instance has a type. You've already seen some types that are available to C programmers. 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.

Entities

An *object* is storage in which you can represent values. To be precise, an object is defined by the C standard (ISO/IEC 9899:2024) 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. Type is important because the collection of bits that represent one type of object can have a different value if interpreted as a different type of object. For example, the number 1 is represented in the IEEE Standard for Floating-Point Arithmetic by the bit pattern `0x3f800000` (IEEE 754-2019). 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 and 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.

Just like objects and functions, object pointers and function pointers are different things and must not be interchanged. 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 the variable is referenced. Optionally, you can also *initialize* 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>
#include <stdlib.h>

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

int main() {
  int a = 21;
  int b = 17;
❷ swap(a, b);
  printf("main: a = %d, b = %d\n", a, b);
  return EXIT_SUCCESS;
}
```

Listing 2-1: A program meant to swap two integers

This example program shows a `main` function with a single *compound statement* that includes the { } characters and all the statements between them (also referred to as a *block*). 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 become confusing if the variables are pointers or arrays or if the variables are of 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 declared. We're going to try to swap the values stored in these two variables. Listing 2-2 shows 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: A first attempt at implementing the swap function

The `swap` function is declared with two parameters, `a` and `b`, that we 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 we 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's not lost during the swap.

We can now run the generated executable to test the program:

```
% ./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 that these two values were 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 their respective parameters. When the parameter values in the function are changed, the argument 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, we 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;
}
```

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 declare two parameters, `pa` and `pb`, both having the type pointer to `int`.

The unary * operator denotes indirection. If its operand has type pointer to `T`, the result of the operation has type `T`. 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 assignment in the `swap` function:

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

The `*pb` operation reads the value referenced by `pb`, while the `*pa` operation reads the location referenced by `pa`. The value referenced by `pb` is then written to the location referenced by `pa`.

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

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

The unary `&` (*address-of*) operator generates a pointer to its operand. This change is necessary because the `swap` function now accepts arguments of type pointer to `int` instead 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>
#include <stdlib.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() {
  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 EXIT_SUCCESS;
}
```

Listing 2-4: A simulated call-by reference

Upon entering the `main` function 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` are now both declared as 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 their referenced objects are swapped in the swap function, the contents of the original objects declared in main are 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.

Object Types

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

Boolean

A Boolean data type has one of two possible values (true and false) that represent the two truth values of logic and Boolean algebra. Objects declared as bool can store only the values true and false.

RESERVED IDENTIFIERS

A *Boolean type* was introduced in C99 starting with an underscore _Bool to differentiate it in existing programs that had already declared their own identifiers named bool. Identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved. The C standards committee often creates new keywords such as _Bool under the assumption that you have avoided the use of reserved identifiers. If you haven't, as far as the C standards committee is concerned, it's your fault for not having read the standard more carefully. C23 added the bool keyword but retained _Bool as an alternative spelling, and bool is now the preferred spelling. The keywords false and true are constants of type bool with a value of 0 for false and 1 for true. If you are using an older version of C, you can include the header <stdbool.h>, still spell this type as bool, and assign it the values true and false.

The following example declares a function called arm_detonator that takes a single int argument and returns a value of type bool:

```
bool arm_detonator(int);

void arm_missile(void) {
  bool armed = arm_detonator(3);
  if (armed) puts("missile armed");
  else puts("missile disarmed");
}
```

The `arm_missile` function calls the `arm_detonator` function and assigns the return value to the Boolean variable `armed`. This value can then be tested to determine whether the missile is armed.

Historically, Boolean values were represented by integers and still behave as integers. They can be stored in integer variables and used anywhere integers would be valid, including in indexing, arithmetic, parsing, and formatting. C guarantees that any two true values will compare equally (which was impossible to achieve before the introduction of the `bool` type). You should use the `bool` type to represent Boolean values.

Character

The C language defines the following character types: `char`, `signed char`, and `unsigned char`. Each compiler implementation defines `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. Objects of type `char` can represent the *basic execution character set*—the minimum set of characters required in the execution environment—including upper- and lowercase letters, the 10 decimal digits, the space character, punctuation, and control characters. The `char` type is inappropriate for integer data; use `signed char` to represent small, signed integer values, and use `unsigned char` to represent small, unsigned integer values.

The size of objects of type `char` is always 1 byte, and its width is `CHAR_BIT` bits. The `CHAR_BIT` macro from `<limits.h>` defines the number of bits in a byte. The value of `CHAR_BIT` macro cannot be less than 8, and on most modern platforms, it is 8.

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. The `wchar_t` type was not designed to support Unicode and has consequently fallen out of favor for most implementations with the notable exception of Microsoft Visual Studio.

Arithmetic

C provides several *arithmetic 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.

Integer

Signed integer types can be used to represent negative numbers, positive numbers, and zero. The standard signed integer types include `signed char`, `short int`, `int`, `long int`, and `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 represent positive numbers and zero. These unsigned integer types along with type `bool` make up the standard unsigned integer types.

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`.

The signed and unsigned integer types are used to represent integers of various widths. Each platform determines the width 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. This means 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 implementation-defined minimum and maximum representable values for integer types are specified in the `<limits.h>` header file.

Extended integer types may be provided in addition to the standard integer types. They are implementation defined, meaning that their width, precision, and behavior are up to the compiler. Extended integer types are typically larger than the standard integer types, for example, `__int128`.

In addition to the standard and extended integer types, C23 adds *bit-precise integer types*. These types accept an operand specifying the width of the integer, so a `_BitInt(32)` is a signed 32-bit integer, and an `unsigned _BitInt(32)` is an unsigned 32-bit integer. Bit-precise integer types do not require their width to be a power of two; the maximum width supported is specified by `BITINT_MAXWIDTH` (which must be at least the same as the width of `unsigned long long`).

The `int` type is typically assigned the natural width suggested by the architecture of the execution environment, for example, 16 bits on a 16-bit architecture and 32 bits on a 32-bit or 64-bit architecture. You can specify actual-width integers by using type definitions from the `<stdint.h>` or `<inttypes.h>` header, like `uint32_t`. These headers also provide type definitions for the greatest-width integer types: `uintmax_t` and `intmax_t`. The `intmax_t` type, for example, can represent any value of any signed integer type with the possible exceptions of signed bit-precise integer types and of signed extended integer types.

Chapter 3 covers integer types in excruciating detail.

enum

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 that 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.

Starting with C23, you can specify the underlying type of the enumeration. For portability and other reasons (Meneide and Pygott 2022), it is always better to specify the enumeration type. In the following example, the enumeration constant `a0` can be assigned the value `0xFFFFFFFFFFFFFFFFULL` because the type is specified as `unsigned long long`:

```
enum a : unsigned long long {
  a0 = 0xFFFFFFFFFFFFFFFFULL
};
```

An omitted type is implementation defined. Visual C++ uses a `signed int` for the type, and GCC uses an `unsigned int`.

Floating

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, representations in the IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019). ISO/IEC 60559:2011 has content identical to IEEE 754-2019 but is referenced by the C standard because it is published by the same standards organization. The choice of floating-point representation is implementation defined. Chapter 3 covers floating types in detail.

The C language supports three standard floating types: `float`, `double`, and `long double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`.

C23 adds three *decimal floating types* (ISO/IEC TS 18661-2:2015), designated as `_Decimal32`, `_Decimal64`, and `_Decimal128`. Respectively, these have the decimal32, decimal64, and decimal128 IEC 60559 formats. Decimal floating types are not covered in this book.

The standard floating types and the decimal floating types are collectively called the *real floating types*.

There also are three *complex types*, designated as `float complex`, `double complex`, and `long double complex`. Complex types are not covered in this book.

The real floating and complex types are collectively called the *floating types*. Figure 2-1 shows the hierarchy of floating types.

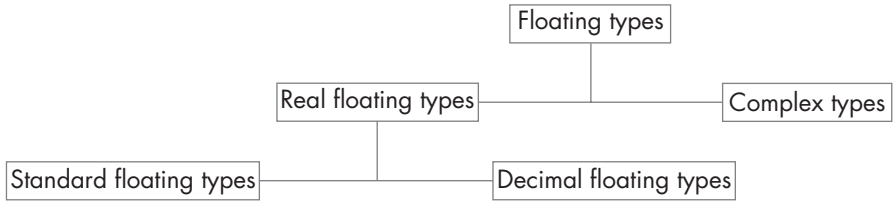

Figure 2-1: The hierarchy of floating types

Complex types and decimal floating types are not covered in detail in this book.

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.

Derived Types

Derived types are constructed from other types. These include function types, pointer types, array types, type definitions, structure types, and union types, all of which are covered here.

Function

A *function type* is derived from the return type and the number and types of its parameters. A function can return any complete object type except for 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 fprime();
int *fip();
void g(int i, int j);
void h(int, int);
```

First, we declare two functions, `f` and `fprime`, with no parameter that returns an `int`. Next, we declare a function `fip` with no parameters that return 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 can be problematic. Prior to C23, `fip` declares a function accepting any number of arguments of any type and returning an `int *`. The same function declaration for `fip` in C++ declares a function accepting no arguments and returning an `int *`. Starting with C23, a function declarator with an empty parameter list declares a prototype for a function that takes no arguments (like it does in C++).

A function type is also 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. Consider 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 XX). The function body itself uses the condition operator (`? :`), which is explained in Chapter 4. This expression states that if `a` is greater than `b`, return `a`; otherwise, return `b`.

Pointer

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`. 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 = 0; // compliant
char *cp = NULL; // good
void *vp = nullptr; // better
```

Each pointer is initialized to a null pointer value. A null pointer value can be specified as an integer constant expression with the value 0, `(void *)0`, or the predefined constant `nullptr`. The `NULL` macro is defined in `<stddef.h>` as a null pointer constant. If a null pointer constant or the value `nullptr` is converted to a pointer type, the resulting null pointer is guaranteed to compare unequally to a pointer to any object or function.

The `nullptr` constant was introduced in C23 and has advantages to using `NULL` (Gustedt 2022). Table 2-1 shows common `NULL` values and their associated types.

Table 2-1: Common `NULL` Values and Their Associated Types

Value	Type
`0`	`int`
`0L`	`long`
`(void *)0`	`void *`

These different types can have surprising results when invoking a type-generic macro with a `NULL` argument. The conditional expression `(true ? 0 : NULL)` is always defined, regardless of the type of `NULL`. However, the conditional expression `(true ? 1 : NULL)` is a constraint violation if `NULL` has type `void *`.

A `NULL` argument that is passed as a sentinel value to a variadic function, such as the Portable Operating System Interface (POSIX) `execl` function, that expects a pointer can have severe consequences. On most modern architectures, the `int` and `void *` types have different sizes. If `NULL` is defined as 0 on such an architecture, an incorrectly sized argument is passed to the variadic function.

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;
```

The second declaration declares the variable `ip` as a pointer to `int` and initializes it to the address of `i`. You can also use the `&` operator on the result of the `*` operator:

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

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

The unary * operator converts a pointer to a type `T` into a value of type `T`. 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 type `int`. If the pointer is not pointing to a valid object or function, the behavior is undefined.

Array

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) {
  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]`, and `str` now contains the string `"0123456789"`.

In the expression `str[i]`, `str` is automatically converted to a pointer to the first member of the array (pointer to `char`), and `i` has an unsigned integer type. The subscript (`[]`) and addition (`+`) operators 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 `i`th 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`:

```
&str[10] → &*(str + 10) → str + 10
```

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

```
#include <stdlib.h>
void func(int arr[5]);
int main() {
  unsigned int i = 0;
  unsigned int j = 0;
  int arr[3][5];
❶ func(arr[i]);
❷ int x = arr[i][j];
  return EXIT_SUCCESS;
}
```

Listing 2-5: 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 sum to produce an array of five elements of type `int`.

When used in the expression `arr[i][j]` 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 `typedef` to declare an alias for an existing type; it never creates a new type. For example, each of the following declarations creates at least one 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). You should not follow this convention in your own code because the C standard reserves identifiers that match the patterns `int[0-9a-z_]*_t` and `uint[0-9a-z_]*_t`, and 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.

Structure

A *structure type* (also known as a `struct`) contains sequentially allocated members. Each member has its own name and may have a distinct type—unlike array elements, which must all be of the same type. Structures are like record types found in other programming languages.

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.

Listing 2-6 declares a `struct` named `sigline` with type `struct sigrecord` and a pointer to `struct sigrecord` named `sigline_p`.

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

Listing 2-6: A `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.

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 2-7 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 2-7: Referencing structure members

The first three lines of Listing 2-7 directly access members of the `sigline` object by using the dot (`.`) operator. We assign the address of the `sigline` object to the `sigline_p` pointer ❶. 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.

Union

Union types are like structures, except that the memory used by the member objects overlaps. Unions provide multiple different ways to look at the same memory.

Listing 2-8 shows a union that contains a single member `f` of type `float` and a `struct` that contains three bitfields of type `uint32_t`: `significand`, `exponent`, and `sign`.

```
static_assert(
  (__STDC_IEC_60559_BFP__ >= 202311L || __STDC_IEC_559__ == 1)
  && __STDC_ENDIAN_LITTLE__
);

union {
  float f;
  struct {
    uint32_t significand : 23;
    uint32_t exponent : 8;
    uint32_t sign : 1;
  };
} float_encoding;
```

Listing 2-8: Decomposing a `float` using a `union`

This allows a (low-level) C programmer to use the entire floating-point value and examine (and possibly modify) its constituent parts. This union is not portable because implementations may use a different floating-point representation or endianness. The `static_assert` tests to ensure this union matches the implementation.

Listing 2-9 shows a `struct n` that contains a member `type` and a union `u` that itself contains four members: `inode`, `fnode`, `dnode`, and `ldnode`.

```
enum node_type {
  integer_type,
  float_type,
  double_float_type,
```

```
  long_double_type
};

struct node {
  enum node_type type;
  union {
    int inode;
    float fnode;
    double dnode;
    long double ldnode;
  } u;
} n;

n.type = double_type;
n.u.dnode = 3.14;
```

Listing 2-9: Saving memory with a union

This structure might be used in a tree, a graph, or some other data structure that contains differently typed nodes. The `type` member might contain a value between 0 and 3, which indicates the type of the value stored in the structure. It is declared directly in the `struct n` because it is common to all nodes.

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 2-9, the `dnode` member is referenced as `n.u.dnode`. Code that uses this union will typically check the type of the node by examining the value stored in `n.type` and then access the value using `n.u.inode`, `n.u.fnode`, `n.u.dnode`, or `n.u.ldnode`, depending on the value stored in `n.type`. Without the union, each node would contain separate storage for all four data types. The use of a union allows the same storage to be used for all union members. On the x86-64 GCC version 13.2 compiler, using a union saved 16 bytes per node.

Unions are commonly used to describe network or device protocols in cases where you do not know in advance which protocol will be used.

Tags

Tags are a special naming mechanism for structures, unions, and enumerations. For example, the identifier `s` 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 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;
};
```

C requires use of tag types (`struct`, `union`, or `enum`) to include the tag name. The compiler will emit a diagnostic if you do not use `struct tnode` in the declaration of the `left` and `right` pointers. Consequently, you must declare a tag for the structure.

You can create an alias for the structure using a `typedef`:

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

The declaration of the `left` and `right` pointers must still use the tag name because the `typedef` name is not introduced until after the `struct` declaration is complete. You can use the same name for the tag and the `typedef`, but a common idiom is to name the tag something ugly such as `tnode_` to encourage programmers to use the type name:

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

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;
};
```

Type definitions can improve code readability beyond their use with structures. For example, given the following type definition

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

these declarations of the `signal` function all specify the same type:

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

The last two declarations are clearly easier to read.

Type Qualifiers

All the types examined so far have been unqualified types. You can qualify types 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 structure and union members.

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

const

Objects declared with the `const` qualifier (`const`-qualified types) are not assignable but can have constant initializers. This means the compiler can place objects with `const`-qualified types in read-only memory, 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 it's 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 referenced 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
```

Be careful not to pass a `const`-qualified pointer to a function that modifies the object.

volatile

Objects are given a `volatile`-qualified type to allow for processes that are *extrinsic* to the compiler. The values stored in these objects may change without the knowledge of the compiler, or a write may synchronize externally. 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 without its knowledge 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.

A `volatile`-qualified type can be used to access memory-mapped registers, which are accessed through an address just like any other memory. Input/output (I/O) devices often have memory-mapped registers, where you can write to, or read from, a specific address to set or retrieve information or data. Each read and write operation must occur, even if the compiler can see no reason for it. Declaring an object as `volatile` ensures that each read or write of that object at runtime occurs the same number of times and in the same order as indicated by the source code. For example, if `port` is defined as a `volatile`-qualified `int`, the compiler must generate instructions to read the value from `port` and then write this value back to `port` in the assignment:

```
port = port;
```

Without the `volatile` qualification, the compiler would see this as a no-op (a programming statement that does nothing) and might eliminate both the read and the write. Reads and writes of volatile memory are touched exactly once. A volatile operation cannot be eliminated or fused with a subsequent one, even if the compiler believes it's useless. A volatile operation cannot be speculated, even if the compiler can undo or otherwise make that speculation benign.

Objects with `volatile`-qualified types are used when a compiler is unaware of external interactions. For example, `volatile`-qualified types can be used for memory shared with untrusted code to avoid time-of-check to time-of-use (ToCToU) vulnerabilities. Such types are used to access objects from a signal handler and with `setjmp/longjmp` (refer to the C standard for information on signal handlers and `setjmp/longjmp`). Unlike Java and other programming languages, `volatile`-qualified types should not be used in C for synchronization between threads.

Memory-mapped I/O ports are modeled by a `static volatile`–qualified objects model. Memory-mapped input ports such as a real-time clock are modeled by `static const volatile`–qualified objects. A `const volatile`–qualified object models a variable that can be altered by a separate thread. The meaning of the `static` storage-class specifier is explained later in this chapter.

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 whether 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.

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 its scope is the entire text file in which it appears as well as any included files.

If the declaration appears inside a block or within the list of parameters, it has *block scope*, meaning that the identifier is accessible only from within the block. The identifiers for `a` and `b` from Listing 2-4 have block scope and can be referenced only from 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 they identify a statement in the same function to which control may be transferred. (Chapter 5 covers labels and control transfer.)

Scopes also can be *nested*, with *inner* and *outer* scopes. For example, you can define 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 any outer scope that encompasses it.

If you declare the same identifier in both an inner scope and an outer scope, the identifier declared in the outer scope is *hidden* (also known as *shadowed*) by the identifier declared in the inner scope. Referencing the

identifier from the inner scope will refer to the object in the inner scope; the object in 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 2-10 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 2-10: Identifiers declared in inner scopes hiding identifiers declared in outer scopes

There is nothing wrong with this code, provided the comments accurately describe your intent. However, it's better 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 with automatic storage duration are declared within a block or as a function parameter. The lifetime of these objects starts when the block in which they're declared begins execution and ends when execution of this block completes. If the block is entered recursively, a new object is created each time the block is entered, and each object has 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 period for which the object exists.*

Objects declared in file scope have *static* storage duration. The lifetime of those objects is the entire execution of the program, and their stored value is initialized prior to program startup.

Thread storage duration is used in concurrent programming and is not covered by this book. *Allocated* storage duration involves dynamically

allocated memory and is discussed in Chapter 6. Finally, as described in the next section, a storage-class specifier can determine or influence storage duration.

Storage Class

You can specify the storage class of an object or functions using storage-class specifiers. For C23, these include `auto`, `constexpr`, `extern`, `register`, `static`, `thread_local`, and `typedef`. The `constexpr` storage-class specifier is new in C23, and the `auto` storage-class specifier is significantly changed.

Storage-class specifiers specify various properties of identifiers and declared features:

- Storage duration: `static` in block scope, `thread_local`, `auto`, and `register`
- Linkage: `extern`, `static` and `constexpr` in file scope, and `typedef`
- Value: `constexpr`
- Type: `typedef`

With a few exceptions, only one storage-class specifier is allowed for each declaration. For example, `auto` may appear with all the others except `typedef`.

static

The `static` storage-class specifier is used to specify both storage duration and linkage.

File scope identifiers that are specified as `static` or `constexpr`, or functions specified as static, have `internal` linkage.

You can also declare a variable with block scope to have static storage duration by using the storage-class specifier `static`, as shown in the counting example in Listing 2-11. These objects persist after the function has exited.

```
#include <stdio.h>
#include <stdlib.h>

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

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

Listing 2-11: A counting example

This program outputs `1 2 3 4 5`. The static variable `counter` is initialized to 0 once at program startup and incremented 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's good software engineering practice to limit the scope of an object whenever possible.

extern

The `extern` specifier specifies static storage duration and external linkage. It can be used with function and object declarations in both file and block scope (but not function parameter lists). If `extern` is specified for the redeclaration of an identifier that has already been declared with internal linkage, the linkage remains internal. Otherwise (if the prior declaration was external, has no linkage, or is not in scope), the linkage is external.

thread_local

An object whose identifier is declared with the storage-class specifier `thread_local` has *thread storage duration*. Its initializer is evaluated prior to program execution, its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized with the previously determined value when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. (The topic of threading is beyond the scope of this book.)

constexpr

A scalar object declared with the `constexpr` storage-class specifier is a constant and has its value permanently fixed at translation time. The `constexpr` storage-class specifier may appear with `auto`, `register`, or `static`. If not already present, a `const` qualification is implicitly added to the object's type. The resulting object cannot be modified at runtime in any way. The compiler can then use this value in any other constant expression.

Additionally, the constant expression that is used for the initializer of such a constant is checked at compile time. Before the introduction of `constexpr` in C23, a very large object constant might be declared as follows:

```
static size_t const BFO = 0x100000000;
```

The initializer may or may not fit into `size_t`; a diagnostic is not required. In C23, this same object can be declared using `constexpr` as follows:

```
constexpr size_t BFO = 0x100000000;
```

Now, a diagnostic is required on implementations where `size_t` has a width of 32 or less.

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;
}
```

Arithmetic constant expressions are allowed in initializers. Constant values are literal constants (for example, `1`, `'a'`, or `0xFF`), `enum` members, a scalar object declared with the `constexpr` storage-class specifier, and the result of operators such as `alignof` or `sizeof` (provided the operand does not have a variable-length array type). Unfortunately, `const`-qualified objects are not constant values. Starting with C23, an implementation may accept other forms of constant expressions; it is implementation defined whether they are integer constant expressions.

register

The `register` storage-class specifier suggests that access to an object be as fast as possible. The extent to which such suggestions are effective is implementation defined. Frequently, compilers can make better decisions about register allocation and ignore these programmer suggestions. The `register` storage class can be used only for an object that never has its address taken. A compiler can treat any register declaration simply as an `auto` declaration. However, whether addressable storage is used, the address of any part of an object declared with a storage-class specifier register cannot be computed, either explicitly by use of the unary `&` operator or implicitly by converting an array name to a pointer.

typedef

The `typedef` storage-class specifier defines an identifier to be a `typedef` name that denotes the type specified for the identifier. The `typedef` storage-class specifier was discussed earlier in the "Type Definitions" box.

auto

Prior to C23, the `auto` specifier was allowed only for objects declared at block scope (except function parameter lists). It indicates automatic storage duration and no linkage, which are the defaults for these kinds of declarations.

C23 introduced type inference into the C language by expanding the definition of the existing `auto` storage-class specifier. Prior to C23, declaring a variable in C requires the user to name a type. However, when the declaration includes an initializer, the type can be derived directly from the type

of the expression used to initialize the variable. This has been a C++ feature since 2011.

The auto storage duration class specifier has similar behavior to C++ in that it allows the type to be inferred from the type of the assignment value. Take the following file scope definitions, for example:

```
static auto a = 3;
auto p = &a;
```

Because the integer literal 3 has an implicit type of int, these declarations are interpreted as if they had been written as:

```
static int a = 3;
int * p = &a;
```

Effectively, a is an int, and p is an int *. Type inference is extremely useful when implementing or invoking type-generic macros, as we'll see in Chapter 9.

typeof Operators

C23 introduced the typeof operators including typeof and typeof_unqual. The typeof operators can operate on an expression or a type name and yield the type of their operand. If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluated.

The typeof operators and the auto storage duration class specifier both perform automatic type inference. They can both be used to determine the type of expression.

The auto storage duration class specifier is commonly used to declare initialized variables where the type can be inferred from the initial value. However, to form a derived type, you must use the typeof operator:

```
_Atomic(typeof(x)*) apx = &x;
```

The auto storage duration class specifier cannot be used with _Generic (described in Chapter 9) and typedef (described later in this chapter).

The result of the typeof_unqual operator is the nonatomic, unqualified version of the type that would result from the typeof operator. The typeof operator preserves all qualifiers.

The typeof operator is like the sizeof operator, which executes the expression in an unevaluated context to understand the final type. You can use the typeof operator anywhere you can use a type name. The following example illustrates the use of both typeof operators:

```
#include <stdlib.h>
const _Atomic int asi = 0;
const int si = 1;
const char* const beatles[] = {
```

```
    "John",
    "Paul",
    "George",
    "Ringo"
};

❶ typeof_unqual(si) main() {
  ❷ typeof_unqual(asi) plain_si;
  ❸ typeof(_Atomic ❹ typeof(si)) atomic_si;
  ❺ typeof(beatles) beatles_array;
  ❻ typeof_unqual(beatles) beatles2_array;
    return EXIT_SUCCESS;
}
```

At the first use of the `typeof_unqual` operator ❶, the operand is `si`, which has the type `const int`. The `typeof_unqual` operator strips the `const` qualifier, resulting in just plain `int`. This use of the `typeof_unqual` operator is illustrative and not meant for production code. The `typeof_unqual` operator is used again on operand `asi` ❷, which has the type `const _Atomic int`. All qualifiers are once again stripped, resulting in a plain `int`. The operand to the `typeof` specifier at ❸ includes another `typeof` specifier. If the `typeof` operand is itself a `typeof` specifier, the operand is evaluated before evaluating the current `typeof` operator. This evaluation happens recursively until a `typeof` specifier is no longer the operand. In this case, the `typeof` specifier at ❸ does nothing and can be omitted. The `typeof` operator at ❹ is evaluated before the `typeof` operator at ❸ and returns `const int`. The `typeof` operator at ❸ is now evaluated and returns `const _Atomic int`. This happens recursively until a `typeof` operator is no longer the operand. The `typeof` operator at ❺ returns a `const` array of four `const char` pointers. The `typeof_unqual` operator at ❻ strips the qualifier and returns an array of four `const char` pointers. The qualifiers, in this case, are only stripped from the array and not the element types the array contains.

The following `main` function is equivalent but doesn't use `typeof` operators:

```
int main() {
  int plain_si;
  const _Atomic int atomic_si;
  const char* const beatles_array[4];
  const char* beatles2_array[4];
  return EXIT_SUCCESS;
}
```

You can use the `typeof` operator to refer to a macro parameter to construct objects with the required types without specifying the type names explicitly as macro arguments.

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. Central processing units (CPUs) may have different behavior when accessing aligned data (for example, where the data address is a multiple of the data size) versus unaligned data.

Some machine instructions can perform multibyte accesses on nonword boundaries, but with a performance penalty. A *word* is a natural, fixed-sized unit of data handled by the instruction set or the hardware of the processor. 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. 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 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 non-negative 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 a declaration. Listing 2-12 uses the alignment specifier to ensure that `good_buff` is properly aligned (`bad_buff` may have incorrect alignment for member-access expressions).

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

int main() {
  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;
  struct S *good_s_ptr = (struct S *)good_buff; // correct alignment
  good_s_ptr->i = 12;
  return good_s_ptr->i;
}
```

Listing 2-12: Use of the `alignas` keyword

Although `good_buff` has proper alignment to be accessed through an lvalue of type `struct S`, this program still has undefined behavior. This undefined behavior stems from the underlying object `good_buff` being

declared as an array of objects of type `unsigned char` and being accessed through an lvalue of a different type. The cast to `(struct S *)`, like any pointer cast, doesn't change the effective type of the storage allocated to each array. Because it is an established practice to use areas of character type for low-level storage management, I coauthored a paper to make such code conforming in a future revision of the C standard (Seacord et al. 2024).

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.

Alignment of dynamically allocated memory is covered in Chapter 6.

Variably Modified Types

Variably modified types (VMTs) define a base type and an extent (number of elements), which is determined at runtime. VMTs are a mandatory feature of C23.

VMTs can be used as function parameters. Remember from earlier in this chapter that, when used in an expression, an array is converted to a pointer to the first element of the array. This means that we must add an explicit parameter to specify the size of the array—for example, the `n` parameter in the signature for `memset`:

```
void *memset(void *s, int c, size_t n);
```

When you call such a function, `n` should accurately represent the size of the array referenced by `s`. Undefined behavior results if this size is larger than the array.

When declaring a function to take an array as an argument that specifies a size, we must declare the size of the array before referencing the size in the array declaration. We could, for example, modify the signature for the `memset` function as follows to take the number of elements `n` and an array of at least `n` elements:

```
void *memset_vmt(size_t n, char s[n], int c);
```

For arrays of character type, the number of elements is equal to the size. In this function signature, `s[n]` is a variably modified type because `s[n]` depends on the runtime value of `n`.

We've changed the order of the parameters so that the size parameter `n` is declared before we use it in the array declaration. The array argument `s` is still adjusted to a pointer, and no storage is allocated because of this declaration (except for the pointer itself). When calling this function, you must declare the actual storage for the array referenced by `s` and ensure that `n` is a valid size for it. Just like a non-VMT parameter, the actual array storage may be a fixed-size array, variable-length array (covered in Chapter 6), or dynamically allocated storage.

VMTs can generalize your functions, making them more useful. For example, the `matrix_sum` function sums all the values in a two-dimensional array. The following version of this function accepts a matrix with a fixed column size:

```
int matrix_sum(size_t rows, int m[][4]);
```

When passing a multidimensional array to a function, the number of elements in the initial dimension of the array (the rows) is lost and needs to be passed in as an argument. The `rows` parameter provides this information in this example. You can call this function to sum the values of any matrix with exactly four columns, as shown in Listing 2-13.

```
int main(void) {
  int m1[5][4];
  int m2[100][4];
  int m3[2][4];
  printf("%d.\n", matrix_sum(5, m1));
  printf("%d.\n", matrix_sum(100, m2));
  printf("%d.\n", matrix_sum(2, m3));
}
```

Listing 2-13: Summing matrices with four columns

This is fine until you need to sum the values of a matrix that does not have four columns. For example, changing `m3` to have five columns would result in a warning such as this:

```
warning: incompatible pointer types passing 'int [2][5]' to parameter of type 'int (*)[4]'
```

To accept this argument, you would have to write a new function with a signature that matches the new dimensions of the multidimensional array. The problem with this approach, then, is that it fails to generalize sufficiently.

Instead of doing that, we can rewrite the `matrix_sum` function to use a VMT, as shown in Listing 2-14. This change allows us to call `matrix_sum` with matrices of any dimension.

```
int matrix_sum(size_t rows, size_t cols, int m[rows][cols]) {
  int total = 0;

  for (size_t r = 0; r < rows; r++)
    for (size_t c = 0; c < cols; c++)
      total += ❶ m[r][c];
  return total;
}
```

Listing 2-14: Using a VMT as a function parameter

The compiler performs the matrix indexing ❶. Without VMTs, this would require either manual indexing or double indirection, which are both error prone.

Again, no storage is allocated by either the function declaration or the function definition. As with a non-VMT parameter, you need to allocate the storage for the matrix separately, and its dimensions must match those passed to the function as the `rows` and `cols` arguments. Failing to do so can result in undefined behavior.

Attributes

Starting with C23, you can use *attributes* to associate additional information with a declaration, statement, or type. This information can be used by the implementation to improve diagnostics, improve performance, or modify the behavior of the program in other ways. A comma-delimited list of zero or more attributes is specified within a pair of double square brackets, for example, `[[foo]]` or `[[foo, bar]]`.

Declarations attributes are specified in two ways. If the attribute specifier is at the start of a declaration, the attributes are applied to all declarations in the declaration group. Otherwise, the attributes are applied to the declaration to the immediate left of the attribute specifier. For example, in the following declaration group, the `foo` attribute is applied to `x`, `y`, and `z`:

```
[[foo]] int x, y, *z;
```

While in the second declaration group, the `foo` and `bar` attributes are applied only to `b`:

```
int a, b [[foo, bar]], *c;
```

C23 defines several attributes that apply to declarations, such as `nodiscard` and `deprecated`. The `nodiscard` attribute is used with function declarations to denote that the value returned by the function is expected to be used within an expression or initializer. The `deprecated` attribute is used with the declaration of a function or a type to denote that use of the function or type should be diagnosed as discouraged.

In addition to standard attributes, the implementation may provide nonportable attributes. Such attributes are also specified within double square brackets, but they include a vendor prefix to distinguish between attributes from different vendors. For example, the `[[clang::overloadable]]` attribute is used on a function declaration to specify that it can use C++-style function overloading in C, and the `[[gnu::packed]]` attribute is used on a structure declaration to specify that the member declarations of the structure should avoid using padding between member declarations whenever possible for a more space-efficient layout. Vendors typically use their own prefixes, and they may use whatever prefixes they choose. For example, Clang implements many attributes with the `gnu` prefix for improved

compatibility with GCC. Your compiler should ignore unknown attributes, although they may still be diagnosed so you know that the attribute has no effect. Refer to your compiler's documentation for the full list of supported attributes.

EXERCISES

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.
3. Repair the following program by the appropriate use of the `volatile` type qualifier:

```
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

int main() {
  int foo = 5;
  if (setjmp(buf) != 2) {
    if (foo != 5) { puts("hi"); longjmp(buf, 2); }
    foo = 6;
    longjmp(buf, 1);
  }
  return EXIT_SUCCESS;
}
```

Hint: The problem may only manifest for optimized builds.

Summary

In this chapter, you learned about objects and functions and how they differ. You learned how to declare objects 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 further explore 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.