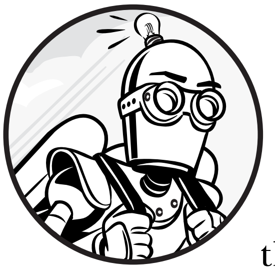
2

TYPES

Hardin once said, "To succeed, planning alone is insufficient. One must improvise as well." I'll improvise.
—Isaac Asimov, Foundation



As discussed in Chapter 1, a type declares how an object will be interpreted and used by the compiler. Every object in a C++ program has a type. This chapter begins with a thorough discussion of fundamental types and then introduces user-defined types. Along the way, you'll learn about several control flow structures.

Fundamental Types

Fundamental types are the most basic types of object and include integer, floating-point, character, Boolean, `byte`, `size_t`, and `void`. Some refer to fundamental types as *primitive* or *built-in* types because they're part of the core language and almost always available to you. These types will work on any platform, but their features, such as size and memory layout, depend on implementation.

Fundamental types strike a balance. On one hand, they try to map a direct relationship from C++ construct to computer hardware; on the other hand, they simplify writing cross-platform code by allowing a programmer to write code once that works on many platforms. The sections that follow provide additional detail about these fundamental types.

Integer Types

Integer types store whole numbers: those that you can write without a fractional component. The four sizes of integer types are *short int*, *int*, *long int*, and *long long int*. Each can be either signed or unsigned. A *signed* variable can be positive, negative, or zero, and an *unsigned* variable must be non-negative.

Integer types are signed and `int` by default, which means you can use the following shorthand notations in your programs: `short`, `long`, and `long long` rather than `short int`, `long int`, and `long long int`. Table 2-1 lists all available C++ integer types, whether each is signed or unsigned, the size of each (in bytes) across platforms, as well as the format specifier for each.

Table 2-1: Integer Types, Sizes, and Format Specifiers

		Size in bytes				
		32-bit OS		64-bit OS		`printf` format specifier
Type	**Signed**	**Windows**	**Linux/Mac**	**Windows**	**Linux/Mac**	
`short`	Yes	2	2	2	2	`%hd`
`unsigned short`	No	2	2	2	2	`%hu`
`int`	Yes	4	4	4	4	`%d`
`unsigned int`	No	4	4	4	4	`%u`
`long`	Yes	4	4	4	8	`%ld`
`unsigned long`	No	4	4	4	8	`%lu`
`long long`	Yes	8	8	8	8	`%lld`
`unsigned long long`	No	8	8	8	8	`%llu`

Notice that the integer type sizes vary across platforms: 64-bit Windows and Linux/Mac have different sizes for a `long` integer (4 and 8, respectively).

Usually, a compiler will warn you of a mismatch between format specifier and integer type. But you must ensure that the format specifiers are correct when you're using them in `printf` statements. Format specifiers appear here so you can print integers to console in examples to follow.

NOTE *If you want to enforce guaranteed integer sizes, you can use integer types in the `<cstdint>` library. For example, if you need a signed integer with exactly 8, 16, 32, or 64 bits, you could use `int8_t`, `int16_t`, `int32_t`, or `int64_t`. You'll find options for the fastest, smallest, maximum, signed, and unsigned integer types to meet your requirements. But because this header is not always available in every platform, you should only use `cstdint` types when there is no other alternative.*

A *literal* is a hardcoded value in a program. You can use one of four hardcoded, *integer literal* representations:

`binary` Uses the prefix `0b`

`octal` Uses the prefix `0`

`decimal` This is the default

`hexadecimal` Uses the prefix `0x`

These are four different ways of writing the same set of whole numbers. For example, Listing 2-1 shows how you might assign several integer variables with integer literals using each of the non-decimal representations.

```
#include <cstdio>

int main() {
  unsigned short a = 0b10101010; ❶
  printf("%hu\n", a);
  int b = 0123; ❷
  printf("%d\n", b);
  unsigned long long d = 0xFFFFFFFFFFFFFFFF; ❸
  printf("%llu\n", d);
}
--------------------------------------------------------------------------
170 ❶
83 ❷
18446744073709551615 ❸
```

Listing 2-1: A program that assigns several integer variables and prints them with the appropriate format specifier

This program uses each of the non-decimal integer representations (binary ❶, octal ❷, and hexadecimal ❸) and prints each with `printf` using the appropriate format specifier listed in Table 2-1. The output from each `printf` appears as a following comment.

NOTE *Integer literals can contain any number of single quotes (`'`) for readability. These are completely ignored by the compiler. For example, `1000000` and `1'000'000` are both integer literals equal to one million.*

Sometimes, it's useful to print an unsigned integer in its hexadecimal representation or (rarely) its octal representation. You can use the `printf` specifiers `%x` and `%o` for these purposes, respectively, as shown in Listing 2-2.

```
#include <cstdio>

int main() {
  unsigned int a = 3669732608;
  printf("Yabba %x❶!\n", a);
  unsigned int b = 69;
  printf("There are %u❷,%o❸ leaves here.\n", b❹, b❺);
}
```

```
Yabba dabbad00❶!
There are 69❷,105❸ leaves here.
```

Listing 2-2: A program that uses octal and hexadecimal representations of unsigned integers

The hexadecimal representation of the decimal `3669732608` is `dabbad00`, which appears in the first line of output as a result of the hexadecimal format specifier `%x` ❶. The decimal 69 is 105 in octal. The format specifiers for unsigned integer `%u` ❷ and octal integer `%o` ❸ correspond with the arguments at ❹ and ❺, respectively. The `printf` statement substitutes these quantities ❷❸ into the format string, yielding the message `There are 69,105 leaves in here`.

WARNING *The octal prefix is a holdover from the B language, back in the days of the PDP-8 computer and ubiquitous octal literals. C, and by extension C++, continues the dubious tradition. You must be careful, for example, when you're hardcoding ZIP codes:*

```
int mit_zip_code = 02139; // Won't compile
```

Eliminate leading zeros on decimal literals; otherwise, they'll cease to be decimal. This line doesn't compile because 9 is not an octal digit.

By default, an integer literal's type is one of the following: `int`, `long`, or `long long`. An integer literal's type is the smallest of these three types that fits. (This is defined by the language and will be enforced by the compiler.)

If you want more control, you can supply *suffixes* to an integer literal to specify its type (suffixes are case insensitive, so you can choose the style you like best):

- The `unsigned` suffix `u` or `U`
- The `long` suffix `l` or `L`
- The `long long` suffix `ll` or `LL`

You can combine the `unsigned` suffix with either the `long` or the `long long` suffix to specify signed-ness and size. Table 2-2 shows the possible types that a suffix combination can take. Allowed types are shown with a check mark (✓). For binary, octal, and hexadecimal literals, you can omit the `u` or `U` suffix. These are depicted with an asterisk (*).

Table 2-2: Integer Suffixes

Type	(none)	`l/L`	`ll/LL`	`u/U`	`ul/UL`	`ull/ULL`
`int`	✓					
`long`	✓	✓				
`long long`	✓	✓	✓			
`unsigned int`	*			✓		
`unsigned long`	*	*		✓	✓	
`unsigned long long`	*	*	*	✓	✓	✓

The smallest allowed type that still fits the integer literal is the resulting type. This means that among all types allowed for a particular integer, the smallest type will apply. For example, the integer literal `112114` could be an `int`, a `long`, or a `long long`. Since an `int` can store 112114, the resulting integer literal is an `int`. If you really want, say, a `long`, you can instead specify `112114L` (or `112114l`).

Floating-Point Types

Floating-point types store approximations of real numbers (which in our case can be defined as any number that has a decimal point and a fractional part, such as 0.33333 or 98.6). Although it's not possible to represent an arbitrary real number exactly in computer memory, it's possible to store an approximation. If this seems hard to believe, just think of a number like π, which has infinitely many digits. With finite computer memory, how could you possibly represent infinitely many digits?

As with all types, floating-point types take up a finite amount of memory, which is called the type's *precision*. The more precision a floating-point type has, the more accurate it will be at approximating a real number. C++ offers three levels of precision for approximations:

`float`	single precision
`double`	double precision
`long double`	extended precision

As with integer types, each floating-point representation depends on implementation. This section won't go into detail about floating-point types, but note that there is substantial nuance involved in these implementations.

On major desktop operating systems, the `float` level usually has 4 bytes of precision. The `double` and `long double` levels usually have 8 bytes of precision (*double precision*).

Most users not involved in scientific computing applications can safely ignore the details of floating-point representation. In such cases, a good general rule is to use a `double`.

NOTE *For those who cannot safely ignore the details, look at the floating-point specification relevant to your hardware platform. The predominant implementation of floating-point storage and arithmetic is outlined in* The IEEE Standard for Floating-Point Arithmetic, IEEE 754.

Floating-Point Literals

Floating-point literals are double precision by default. If you need single precision, use an `f` or `F` suffix; for extended precision, use `l` or `L`, as shown here:

```
float a = 0.1F;
double b = 0.2;
long double c = 0.3L;
```

You can also use scientific notation in literals:

```
double plancks_constant = 6.62607004❶e-34❷;
```

No spaces are permitted between the *significand* (the base ❶) and the *suffix* (the exponential portion ❷).

Floating-Point Format Specifiers

The format specifier `%f` displays a `float` with decimal digits, whereas `%e` displays the same number in scientific notation. You can let `printf` decide which of these two to use with the `%g` format specifier, which selects the more compact of `%e` or `%f`.

For `double`, you simply prepend an `l` (lowercase *L*) to the desired specifier; for `long double`, prepend an `L`. For example, if you wanted a `double` with decimal digits, you would specify `%lf`, `%le`, or `%lg`; for a long double, you would specify `%Lf`, `%Le`, or `%Lg`.

Consider Listing 2-3, which explores the different options for printing floating points.

```
#include <cstdio>

int main() {
  double an = 6.0221409e23; ❶
  printf("Avogadro's Number:  %le❷ %lf❸ %lg❹\n", an, an, an);
  float hp = 9.75; ❺
  printf("Hogwarts' Platform: %e %f %g\n", hp, hp, hp);
}
--------------------------------------------------------------------------------
Avogadro's Number:  6.022141e+23❷ 602214090000000006225920.000000❸
6.02214e+23❹
Hogwarts' Platform: 9.750000e+00 9.750000 9.75
```

Listing 2-3: A program printing several floating points

This program declares a `double` called `an` ❶. The format specifier `%le` ❷ gives you scientific notation `6.022141e-23`, and `%lf` ❸ gives the decimal representation `602214090000000006225920.000000`. The `%lg` ❹ specifier chose the scientific notation `6.02214e-23`. The `float` called `hp` ❺ produces similar `printf` output using the `%e` and `%f` specifiers. But the format specifier `%g` decided to provide the decimal representation 9.75 rather than scientific notation.

As a general rule, use `%g` to print floating-point types.

NOTE *In practice, you can omit the* `l` *prefix on the format specifiers for* `double`*, because* `printf` *promotes* `float` *arguments to* `double` *precision.*

Character Types

Character types store human language data. The six character types are:

`char` The default type, always 1 byte. May or may not be signed. (Example: ASCII.)

`char16_t` Used for 2-byte character sets. (Example: UTF-16.)

`char32_t` Used for 4-byte character sets. (Example: UTF-32.)

`signed char` Same as `char` but guaranteed to be signed.

`unsigned char` Same as `char` but guaranteed to be unsigned.

`wchar_t` Large enough to contain the largest character of the implementation's locale. (Example: Unicode.)

The character types `char`, `signed char`, and `unsigned char` are called *narrow characters*, whereas `char16_t`, `char32_t`, and `wchar_t` are called *wide characters* due to their relative storage requirements.

Character Literals

A *character literal* is a single, constant character. Single quotation marks (`' '`) surround all characters. If the character is any type but `char`, you must also provide a prefix: `L` for `wchar_t`, `u` for `char16_t`, and `U` for `char32_t`. For example, `'J'` declares a `char` literal and `L'J'` declares a `wchar_t`.

Escape Sequences

Some characters don't display on the screen. Instead, they force the display to do things like move the cursor to the left side of the screen (carriage return) or move the cursor down one line (newline). Other characters can display onscreen, but they're part of the C++ language syntax, such as single or double quotes, so you must use them very carefully. To put these characters into a `char`, you use the *escape sequences*, as listed in Table 2-3.

Table 2-3: Reserved Characters and Their Escape Sequences

Value	Escape sequence
Newline	`\n`
Tab (horizontal)	`\t`
Tab (vertical)	`\v`
Backspace	`\b`
Carriage return	`\r`
Form feed	`\f`
Alert	`\a`
Backslash	`\\`
Question mark	`?` or `\?`
Single quote	`\'`
Double quote	`\"`
The null character	`\0`

Unicode Escape Characters

You can specify Unicode character literals using the *universal character names*, and you can form a universal character name in one of two ways: the prefix `\u` followed by a 4-digit Unicode code point or the prefix `\U` followed by an 8-digit Unicode code point. For example, you can represent the `A` character as `'\u0041'` and represent the beer mug character 🍺 as `U'\U0001F37A'`.

Format Specifiers

The `printf` format specifier for `char` is `%c`. The `wchar_t` format specifier is `%lc`.

Listing 2-4 initializes two character literals, `x` and `y`. You use these variables to build a `printf` call.

```
#include <cstdio>

int main() {
  char x = 'M';
  wchar_t y = L'Z';
  printf("Windows binaries start with %c%lc.\n", x, y);
}
```

```
Windows binaries start with MZ.
```

Listing 2-4: A program that assigns several character-typed variables and prints them

This program outputs *Windows binaries start with MZ*. Even though the *M* is a narrow character `char` and the *Z* is a wide character, `printf` works because the program uses the correct format specifiers.

NOTE *The first two bytes of all Windows binaries are the characters* M *and* Z*, an homage to Mark Zbikowski, the designer of the MS-DOS executable binary file format.*

Boolean Types

Boolean types have two states: true and false. The sole Boolean type is `bool`. Integer types and the `bool` types convert readily: the `true` state converts to 1, and `false` converts to 0. Any non-zero integer converts to true, and 0 converts to `false`.

Boolean Literals

To initialize Boolean types, you use two Boolean literals, `true` and `false`.

Format Specifiers

There is no format specifier for `bool`, but you can use the `int` format specifier `%d` within `printf` to yield a `1` for true and a `0` for false. The reason is that `printf` promotes any integral value smaller than an `int` to an `int`. Listing 2-5 illustrates how to declare a Boolean variable and inspect its value.

```
#include <cstdio>

int main() {
  bool b1 = true;  ❶ // b1 is true
  bool b2 = false; ❷ // b2 is false
  printf("%d %d\n", b1, b2); ❸
}
--------------------------------------------------------------------------
1 0 ❸
```

Listing 2-5: Printing `bool` variables with a `printf` statement

You initialize `b1` to `true` ❶ and `b2` to `false` ❷. By printing `b1` and `b2` as integers (using `%d` format specifiers), you get 1 for `b1` and 0 for `b2` ❸.

Comparison Operators

Operators are functions that perform computations on *operands*. Operands are simply objects. ("Logical Operators" on page 182 covers a full menu of operators.) In order to have meaningful examples using `bool` types, you'll take a quick look at comparison operators in this section and logical operators in the next.

You can use several operators to build Boolean expressions. Recall that comparison operators take two arguments and return a `bool`. The available operators are equality (`==`), inequality (`!=`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`).

Listing 2-6 shows how you can use these operators to produce Booleans.

```
#include <cstdio>

int main() {
  printf(" 7 ==  7: %d❶\n", 7  ==  7❷);
  printf(" 7 !=  7: %d\n", 7  !=  7);
  printf("10 >  20: %d\n", 10 >  20);
  printf("10 >= 20: %d\n", 10 >= 20);
  printf("10 <  20: %d\n", 10 <  20);
  printf("20 <= 20: %d\n", 20 <= 20);
}
--------------------------------------------------------------------------
 7 ==  7: 1 ❶
 7 !=  7: 0
10 >  20: 0
10 >= 20: 0
10 <  20: 1
20 <= 20: 1
```

Listing 2-6: Using comparison operators

Each comparison produces a Boolean result ❷, and the `printf` statement prints the Boolean as an `int` ❶.

Logical Operators

Logical operators evaluate Boolean logic on `bool` types. You characterize operators by how many operands they take. A *unary operator* takes a single operand, a *binary operator* takes two, a *ternary operator* takes three, and so on. You categorize operators further by describing the types of their operands.

The unary *negation* operator (`!`) takes a single operand and returns its opposite. In other words, `!true` yields `false`, and `!false` yields `true`.

The logical operators AND (`&&`) and OR (`||`) are binary. Logical AND returns `true` only if both of its operands are `true`. Logical OR returns `true` if either or both of its operands are `true`.

NOTE *When you're reading a Boolean expression, the `!` is pronounced "not," as in "a and not b" for the expression `a && !b`.*

Logical operators might seem confusing at first, but they quickly become intuitive. Listing 2-7 showcases the logical operators.

```
#include <cstdio>

int main() {
  bool t = true;
  bool f = false;
  printf("!true: %d\n", !t); ❶
  printf("true  &&  false: %d\n", t &&  f); ❷
  printf("true  && !false: %d\n", t && !f); ❸
  printf("true  ||  false: %d\n", t ||  f); ❹
  printf("false ||  false: %d\n", f ||  f); ❺
}
```

```
!true: 0 ❶
true  &&  false: 0 ❷
true  && !false: 1 ❸
true  ||  false: 1 ❹
false ||  false: 0 ❺
```

Listing 2-7: A program that illustrates the use of logical operators

Here, you see the negation operator ❶, the logical AND operator ❷❸, and the logical OR operator ❹❺.

The std::byte Type

System programmers sometimes work directly with *raw memory*, which is a collection of bits without a type. Employ the `std::byte` type, available in the `<cstddef>` header, in such situations. The `std::byte` type permits bitwise logical operations (which you'll meet in Chapter 7) and little else. Using this type for raw data rather than an integral type can help to avoid common sources of difficult-to-debug programming errors.

Note that unlike most other fundamental types in `<cstddef>`, `std::byte` doesn't have an exact corollary type in the C language (a "C type"). Like C++, C has `char` and `unsigned char`. These types are less safe to use because they support many operations that std::byte doesn't. For example, you can perform arithmetic, like addition (+), on a char but not a std::byte. The odd-looking std:: prefix is called a *namespace*, which you'll meet in "Namespaces" on page 216 (for now, just think of the namespace std:: as part of the type name).

NOTE *There are two schools of thought on how to pronounce* std. *One is to treat it as an initialism, as in "ess-tee-dee," and another is to treat it as an acronym, as in "stood." When referring to a class in the* std *namespace, speakers typically imply the namespace operator* ::. *So you could pronounce* std::byte *as "stood byte" or, if you're not into the whole brevity thing, as "ess-tee-dee colon colon byte."*

The size_t Type

You use the size_t type, also available in the <cstddef> header, to encode size of objects. The size_t objects guarantee that their maximum values are sufficient to represent the maximum size in bytes of all objects. Technically, this means a size_t could take 2 bytes or 200 bytes depending on the implementation. In practice, it's usually identical to an unsigned long long on 64-bit architectures.

NOTE *The type* size_t *is a C type in the* <stddef> *header, but it's identical to the C++ version, which resides in the* std *namespace. Occasionally, you'll see the (technically correct) construction* std::size_t *instead.*

sizeof

The unary operator sizeof takes a type operand and returns the size (in bytes) of that type. The sizeof operator always returns a size_t. For example, sizeof(float) returns the number of bytes of storage a float takes.

Format Specifiers

The usual format specifiers for a size_t are %zd for a decimal representation or %zx for a hexadecimal representation. Listing 2-8 shows how you might check a system for several integer types' sizes.

```
#include <cstddef>
#include <cstdio>

int main() {
  size_t size_c = sizeof(char); ❶
  printf("char: %zd\n", size_c);
  size_t size_s = sizeof(short); ❷
  printf("short: %zd\n", size_s);
  size_t size_i = sizeof(int); ❸
  printf("int: %zd\n", size_i);
  size_t size_l = sizeof(long); ❹
  printf("long: %zd\n", size_l);
  size_t size_ll = sizeof(long long); ❺
  printf("long long: %zd\n", size_ll);
}
```

```
char: 1 ❶
short: 2 ❷
int: 4 ❸
long: 4 ❹
long long: 8 ❺
```

Listing 2-8: A program that prints the sizes in bytes of several integer types. (The output comes from a Windows 10 x64 machine.)

Listing 2-8 evaluates the `sizeof` a `char` ❶, a `short` ❷, an `int` ❸, a `long` ❹, and a `long long` ❺ and prints their sizes using the `%zd` format specifier. Results will vary depending on the operating system. Recall from Table 2-1 that each environment defines its own sizes for the integer types. Pay special attention to the return value of `long` in Listing 2-8; Linux and macOS define 8-byte `long` types.

void

The `void` type has an empty set of values. Because a `void` object cannot hold a value, C++ disallows void objects. You use `void` in special situations, such as the return type for functions that don't return any value. For example, the function `taunt` doesn't return a value, so you declare its return type `void`:

```
#include <cstdio>

void taunt() {
  printf("Hey, laser lips, your mama was a snow blower.");
}
```

In Chapter 3, you'll learn about other special `void` uses.

Arrays

Arrays are sequences of identically typed variables. *Array types* include the contained type and the number of contained elements. You weave this information together in the declaration syntax: the element type precedes square brackets enclosing the array's size.

For example, the following line declares an array of 100 `int` objects:

```
int my_array[100];
```

Array Initialization

There's a shortcut for initializing arrays with values using braces:

```
int array[] = { 1, 2, 3, 4 };
```

You can omit the length of the array because it can be inferred from the number of elements in the braces at compile time.

Accessing Array Elements

You access array elements by using square brackets to enclose the desired index. Array indexing is zero based in C++, so the first element is at index `0`, the tenth element is at index `9`, and so on. Listing 2-9 illustrates reading and writing array elements.

```
#include <cstdio>

int main() {
  int arr[] = { 1, 2, 3, 4 }; ❶
  printf("The third element is %d.\n", arr[2]❷);
  arr[2] = 100; ❸
  printf("The third element is %d.\n", arr[2]❹);
}
--------------------------------------------------------------------------------
The third element is 3. ❷
The third element is 100. ❹
```

Listing 2-9: A program that indexes into an array

This code declares a four-element array named `arr` containing the elements `1`, `2`, `3`, and `4` ❶. On the next line ❷, it prints the third element. It then assigns the third element to 100 ❸, so when it reprints the third element ❹, the value is `100`.

A Nickel Tour of for Loops

A `for` loop lets you repeat (or iterate) the execution of a statement a specified number of times. You can stipulate a starting point and other conditions. The *init statement* executes before the first iteration executes, so you can initialize variables used in the `for` loop. The *conditional* is an expression that is evaluated before each iteration. If it evaluates to `true`, iteration proceeds. If `false`, the `for` loop terminates. The *iteration statement* executes after each iteration, which is useful in situations where you must increment a variable to cover a range of values. The `for` loop syntax is as follows:

```
for(init-statement; conditional; iteration-statement) {
  --snip--
}
```

For example, Listing 2-10 shows you how to use a `for` loop to find the maximum of an array.

```
#include <cstddef>
#include <cstdio>

int main() {
  unsigned long maximum = 0; ❶
  unsigned long values[] = { 10, 50, 20, 40, 0 }; ❷
  for(size_t i=0; i < 5; i++) { ❸
    if (values[i] > maximum❹) maximum = values[i]; ❺
  }
  printf("The maximum value is %lu", maximum); ❻
}
--------------------------------------------------------------------------
The maximum value is 50 ❻
```

Listing 2-10: Finding the maximum value contained in an array

You initialize `maximum` ❶ to the smallest value possible; here that's 0 because it's unsigned. Next, you initialize the array `values` ❷, which you iterate over using the `for` loop ❸. The iterator variable `i` ranges from 0 to 4 inclusive. Within the `for` loop, you access each element of `values` and check whether the element is greater than the current `maximum` ❹. If it is, you set `maximum` to that new value ❺. When the loop is complete, `maximum` will equal the greatest value in the array, which prints the value of `maximum` ❻.

NOTE *If you've programmed C or C++ before, you might be wondering why Listing 2-10 employs* `size_t` *instead of an* `int` *for the type of* `i`*. Consider that* `values` *could theoretically take up the maximum storage allowed. Although* `size_t` *is guaranteed to be able to index any value within it,* `int` *is not. In practice, it makes little difference, but technically* `size_t` *is correct.*

The Range-Based for Loop

In Listing 2-10, you saw how to use the `for` loop at ❸ to iterate over the elements of the array. You can eliminate the iterator variable `i` by using a *range-based for loop*. For certain objects like arrays, `for` understands how to iterate over the range of values within an object. Here's the syntax for a range-based for loop:

```
for(element-type❶ element-name❷ : array-name❸) {
  --snip--
}
```

You declare an iterator variable `element-name` ❷ with type `element-type` ❶. The `element-type` must match the types within the array you're iterating over. This array is called `array-name` ❸.

Listing 2-11 refactors Listing 2-10 with a range-based `for` loop.

```
#include <cstdio>

int main() {
  unsigned long maximum = 0;
  unsigned long values[] = { 10, 50, 20, 40, 0 };
  for(unsigned long value : values❶) {
    if (value❷ > maximum) maximum = value❸;
  }
  printf("The maximum value is %lu.", maximum);
}
```

```
The maximum value is 50.
```

Listing 2-11: Refactoring Listing 2-10 with a range-based `for` *loop*

NOTE *You'll learn about expressions in Chapter 7. For now, think of an expression as some bit of code that produces an effect on your program.*

Listing 2-11 greatly improves Listing 2-10. At a glance, you know that the `for` loop iterates over `values` ❶. Because you've discarded the iterator variable `i`, the body of the `for` loop simplifies nicely; for that reason, you can use each element of values directly ❷❸.

Use range-based `for` loops generously.

Number of Elements in an Array

Use the `sizeof` operator to obtain the total size in bytes of an array. You can use a simple trick to determine the number of elements in an array: divide the size of the array by the size of a single constituent element:

```
short array[] = { 104, 105, 32, 98, 105, 108, 108, 0 };
size_t n_elements = sizeof(array)❶ / sizeof(short)❷;
```

On most systems, `sizeof(array)` ❶ will evaluate to 16 bytes and `sizeof(short)` ❷ will evaluate to 2 bytes. Regardless of the size of a `short`, `n_elements` will always initialize to 8 because the factor will cancel. This evaluation happens at compile time, so there is no runtime cost in evaluating the length of an array in this way.

The `sizeof(x)/sizeof(y)` construction is a bit of a hack, but it's widely used in older code. In Part II, you'll learn other options for storing data that don't require external computation of their lengths. If you really must use an array, you can safely obtain the number of elements using the `std::size` function available in the `<iterator>` header.

NOTE *As an additional benefit,* `std::size` *can be used with any container that exposes a* `size` *method. This includes all the containers in Chapter 13. This is especially useful when writing generic code, a topic you'll explore in Chapter 6. Further, it will refuse to compile if you accidentally pass an unsupported type, like a pointer.*

C-Style Strings

Strings are contiguous blocks of characters. A *C-style string* or *null-terminated string* has a zero-byte appended to its end (a null) to indicate the end of the string. Because array elements are contiguous, you can store strings in arrays of character types.

String Literals

Declare string literals by enclosing text in quotation marks (`""`). Like character literals, string literals support Unicode: just prepend the literal with the appropriate prefix, such as `L`. The following example assigns string literals to the arrays `english` and `chinese`:

```
char english[] = "A book holds a house of gold.";
char16_t chinese[] = u"\u4e66\u4e2d\u81ea\u6709\u9ec4\u91d1\u5c4b";
```

NOTE *Surprise! You've been using string literals all along: the format strings of your* `printf` *statements are string literals.*

This code generates two variables: `english`, which contains `A book holds a house of gold.`, and `chinese`, which contains the Unicode characters for 书中自有黄金屋.

Format Specifier

The format specifier for narrow strings (`char*`) is `%s`. For example, you can incorporate strings into format strings as follows:

```
#include <cstdio>

int main() {
  char house[] = "a house of gold.";
  printf("A book holds %s\n ", house);
}
--------------------------------------------------------------------------------
A book holds a house of gold.
```

NOTE *Printing Unicode to the console is surprisingly complicated. Typically, you need to ensure that the correct* code page *is selected, and this topic is well beyond the scope of this book. If you need to embed Unicode characters into a string literal, look at* `wprintf` *in the* `<cwchar>` *header.*

Consecutive string literals get concatenated together, and any intervening whitespaces or newlines get ignored. So, you can place string literals on multiple lines in your source, and the compiler will treat them as one. For example, you could refactor this example as follows:

```
#include <cstdio>

int main() {
  char house[] = "a "
      "house "
      "of "  "gold.";
  printf("A book holds %s\n ", house);
}
--------------------------------------------------------------------------------
A book holds a house of gold.
```

Usually, such constructions are useful for readability only when you have a long string literal that would span multiple lines in your source code. The generated programs are identical.

ASCII

The *American Standard Code for Information Interchange (ASCII)* table assigns integers to characters. Table 2-4 shows the ASCII table. For each integer value in decimal (0d) and hex (0x), the given control code or printable character is shown.

Table 2-4: The ASCII Table

Control codes			Printable characters								
0d	**0x**	**Code**	**0d**	**0x**	**Char**	**0d**	**0x**	**Char**	**0d**	**0x**	**Char**
0	0	`NULL`	32	20	`SPACE`	64	40	`@`	96	60	`` ` ``
1	1	`SOH`	33	21	`!`	65	41	`A`	97	61	`a`
2	2	`STX`	34	22	`"`	66	42	`B`	98	62	`b`
3	3	`ETX`	35	23	`#`	67	43	`C`	99	63	`c`
4	4	`EOT`	36	24	`$`	68	44	`D`	100	64	`d`
5	5	`ENQ`	37	25	`%`	69	45	`E`	101	65	`e`
6	6	`ACK`	38	26	`&`	70	46	`F`	102	66	`f`
7	7	`BELL`	39	27	`'`	71	47	`G`	103	67	`g`
8	8	`BS`	40	28	`(`	72	48	`H`	104	68	`h`
9	9	`HT`	41	29	`)`	73	49	`I`	105	69	`i`
10	0a	`LF`	42	2a	`*`	74	4a	`J`	106	6a	`j`
11	0b	`VT`	43	2b	`+`	75	4b	`K`	107	6b	`k`
12	0c	`FF`	44	2c	`,`	76	4c	`L`	108	6c	`l`
13	0d	`CR`	45	2d	`-`	77	4d	`M`	109	6d	`m`
14	0e	`SO`	46	2e	`.`	78	4e	`N`	110	6e	`n`
15	0f	`SI`	47	2f	`/`	79	4f	`O`	111	6f	`o`
16	10	`DLE`	48	30	`0`	80	50	`P`	112	70	`p`
17	11	`DC1`	49	31	`1`	81	51	`Q`	113	71	`q`
18	12	`DC2`	50	32	`2`	82	52	`R`	114	72	`r`
19	13	`DC3`	51	33	`3`	83	53	`S`	115	73	`s`
20	14	`DC4`	52	34	`4`	84	54	`T`	116	74	`t`
21	15	`NAK`	53	35	`5`	85	55	`U`	117	75	`u`
22	16	`SYN`	54	36	`6`	86	56	`V`	118	76	`v`
23	17	`ETB`	55	37	`7`	87	57	`W`	119	77	`w`
24	18	`CAN`	56	38	`8`	88	58	`X`	120	78	`x`

(continued)

Table 2-4: The ASCII Table (continued)

Control codes			Printable characters								
0d	**0x**	**Code**	**0d**	**0x**	**Char**	**0d**	**0x**	**Char**	**0d**	**0x**	**Char**
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1a	SUB	58	3a	:	90	5a	Z	122	7a	z
27	1b	ESC	59	3b	;	91	5b	[	123	7b	{
28	1c	FS	60	3c	<	92	5c	\	124	7c	|
29	1d	GS	61	3d	=	93	5d	]	125	7d	}
30	1e	RS	62	3e	>	94	5e	^	126	7e	~
31	1f	US	63	3f	?	95	5f	_	127	7f	DEL

ASCII codes 0 to 31 are the *control code characters* that control devices. These are mostly anachronisms. When the American Standards Association formalized ASCII in the 1960s, modern devices included teletype machines, magnetic tape readers, and dot-matrix printers. Some control codes still in common use are the following:

- 0 (NULL) is used as a string terminator by programming languages.
- 4 (EOT), the end of transmission, terminates shell sessions and PostScript printer communications.
- 7 (BELL) causes a device to make a noise.
- 8 (BS), the backspace, causes the device to erase the last character.
- 9 (HT), the horizontal tab, moves a cursor several spaces to the right.
- 10 (LF), the line feed, is used as the end-of-line marker on most operating systems.
- 13 (CR), the carriage return, is used in combination with LF as the end-of-line marker on Windows systems.
- 26 (SUB), the substitute character/end of file/CTRL-Z, suspends the currently executing interactive process on most operating systems.

The remainder of the ASCII table, codes from 32 to 127, is the printable characters. These represent the English characters, digits, and punctuation.

On most systems, the `char` type's representation is ASCII. Although this relationship is not strictly guaranteed, it is a de facto standard.

Now it's time to combine your knowledge of `char` types, arrays, `for` loops, and the ASCII table. Listing 2-12 shows how to build an array with the letters of the alphabet, print the result, and then convert this array to uppercase and print again.

```
#include <cstdio>

int main() {
  char alphabet[27]; ❶
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 97; ❷
  }
  alphabet[26] = 0; ❸
  printf("%s\n", alphabet); ❹
  for (int i = 0; i<26; i++) {
    alphabet[i] = i + 65; ❺
  }
  printf("%s", alphabet); ❻
}
--------------------------------------------------------------------------
abcdefghijklmnopqrstuvwxyz❹
ABCDEFGHIJKLMNOPQRSTUVWXYZ❻
```

Listing 2-12: Printing the letters of the alphabet in lowercase and uppercase using ASCII

First, you declare a `char` array of length 27 to hold the 26 English letters plus a null terminator ❶. Next, employ a `for` loop to iterate from 0 to 25 using the iterator `i`. The letter *a* has the value 97 in ASCII. By adding 97 to the iterator `i`, you can generate all the lowercase letters in the `alphabet` ❷. To make `alphabet` a null-terminated string, you set `alphabet[26]` to 0 ❸. You then print the result ❹.

Next, you print the uppercase alphabet. The letter *A* has the value 65 in ASCII, so you reset each element of the alphabet accordingly ❺ and invoke `printf` again ❻.

User-Defined Types

User-defined types are types that the user can define. The three broad categories of user-defined types are these:

Enumerations The simplest of the user-defined types. The values that an enumeration can take are restricted to a set of possible values. Enumerations are excellent for modeling categorical concepts.

Classes More fully featured types that give you flexibility to pair data and functions. Classes that only contain data are called plain-old-data classes; you'll learn about them in this section.

Unions A boutique user-defined type. All members share the same memory location. Unions are dangerous and easy to misuse.

Enumeration Types

Declare enumerations using the keywords `enum class` followed by the type name and a listing of the values it can take. These values are arbitrary alphanumeric strings that will represent whatever categories you want to represent. Under the hood, these values are simply integers, but they allow you to write safer, more expressive code by using programmer-defined types rather than integers that could mean anything. For example, Listing 2-13 declares an `enum class` called `Race` that can take one of seven values.

```
enum class Race {
  Dinan,
  Teklan,
  Ivyn,
  Moiran,
  Camite,
  Julian,
  Aidan
};
```

Listing 2-13: An enumeration class containing all the races from Neal Stephenson's Seveneves

To initialize an enumeration variable to a value, use the name of the type followed by two colons `::` and the desired value. For example, here's how to declare the variable `langobard_race` and initialize its value to `Aidan`:

```
Race langobard_race = Race::Aidan;
```

NOTE *Technically, an* `enum class` *is one of two kinds of enumerations: it's called a* scoped enum. *For compatibility with C, C++ also supports an* unscoped enum, *which is declared with* `enum` *rather than* `enum class`. *The major difference is that scoped enums require the enum's type followed by* `::` *to precede the values, whereas unscoped enums don't. Unscoped* `enum` *classes are less safe to use than scoped enums, so shy away from them unless absolutely necessary. They're supported in C++ for mainly historical reasons, especially interoperation with C code. See* Effective Modern C++ *by Scott Meyers, Item 10, for details.*

Switch Statements

The *switch statement* transfers control to one of several statements depending on the value of a *condition*, which evaluates to either an integer or enumeration type. The `switch` keyword denotes a switch statement.

Switch statements provide conditional branching. When a switch statement executes, control transfers to the *case* fitting the condition or to a *default condition* if no case matches the condition expression. The `case` keyword denotes a case, whereas the `default` keyword denotes the default condition.

Somewhat confusingly, execution will continue until the end of the switch statement or the `break` keyword. You'll almost always find a `break` at the end of each condition.

Switch statements have a lot of components. Listing 2-14 shows how they fit together.

```
switch❶(condition❷) {
  case❸ (case-a❹): {
    // Handle case a here
    --snip--
  }❺ break❻;
  case (case-b): {
    // Handle case b here
    --snip--
  } break;
```

```
    // Handle other conditions as desired
    --snip--
  default❼: {
    // Handle the default case here
    --snip--
  }
}
```

Listing 2-14: A sketch of how switch statements fit together

All switch statements begin with the `switch` keyword ❶ followed by the `condition` in parentheses ❷. Each case begins with the `case` keyword ❸ followed by the case's enumeration or integral value ❹. If `condition` ❷ equals `case-a` ❹, for example, the code in the block containing `Handle case a here` will execute. After each statement following a case ❺, you place a `break` keyword ❻. If `condition` matches none of the cases, the `default` case ❼ executes.

NOTE *The braces enclosing each case are optional but highly recommended. Without them, you'll sometimes get surprising behavior.*

Using a Switch Statement with an Enumeration Class

Listing 2-15 uses a switch statement on a `Race` enumeration class to generate a bespoke greeting.

```
#include <cstdio>

enum class Race { ❶
  Dinan,
  Teklan,
  Ivyn,
  Moiran,
  Camite,
  Julian,
  Aidan
};

int main() {
  Race race = Race::Dinan; ❷

  switch(race) { ❸
  case Race::Dinan: { ❹
      printf("You work hard.");
    } break;  ❺
  case Race::Teklan: {
      printf("You are very strong.");
    } break;
  case Race::Ivyn: {
      printf("You are a great leader.");
    } break;
  case Race::Moiran: {
      printf("My, how versatile you are!");
    } break;
  case Race::Camite: {
      printf("You're incredibly helpful.");
    } break;
  case Race::Julian: {
      printf("Anything you want!");
    } break;
  case Race::Aidan: {
      printf("What an enigma.");
    } break;
  default: {
      printf("Error: unknown race!"); ❻
    }
  }
}
--------------------------------------------------------------------------------
You work hard.
```

Listing 2-15: A program that prints a greeting that depends on the `Race` *selected*

The `enum class` ❶ declares the enumeration type `Race`, which you use to initialize `race` to `Dinan` ❷. The switch statement ❸ evaluates the condition `race` to determine which condition to hand control to. Because you hardcoded this to `Dinan` earlier in the code, execution transfers to ❹, which prints `You work hard`. The `break` at ❺ terminates the switch statement.

The `default` condition at ❻ is a safety feature. If someone adds a new `Race` value to the enumeration class, you'll detect that unknown race at runtime and print an error message.

Try setting `race` ❷ to different values. How does the output change?

Plain-Old-Data Classes

Classes are user-defined types that contain data and functions, and they're the heart and soul of C++. The simplest kind of classes are *plain-old-data classes (PODs)*. PODs are simple containers. Think of them as a sort of heterogeneous array of elements of potentially *different* types. Each element of a class is called a *member*.

Every POD begins with the keyword `struct` followed by the POD's desired name. Next, you list the members' types and names. Consider the following `Book` class declaration with four members:

```
struct Book {
  char name[256]; ❶
  int year; ❷
  int pages; ❸
  bool hardcover; ❹
};
```

A single `Book` contains a `char` array called `name` ❶, an `int year` ❷, an `int pages` ❸, and a `bool hardcover` ❹.

You declare POD variables just like any other variables: by type and name. You can then access members of the variable using the dot operator (`.`).

Listing 2-16 uses the Book type.

```
#include <cstdio>

struct Book {
  char name[256];
  int year;
  int pages;
  bool hardcover;
};

int main() {
  Book neuromancer; ❶
  neuromancer.pages = 271; ❷
  printf("Neuromancer has %d pages.", neuromancer.pages); ❸
}
--------------------------------------------------------------------------------
Neuromancer has 271 pages. ❸
```

Listing 2-16: Example using the POD type Book *to read and write members*

First, you declare a Book variable neuromancer ❶. Next, you set the number of pages of neuromancer to 271 using the dot operator (.) ❷. Finally, you print a message and extract the number of pages from neuromancer, again using the dot operator ❸.

NOTE *PODs have some useful low-level features: they're C compatible, you can employ machine instructions that are highly efficient to copy or move them, and they can be efficiently represented in memory.*

C++ guarantees that members will be sequential in memory, although some implementations require members to be aligned along word boundaries, which depend on CPU register length. As a general rule, you should order members from largest to smallest within POD definitions.

Unions

The union is a cousin of the POD that puts all of its members in the same place. You can think of unions as different views or interpretations of a block of memory. They can be useful in some low-level situations, such as when marshalling structures that must be consistent across architectures, dealing with type-checking issues related to C/C++ interoperation, and even when packing bitfields.

Listing 2-17 illustrates how you declare a union: simply use the union keyword instead of struct.

```
union Variant {
  char string[10];
  int integer;
  double floating_point;
};
```

Listing 2-17: An example union

The union Variant can be interpreted as a char[10], an int, or a double. It takes up only as much memory as its largest member (probably string in this case).

You use the dot operator (.) to specify a union's interpretation. Syntactically, this looks like accessing a member of a POD, but it's completely different under the hood.

Because all members of a union are in the same place, you can cause data corruption very easily. Listing 2-18 illustrates the danger.

```
#include <cstdio>

union Variant {
  char string[10];
  int integer;
  double floating_point;
};

int main() {
  Variant v; ❶
  v.integer = 42; ❷
  printf("The ultimate answer: %d\n", v.integer); ❸
  v.floating_point = 2.7182818284; ❹
  printf("Euler's number e:    %f\n", v.floating_point); ❺
  printf("A dumpster fire:     %d\n", v.integer); ❻
}
--------------------------------------------------------------------------------
The ultimate answer: 42 ❸
Euler's number e:    2.718282 ❺
A dumpster fire:     -1961734133  ❻
```

Listing 2-18: A program using the union Variant from Listing 2-17

You declare a Variant v at ❶. Next, you interpret v as an integer, set its value to 42 ❷, and print it ❸. You then reinterpret v as a float and reassign its value ❹. You print it to the console, and all appears well ❺. So far so good.

Disaster strikes only when you try to interpret v as an integer again ❻. You clobbered over the original value of v (42) ❷ when assigning Euler's number ❹.

That's the main problem with unions: it's up to you to keep track of which interpretation is appropriate. The compiler won't help you.

You should avoid using unions in all but the rarest of cases, and you won't see them in this book. "variant" on page 379 discusses some safer options when you require poly-type functionality.

Fully Featured C++ Classes

POD classes contain only data members, and sometimes that's all you want from a class. However, designing a program using only PODs can create a lot of complexity. You can fight such complexity with *encapsulation*, a design pattern that binds data with the functions that manipulate it. Placing related functions and data together helps to simplify code in at least two ways. First, you can put related code in one place, which helps you to reason about your program. You can understand how a code segment works because it describes in one place both program state and how your code modifies that state. Second, you can hide some of a class's code and data from the rest of your program using a practice called *information hiding*.

In C++, you achieve encapsulation by adding methods and access controls to class definitions.

Methods

Methods are member functions. They create an explicit connection among a class, its data members, and some code. Defining a method is as simple as adding a function to a class definition. Methods have access to all of a class's members.

Consider an example class ClockOfTheLongNow that keeps track of the year. You define an int year member and an add_year method that increments it:

```
struct ClockOfTheLongNow {
  void add_year() { ❶
    year++; ❷
  }
  int year; ❸
};
```

The add_year method's declaration ❶ looks like any other function that takes no parameters and returns no value. Within the method, you increment ❷ the member year ❸. Listing 2-19 shows how you can use the class to keep track of a year.

```
#include <cstdio>

struct ClockOfTheLongNow {
  --snip--
};

int main() {
  ClockOfTheLongNow clock; ❶
  clock.year = 2017; ❷
  clock.add_year(); ❸
  printf("year: %d\n", clock.year); ❹
  clock.add_year(); ❺
  printf("year: %d\n", clock.year); ❻
}
--------------------------------------------------------------------------------
year: 2018 ❹
year: 2019 ❻
```

Listing 2-19: A program using the ClockOfTheLongNow struct

You declare the ClockOfTheLongNow instance clock ❶ and then set the year of clock to 2017 ❷. Next, you call the add_year method on clock ❸ and then print the value of clock.year ❹. You complete the program by incrementing ❺ and printing ❻ once more.

Access Controls

Access controls restrict class-member access. *Public* and *private* are the two major access controls. Anyone can access a public member, but only a class can access its private members. All struct members are public by default.

Private members play an important role in encapsulation. Consider again the ClockOfTheLongNow class. As it stands, the year member can be accessed from anywhere—for both reading and writing. Suppose you want to protect against the value of the year being less than 2019. You can accomplish this in two steps: you make year private, and you require anyone using the class (consumers) to interact with year only through the struct's methods. Listing 2-20 illustrates this approach.

```
struct ClockOfTheLongNow {
  void add_year() {
    year++;
  }
  bool set_year(int new_year) { ❶
    if (new_year < 2019) return false; ❷
    year = new_year;
    return true;
  }
  int get_year() { ❸
    return year;
  }
private: ❹
  int year;
};
```

Listing 2-20: An updated ClockOfTheLongNow from Listing 2-19 that encapsulates year

You've added two methods to ClockOfTheLongNow: a *setter* ❶ and a *getter* ❸ for year. Rather than allowing a user of ClockOfTheLongNow to modify year directly, you set the year with set_year. This addition of input validation ensures that new_year will never be less than 2019 ❷. If it is, the code returns false and leaves year unmodified. Otherwise, year is updated and returns true. To obtain the value of year, the user calls get_year.

You've used the access control label private ❹ to prohibit consumers from accessing year. Now, users can access year only from within ClockOfTheLongNow.

The class Keyword

You can replace the struct keyword with the class keyword, which declares members private by default. Aside from default access control, classes declared with the struct and class keywords are the same. For example, you could declare ClockOfTheLongNow in the following way:

```
class ClockOfTheLongNow {
  int year;
public:
  void add_year() {
    --snip--
  }
  bool set_year(int new_year) {
    --snip--
  }
  int get_year() {
    --snip--
  }
};
```

Which way you declare classes is a matter of style. There is absolutely no difference between struct and class aside from the default access control. I prefer using struct keywords because I like having the public members listed first. But you'll see all kinds of conventions out in the wild. Cultivate a style and stick to it.

Initializing Members

Having encapsulated year, you must now use methods to interact with ClockOfTheLongNow. Listing 2-21 shows how you can stitch these methods together into a program that attempts to set the year to 2018. This fails, and the program then sets the year to 2019, increments the year, and prints its final value.

```
#include <cstdio>

struct ClockOfTheLongNow {
  --snip--
}

int main() {
  ClockOfTheLongNow clock; ❶
  if(!clock.set_year(2018)) { ❷ // will fail; 2018 < 2019
    clock.set_year(2019); ❸
  }
  clock.add_year(); ❹
  printf("year: %d", clock.get_year());
}
--------------------------------------------------------------------------------
year: 2020 ❺
```

Listing 2-21: A program using the ClockOfTheLongNow to illustrate the use of methods

You declare a clock ❶ and attempt to set its year to 2018 ❷. This fails because 2018 is less than 2019, and the program then sets the year to 2019 ❸. You increment the year once ❹ and then print its value.

In Chapter 1, you saw how uninitialized variables can contain random data as you stepped through the debugger. The ClockOfTheLongNow struct has the same problem: when clock is declared ❶, year is uninitialized. You want to guarantee that year is never less than 2019 *under any circumstances*. Such a requirement is called a *class invariant*: a feature of a class that is always true (that is, it never varies).

In this program, clock eventually gets into a good state ❸, but you can do better by employing a *constructor*. Constructors initialize objects and enforce class invariants from the very beginning of an object's life.

Constructors

Constructors are special methods with special declarations. Constructor declarations don't state a return type, and their name matches the class's name. For example, the constructor in Listing 2-22 takes no arguments and sets year to 2019, which causes year to default to 2019.

```
#include <cstdio>

struct ClockOfTheLongNow {
  ClockOfTheLongNow() { ❶
    year = 2019; ❷
  }
  --snip--
};

int main() {
  ClockOfTheLongNow clock; ❸
  printf("Default year: %d", clock.get_year()); ❹
}
```

```
Default year: 2019 ❹
```

Listing 2-22: Improving Listing 2-21 with a parameterless constructor

The constructor takes no arguments ❶ and sets year to 2019 ❷. When you declare a new ClockOfTheLongNow ❸, year defaults to 2019. You access year using get_year and print it to the console ❹.

What if you want to initialize a ClockOfTheLongNow with a custom year? Constructors can take any number of arguments. You can implement as many constructors as you'd like, as long as their argument types differ.

Consider the example in Listing 2-23 where you add a constructor taking an int. The constructor initializes year to the argument's value.

```
#include <cstdio>

struct ClockOfTheLongNow {
  ClockOfTheLongNow(int year_in) { ❶
    if(!set_year(year_in)) { ❷
      year = 2019; ❸
    }
  }
  --snip--
};

int main() {
  ClockOfTheLongNow clock{ 2020 }; ❹
  printf("Year: %d", clock.get_year()); ❺
}
--------------------------------------------------------------------------------
Year: 2020 ❺
```

Listing 2-23: Elaborating Listing 2-22 with another constructor

The new constructor ❶ takes a single `year_in` argument of type `int`. You call `set_year` with `year_in` ❷. If `set_year` returns false, the caller provided bad input, and you override `year_in` with the default value of 2019 ❸. In `main`, you make a clock with the new constructor ❹ and then print the result ❺.

The conjuration `ClockOfTheLongNow clock{ 2020 };` is called an initialization.

NOTE *You might not like the idea that invalid* `year_in` *instances were silently corrected to 2019* ❸*. I don't like it either. Exceptions solve this problem; you'll learn about them in "Exceptions" on page 98.*

Initialization

Object initialization, or simply *initialization,* is how you bring objects to life. Unfortunately, object initialization syntax is complicated. Fortunately, the initialization process is straightforward. This section distills the bubbling cauldron of C++ object initialization into a palatable narrative.

Initializing a Fundamental Type to Zero

Let's start by initializing an object of fundamental type to zero. There are four ways to do so:

```
int a = 0;    ❶// Initialized to 0
int b{};      ❷// Initialized to 0
int c = {};   ❸// Initialized to 0
int d;        ❹// Initialized to 0 (maybe)
```

Three of these are reliable: explicitly set the value using a literal ❶, use braces `{}` ❷, or use the equals-plus-braces approach `= {}` ❸. Declaring the object with no extra notation ❹ is unreliable; it works only in certain situations. Even if you know what these situations are, you should avoid relying on this behavior because it sows confusion.

Using braces `{}` to initialize a variable is, unsurprisingly, called *braced initialization.* Part of the reason C++ initialization syntax is such a mess is that the language grew out of C, where object life cycles are primitive, into a language with a robust and featureful object life cycle. Language designers incorporated braced initialization into modern C++ to help smooth over the sharp corners this has caused in the initialization syntax. In short, no matter the object's scope or type, *braced initialization is always applicable,* whereas the other notations are not. Later in the chapter, you'll learn a general rule that encourages widespread use of braced initialization.

Initializing a Fundamental Type to an Arbitrary Value

Initializing to an arbitrary value is similar to initializing a fundamental type to zero:

```
int e = 42;    ❶ // Initialized to 42
int f{ 42 };   ❷ // Initialized to 42
int g = { 42 };❸ // Initialized to 42
int h(42);     ❹ // Initialized to 42
```

There are four ways: equals ❶, braced initialization ❷, equals-plus-braces initialization ❸, and parentheses ❹. All of these produce identical code.

Initializing PODs

The notation for initializing a POD mostly follows fundamental types. Listing 2-24 illustrates the similarity by declaring a POD type containing three members and initializing instances of it with various values.

```
#include <cstdint>

struct PodStruct {
  uint64_t a;
  char b[256];
  bool c;
};

int main() {
  PodStruct initialized_pod1{};    ❶   // All fields zeroed
  PodStruct initialized_pod2 = {}; ❷   // All fields zeroed

  PodStruct initialized_pod3{ 42, "Hello" }; ❸       // Fields a & b set; c = 0
  PodStruct initialized_pod4{ 42, "Hello", true }; ❹ // All fields set
}
```

Listing 2-24: A program illustrating various ways to initialize a POD

Initializing a POD object to zero is similar to initializing objects of fundamental types to zero. The braces ❶ and equals-plus-braces ❷ approaches produce the same code: fields initialize to zero.

WARNING *You cannot use the equals-zero approach with PODs. The following will not compile because it's expressly forbidden in the language rules:*

```
PodStruct initialized_pod = 0;
```

Initializing PODs to Arbitrary Values

You can initialize fields to arbitrary values using braced initializers. The arguments within braced initializers must match types with POD members. The order of arguments from left to right matches the order of members from top to bottom. Any omitted members are zeroed. Members `a` and `b` initialize to `42` and `Hello` after the initialization of `initialized_pod3` ❸, and `c` is zeroed (set to false) because you omitted it from the braced initialization. The initialization of `initialized_pod4` ❹ includes an argument for `c` (`true`), so its value is set to true after initialization.

The equals-plus-braces initialization works identically. For example, you could replace ❹ with this:

```
PodStruct initialized_pod4 = { 42, "Hello", true };
```

You can only omit fields from right to left, so the following won't compile:

```
PodStruct initialized_pod4 = { 42, true };
```

WARNING *You cannot use parentheses to initialize PODs. The following will not compile:*

```
PodStruct initialized_pod(42, "Hello", true);
```

Initializing Arrays

You initialize arrays like PODs. The main difference between array and POD declarations is that arrays specify length. Recall that this argument goes in square brackets `[]`.

When you use braced initializers to initialize arrays, the length argument becomes optional; the compiler can infer the size argument from the number of braced initializer arguments.

Listing 2-25 illustrates some ways to initialize an array.

```
int main() {
  int array_1[]{ 1, 2, 3 };  ❶ // Array of length 3; 1, 2, 3
  int array_2[5]{};          ❷ // Array of length 5; 0, 0, 0, 0, 0
  int array_3[5]{ 1, 2, 3 }; ❸ // Array of length 5; 1, 2, 3, 0, 0
  int array_4[5];            ❹ // Array of length 5; uninitialized values
}
```

Listing 2-25: A program listing various ways to initialize an array

The array `array_1` has length three, and its elements equal 1, 2, and 3 ❶. The array `array_2` has length five because you specified a length argument ❷. The braced initializer is empty, so all five elements initialize to zero. The array `array_3` also has length five, but the braced initializer is not empty. It contains three elements, so the remaining two elements initialize to zero ❸. The array `array_4` has no braced initializer, so it contains uninitialized objects ❹.

WARNING *Whether `array_5` is initialized or not actually depends on the same rules as does initializing a fundamental type. The object's storage duration, which you'll learn about in "An Object's Storage Duration" on page 89, determines the rules. You don't have to memorize these rules if you're explicit about initialization.*

Fully Featured Classes

Unlike fundamental types and PODs, fully featured classes are *always initialized*. In other words, one of a fully featured class's constructors always gets called during initialization. Which constructor is called depends on the arguments given during initialization.

The class in Listing 2-26 helps clarify how to use fully featured classes.

```
#include <cstdio>

struct Taxonomist {
  Taxonomist() { ❶
    printf("(no argument)\n");
  }
  Taxonomist(char x) { ❷
    printf("char: %c\n", x);
  }
  Taxonomist(int x) { ❸
    printf("int: %d\n", x);
  }
  Taxonomist(float x) { ❹
    printf("float: %f\n", x);
  }
};
```

Listing 2-26: A class announcing which of its several constructors gets called during initialization

The `Taxonomist` class has four constructors. If you supply no argument, the constructor with no arguments gets called ❶. If you supply a `char`, `int`, or `float` during initialization, the corresponding constructor gets called: ❷, ❸, or ❹, respectively. In each case, the constructor alerts you with a `printf` statement.

Listing 2-27 initializes several `Taxonomists` using different syntaxes and arguments.

```
#include <cstdio>

struct Taxonomist {
  --snip--
};

int main() {
  Taxonomist t1; ❶
  Taxonomist t2{ 'c' }; ❷
  Taxonomist t3{ 65537 }; ❸
  Taxonomist t4{ 6.02e23f }; ❹
  Taxonomist t5('g'); ❺
  Taxonomist t6 = { 'l' }; ❻
  Taxonomist t7{}; ❼
  Taxonomist t8(); ❽
}
```

```
(no argument) ❶
char: c ❷
int: 65537 ❸
float: 6020000172718952294645576.000000 ❹
char: g ❺
char: l ❻
(no argument) ❼
```

Listing 2-27: A program using the `Taxonomist` *class with various initialization syntaxes*

Without any braces or parentheses, the no argument constructor gets called ❶. Unlike with POD and fundamental types, you can rely on this initialization no matter where you've declared the object. With braced initializers, the `char` ❷, `int` ❸, and `float` ❹ constructors get called as expected. You can also use parentheses ❺ and the equals-plus-braces syntaxes ❻; these invoke the expected constructors.

Although fully featured classes always get initialized, some programmers like the uniformity of using the same initialization syntax for all objects. This is no problem with braced initializers; the default constructor gets invoked as expected ❼.

Unfortunately, using parentheses ❽ causes some surprising behavior. You get no output.

If you squint a little bit, this initialization ❽ looks like a function declaration, and that's because it is. Because of some arcane language-parsing rules, what you've declared to the compiler is that a yet-to-be-defined function `t8` takes no arguments and returns an object of type `Taxonomist`. Ouch.

NOTE *"Function Declarations" on page 244 covers function declarations in more detail. But for now, just know that you can provide a function declaration that defines a function's modifiers, name, arguments, and return type and then later provide the body in its definition.*

This widely known problem is called the *most vexing parse*, and it's a major reason why the C++ community added braced initialization syntax to the language. *Narrowing conversions* are another problem.

Narrowing Conversions

Braced initialization will generate warnings whenever implicit narrowing conversions are encountered. This is a nice feature that can save you from nasty bugs. Consider the following example:

```
float a{ 1 };
float b{ 2 };
int narrowed_result(a/b); ❶ // Potentially nasty narrowing conversion
int result{ a/b };        ❷ // Compiler generates warning
```

Dividing two `float` literals yields a float. When initializing `narrowed_result` ❶, the compiler silently narrows the result of `a/b` (0.5) to 0 because you've used parentheses `( )` to initialize. When you use braced initializers, the compiler generates a warning ❷.

Initializing Class Members

You can use braced initializers to initialize the members of classes, as demonstrated here:

```
struct JohanVanDerSmut {
  bool gold = true; ❶
  int year_of_smelting_accident{ 1970 }; ❷
  char key_location[8] = { "x-rated" }; ❸
};
```

The `gold` member is initialized using the equals initialization ❶, `year_of_smelting_accident` using braced initialization ❷, and `key_location` using braces-plus-equals initialization ❸. It's not possible to use parentheses to initialize member variables.

Brace Yourself

The options for initializing objects bewilder even experienced C++ programmers. Here's a general rule to make initialization simple: *use braced initializers everywhere*. Braced initializers work as intended almost everywhere, and they cause the fewest surprises. For this reason, braced initialization is also called *uniform initialization*. The remainder of the book follows this guidance.

WARNING *You'll break the* use braced initializers everywhere *rule for certain classes in C++ stdlib. Part II will make these exceptions to the rule very clear.*

The Destructor

An object's *destructor* is its cleanup function. The destructor is invoked before an object is destroyed. Destructors are almost never called explicitly: the compiler will ensure that each object's destructor is called as appropriate. You declare a class's destructor with the tilde ~ followed by the name of the class.

The following `Earth` class has a destructor that prints `Making way for hyperspace bypass`:

```
#include <cstdio>

struct Earth {
  ~Earth() { // Earth's destructor
      printf("Making way for hyperspace bypass");
  }
}
```

Defining a destructor is optional. If you do decide to implement a destructor, it must not take any arguments. Examples of actions you might want to take in a destructor include releasing file handles, flushing network sockets, and freeing dynamic objects.

If you don't define a destructor, a default destructor is automatically generated. The default destructor's behavior is to perform no action.

You'll learn a whole lot more about destructors in "Tracing the Object Life Cycle" on page 96.

Summary

This chapter presented the foundation of C++, which is its type system. You first learned about fundamental types, the building blocks of all other types. Then you continued with user-defined types, including the `enum class`, POD classes, and fully featured C++ classes. You capped off your tour of classes with a discussion of constructors, initialization syntax, and destructors.

EXERCISES

2-1. Create an `enum class Operation` that has values `Add`, `Subtract`, `Multiply`, and `Divide`.

2-2. Create a `struct Calculator`. It should have a single constructor that takes an `Operation`.

2-3. Create a method on `Calculator` called `int calculate(int a, int b)`. Upon invocation, this method should perform addition, subtraction, multiplication, or division based on its constructor argument and return the result.

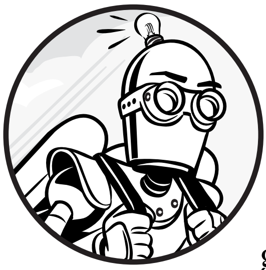
2-4. Experiment with different means of initializing `Calculator` instances.

FURTHER READING

- *ISO International Standard ISO/IEC (2017) – Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)
- *The C++ Programming Language*, 4th Edition, by Bjarne Stroustrup (Pearson Education, 2013)
- *Effective Modern C++* by Scott Meyers (O'Reilly Media, 2014)
- "C++ Made Easier: Plain Old Data" by Andrew Koenig and Barbara E. Moo (Dr. Dobb's, 2002; *http://www.drdobbs.com/c-made-easier-plain-old-data/184401508/*)

3
REFERENCE TYPES

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
—Brian Kernighan



Reference types store the memory addresses of objects. These types enable efficient programming, and many elegant design patterns feature them. In this chapter, I'll discuss the two kinds of reference types: pointers and references. I'll also discuss `this`, `const`, and `auto` along the way.

Pointers

Pointers are the fundamental mechanism used to refer to memory addresses. Pointers encode both pieces of information required to interact with another object—that is, the object's address and the object's type.

You can declare a pointer's type by appending an asterisk (*) to the pointed-to type. For example, you declare a pointer to int called my_ptr as follows:

```
int* my_ptr;
```

The format specifier for a pointer is %p. For example, to print the value in my_ptr, you could use the following:

```
printf("The value of my_ptr is %p.", my_ptr);
```

Pointers are very low-level objects. Although they play a central role in most C programs, C++ offers higher-level, sometimes more efficient, constructs that obviate the need to deal with memory addresses directly. Nonetheless, pointers are a foundational concept that you'll no doubt come across in your system-programming travels.

In this section, you'll learn how to find the address of an object and how to assign the result to a pointer variable. You'll also learn how to perform the opposite operation, which is called *dereferencing*: given a pointer, you can obtain the object residing at the corresponding address.

You'll learn more about *arrays*, the simplest construct for managing an object collection, as well as how arrays relate to pointers. As low-level constructs, arrays and pointers are relatively dangerous. You'll learn about what can go wrong when pointer- and array-based programs go awry.

This chapter introduces two special kinds of pointers: void pointers and std::byte pointers. These very useful types have some special behaviors that you'll need to keep in mind. Additionally, you'll learn how to encode empty pointers with nullptr and how to use pointers in Boolean expressions to determine whether they're empty.

Addressing Variables

You can obtain the address of a variable by prepending the *address-of operator* (&). You might want to use this operator to initialize a pointer so it "points to" the corresponding variable. Such programming requirements arise very often in operating systems programming. For example, major operating systems, such as Windows, Linux, and FreeBSD, have interfaces that use pointers heavily.

Listing 3-1 demonstrates how to obtain the address of an int.

```
#include <cstdio>

int main() {
  int gettysburg{}; ❶
  printf("gettysburg: %d\n", gettysburg); ❷
  int *gettysburg_address = &gettysburg; ❸
  printf("&gettysburg: %p\n", gettysburg_address); ❹
}
```

Listing 3-1: A program featuring the address-of operator & and a terrible pun

First, you declare the integer gettysburg ❶ and print its value ❷. Then you declare a pointer, called gettysburg_address, to that integer's address ❸; notice that the asterisk prepends the pointer and the ampersand prepends gettysburg. Finally, you print the pointer to the screen ❹ to reveal the gettysburg integer's address.

If you run Listing 3-1 on Windows 10 (x86), you should see the following output:

```
gettysburg: 0
&gettysburg: 0053FBA8
```

Running the same code on Windows 10 x64 yields the following output:

```
gettysburg: 0
&gettysburg: 0000007DAB53F594
```

Your output should have an identical value for gettysburg, but gettysburg_address should be different each time. This variation is due to *address space layout randomization*, which is a security feature that scrambles the base address of important memory regions to hamper exploitation.

ADDRESS SPACE LAYOUT RANDOMIZATION

Why does address space layout randomization hamper exploitation? When a hacker finds an exploitable condition in a program, they can sometimes cram a malicious payload into user-provided input. One of the first security features designed to prevent a hacker from getting this malicious payload to execute is to make all data sections non-executable. If the computer attempts to execute data as code, then the theory is that it knows something's amiss and should terminate the program with an exception.

Some exceedingly clever hackers figured out how to repurpose executable code instructions in totally unforeseen ways by carefully crafting exploits containing so-called *return-oriented programs*. These exploits could arrange to invoke the relevant system APIs to mark their payload executable, hence defeating the non-executable-memory mitigation.

Address space layout randomization combats return-oriented programming by randomizing memory addresses, making it difficult to repurpose existing code because the attacker doesn't know where it resides in memory.

Also note that in the outputs for Listing 3-1, gettysburg_address contains 8 hexadecimal digits (4 bytes) for an x86 architecture and 16 hexadecimal digits (8 bytes) for an x64 architecture. This should make some sense because on modern desktop systems, the pointer size is the same as the CPU's general-purpose register. An x86 architecture has 32-bit (4-byte) general-purpose registers, whereas an x64 architecture has 64-bit (8-byte) general-purpose registers.

Dereferencing Pointers

The *dereference operator* (*) is a unary operator that accesses the object to which a pointer refers. This is the inverse operation of the address-of operator. Given an address, you can obtain the object residing there. Like the address-of operator, system programmers use the dereference operator very often. Many operating system APIs will return pointers, and if you want to access the referred-to object, you'll use the dereference operator.

Unfortunately, the dereference operator can cause a lot of notation-based confusion for beginners because the dereference operator, the pointer declaration, and multiplication all use asterisks. Remember that you append an asterisk to the end of the pointed-to object's type to declare a pointer; however, you prepend the dereference operator—an asterisk—to the pointer, like this:

```
*gettysburg_address
```

After accessing an object by prepending the dereference operator to a pointer, you can treat the result like any other object of the pointed-to type. For example, because `gettysburg` is an integer, you can write the value 17325 into `gettysburg` using `gettysburg_address`. The correct syntax is as follows:

```
*gettysburg_address = 17325;
```

Because the dereferenced pointer—that is, `*gettysburg_address`—appears on the left side of the equal sign, you're writing to the address where `gettysburg` is stored.

If a dereferenced pointer appears anywhere except the left side of an equal sign, you're reading from the address. To retrieve the `int` pointed to by `gettysburg_address`, you just tack on the dereference operator. For instance, the following statement will print the value stored in `gettysburg`:

```
printf("%d", *gettysburg_address);
```

Listing 3-2 uses the dereference operator to read and write.

```
#include <cstdio>

int main() {
  int gettysburg{};
  int* gettysburg_address = &gettysburg; ❶
  printf("Value at gettysburg_address: %d\n", *gettysburg_address); ❷
  printf("Gettysburg Address: %p\n", gettysburg_address); ❸
  *gettysburg_address = 17325; ❹
  printf("Value at gettysburg_address: %d\n", *gettysburg_address); ❺
  printf("Gettysburg Address: %p\n", gettysburg_address); ❻
}
--------------------------------------------------------------------------------
Value at gettysburg_address: 0 ❷
Gettysburg Address: 000000B9EEEFFB04 ❸
```

```
Value at gettysburg_address: 17325 ❺
Gettysburg Address: 000000B9EEEFFB04 ❻
```

Listing 3-2: An example program illustrating reads and writes using a pointer (output is from a Windows 10 x64 machine)

First, you initialize `gettysburg` to zero. Then, you initialize the pointer `gettysburg_address` to the address of `gettysburg` ❶. Next, you print the `int` pointed to by `gettysburg_address` ❷ and the value of `gettysburg_address` itself ❸.

You write the value 17325 into the memory pointed to by `gettysburg_address` ❹ and then print the pointed-to value ❺ and address ❻ again.

Listing 3-2 would be functionally identical if you assigned the value 17325 directly to `gettysburg` instead of to the `gettysburg_address` pointer, like this:

```
  gettysburg = 17325;
```

This example illustrates the close relationship between a pointed-to object (`gettysburg`) and a dereferenced pointer to that object (`*gettysburg_address`).

The Member-of-Pointer Operator

The *member-of-pointer operator*, or *arrow operator* (`->`), performs two simultaneous operations:

- It dereferences a pointer.
- It accesses a member of the pointed-to object.

You can use this operator to reduce *notational friction*, the resistance a programmer feels in expressing their intent in code, when you're handling pointers to classes. You'll need to handle pointers to classes in a variety of design patterns. For example, you might want to pass a pointer to a class as a function parameter. If the receiving function needs to interact with a member of that class, the member-of-pointer operator is the tool for the job.

Listing 3-3 employs the arrow operator to read the `year` from a `ClockOfTheLongNow` object (which you implemented in Listing 2-22 on page 58).

```
#include <cstdio>

struct ClockOfTheLongNow {
  --snip--
};

int main() {
  ClockOfTheLongNow clock;
  ClockOfTheLongNow* clock_ptr = &clock; ❶
  clock_ptr->set_year(2020); ❷
  printf("Address of clock: %p\n", clock_ptr); ❸
  printf("Value of clock's year: %d", clock_ptr->get_year()); ❹
}
```

```
Address of clock: 000000C6D3D5FBE4 ❸
Value of clock's year: 2020 ❹
```

Listing 3-3: Using a pointer and the arrow operator to manipulate the `ClockOfTheLongNow` *object (output is from a Windows 10 x64 machine)*

You declare a `clock` and then store its address in `clock_ptr` ❶. Next, you use the arrow operator to set the `year` member of `clock` to 2020 ❷. Finally, you print the address of `clock` ❸ and the value of `year` ❹.

You could achieve an identical result using the dereference (`*`) and member of (`.`) operators. For example, you could have written the last line of Listing 3-3 as follows:

```
  printf("Value of clock's year: %d", (*clock_ptr).get_year());
```

First, you dereference `clock_ptr`, and then you access the `year`. Although this is equivalent to invoking the pointer-to-member operator, it's a more verbose syntax and provides no benefit over its simpler alternative.

NOTE *For now, use parentheses to emphasize the order of operations. Chapter 7 walks through the precedents rules for operators.*

Pointers and Arrays

Pointers share several characteristics with arrays. Pointers encode object location. Arrays encode the location and length of contiguous objects.

At the slightest provocation, an array will *decay* into a pointer. A decayed array loses length information and converts to a pointer to the array's first element. For example:

```
int key_to_the_universe[]{ 3, 6, 9 };
int* key_ptr = key_to_the_universe; // Points to 3
```

First, you initialize an `int` array `key_to_the_universe` with three elements. Next, you initialize the `int` pointer `key_ptr` to `key_to_the_universe`, which decays into a pointer. After initialization, `key_ptr` points to the first element of `key_to_the_universe`.

Listing 3-4 initializes an array containing `College` objects and passes the array to a function as a pointer.

```
#include <cstdio>

struct College {
  char name[256];
};
```

```
void print_name(College* college_ptr❶) {
  printf("%s College\n", college_ptr->name❷);
}

int main() {
  College best_colleges[] = { "Magdalen", "Nuffield", "Kellogg" };
  print_name(best_colleges);
}
--------------------------------------------------------------------------
Magdalen College ❷
```

Listing 3-4: A program illustrating array decay into a pointer

The `print_name` function takes a pointer-to-`College` argument ❶, so the `best_colleges` array decays into a pointer when you call `print_name`. Because arrays decay into pointers to their first element, `college_ptr` at ❶ points to the first `College` in `best_colleges`.

There's another array decay in Listing 3-4 ❷ as well. You use the arrow operator (`->`) to access the `name` member of the `College` pointed to by `college_ptr`, which is itself a `char` array. The `printf` format specifier `%s` expects a C-style string, which is a `char` pointer, and `name` decays into a pointer to satisfy `printf`.

Handling Decay

Often, you pass arrays as two arguments:

- A pointer to the first array element
- The array's length

The mechanism that enables this pattern is square brackets (`[]`), which work with pointers just as with arrays. Listing 3-5 employs this technique.

```
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges❶, size_t n_colleges❷) {
  for (size_t i = 0; i < n_colleges; i++) { ❸
    printf("%s College\n", colleges[i]❹.name❺);
  }
}

int main() {
  College oxford[] = { "Magdalen", "Nuffield", "Kellogg" };
  print_names(oxford, sizeof(oxford) / sizeof(College));
}
```