CHAPTER 2

VARIABLES AND BASIC TYPES

CONTENTS

Types are fundamental to any program: They tell us what our data mean and what operations we can perform on those data.

C++ has extensive support for types. The language defines several primitive types (characters, integers, floating-point numbers, etc.) and provides mechanisms that let us define our own data types. The library uses these mechanisms to define more complicated types such as variable-length character strings, vectors, and so on. This chapter covers the built-in types and begins our coverage of how C++ supports more complicated types.

Types determine the meaning of the data and operations in our programs. The meaning of even as simple a statement as

```
i = i + j;
```

depends on the types of `i` and `j`. If `i` and `j` are integers, this statement has the ordinary, arithmetic meaning of `+`. However, if `i` and `j` are `Sales_item` objects (§ 1.5.1, p. 20), this statement adds the components of these two objects.

2.1 Primitive Built-in Types

C++ defines a set of primitive types that include the **arithmetic types** and a special type named **`void`**. The arithmetic types represent characters, integers, boolean values, and floating-point numbers. The `void` type has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

2.1.1 Arithmetic Types

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

The size of—that is, the number of bits in—the arithmetic types varies across machines. The standard guarantees minimum sizes as listed in Table 2.1. However, compilers are allowed to use larger sizes for these types. Because the number of bits varies, the largest (or smallest) value that a type can represent also varies.

Table 2.1: C++: Arithmetic Types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

The `bool` type represents the truth values `true` and `false`.

There are several character types, most of which exist to support internationalization. The basic character type is `char`. A `char` is guaranteed to be big enough to hold numeric values corresponding to the characters in the machine's basic character set. That is, a `char` is the same size as a single machine byte.

The remaining character types—`wchar_t`, `char16_t`, and `char32_t`—are used for extended character sets. The `wchar_t` type is guaranteed to be large enough to hold any character in the machine's largest extended character set. The types `char16_t` and `char32_t` are intended for Unicode characters. (Unicode is a standard for representing characters used in essentially any natural language.)

The remaining integral types represent integer values of (potentially) different sizes. The language guarantees that an `int` will be at least as large as `short`, a `long` at least as large as an `int`, and `long long` at least as large as `long`. The type `long long` was introduced by the new standard. C++ 11

Machine-Level Representation of the Built-in Types

Computers store data as a sequence of bits, each holding a 0 or 1, such as

```
00011011011100010110010000111011 ...
```

Most computers deal with memory as chunks of bits of sizes that are powers of 2. The smallest chunk of addressable memory is referred to as a "byte." The basic unit of storage, usually a small number of bytes, is referred to as a "word." In C++ a byte has at least as many bits as are needed to hold a character in the machine's basic character set. On most machines a byte contains 8 bits and a word is either 32 or 64 bits, that is, 4 or 8 bytes.

Most computers associate a number (called an "address") with each byte in memory. On a machine with 8-bit bytes and 32-bit words, we might view a word of memory as follows

736424	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**
736425	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**1**
736426	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**
736427	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**

Here, the byte's address is on the left, with the 8 bits of the byte following the address.

We can use an address to refer to any of several variously sized collections of bits starting at that address. It is possible to speak of the word at address 736424 or the byte at address 736427. To give meaning to memory at a given address, we must know the type of the value stored there. The type determines how many bits are used and how to interpret those bits.

If the object at location 736424 has type `float` and if `float`s on this machine are stored in 32 bits, then we know that the object at that address spans the entire word. The value of that `float` depends on the details of how the machine stores floating-point numbers. Alternatively, if the object at location 736424 is an `unsigned char` on a machine using the ISO-Latin-1 character set, then the byte at that address represents a semicolon.

The floating-point types represent single-, double-, and extended-precision values. The standard specifies a minimum number of significant digits. Most compilers provide more precision than the specified minimum. Typically, `float`s are represented in one word (32 bits), `double`s in two words (64 bits), and `long double`s in either three or four words (96 or 128 bits). The `float` and `double` types typically yield about 7 and 16 significant digits, respectively. The type `long double`

is often used as a way to accommodate special-purpose floating-point hardware; its precision is more likely to vary from one implementation to another.

Signed and Unsigned Types

Except for `bool` and the extended character types, the integral types may be **signed** or **unsigned**. A signed type represents negative or positive numbers (including zero); an unsigned type represents only values greater than or equal to zero.

The types `int`, `short`, `long`, and `long long` are all signed. We obtain the corresponding unsigned type by adding `unsigned` to the type, such as `unsigned long`. The type `unsigned int` may be abbreviated as `unsigned`.

Unlike the other integer types, there are three distinct basic character types: `char`, `signed char`, and `unsigned char`. In particular, `char` is not the same type as `signed char`. Although there are three character types, there are only two representations: signed and unsigned. The (plain) `char` type uses one of these representations. Which of the other two character representations is equivalent to `char` depends on the compiler.

In an unsigned type, all the bits represent the value. For example, an 8-bit `unsigned char` can hold the values from 0 through 255 inclusive.

The standard does not define how signed types are represented, but does specify that the range should be evenly divided between positive and negative values. Hence, an 8-bit `signed char` is guaranteed to be able to hold values from –127 through 127; most modern machines use representations that allow values from –128 through 127.

Advice: Deciding Which Type to Use

C++, like C, is designed to let programs get close to the hardware when necessary. The arithmetic types are defined to cater to the peculiarities of various kinds of hardware. Accordingly, the number of arithmetic types in C++ can be bewildering. Most programmers can (and should) ignore these complexities by restricting the types they use. A few rules of thumb can be useful in deciding which type to use:

- **Use an unsigned type when you know that the values cannot be negative.**
- **Use `int` for integer arithmetic. `short` is usually too small and, in practice, `long` often has the same size as `int`. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.**
- **Do not use plain `char` or `bool` in arithmetic expressions. Use them *only* to hold characters or truth values. Computations using `char` are especially problematic because `char` is `signed` on some machines and `unsigned` on others. If you need a tiny integer, explicitly specify either `signed char` or `unsigned char`.**
- **Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable run-time cost.**

EXERCISES SECTION 2.1.1

Exercise 2.1: What are the differences between `int`, `long`, `long long`, and `short`? Between an unsigned and a signed type? Between a `float` and a `double`?

Exercise 2.2: To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

2.1.2 Type Conversions

The type of an object defines the data that an object might contain and what operations that object can perform. Among the operations that many types support is the ability to **convert** objects of the given type to other, related types.

Type conversions happen automatically when we use an object of one type where an object of another type is expected. We'll have more to say about conversions in § 4.11 (p. 159), but for now it is useful to understand what happens when we assign a value of one type to an object of another type.

When we assign one arithmetic type to another:

```
bool b = 42;              // b is true
int i = b;                // i has value 1
i = 3.14;                 // i has value 3
double pi = i;            // pi has value 3.0
unsigned char c = -1;     // assuming 8-bit chars, c has value 255
signed char c2 = 256;     // assuming 8-bit chars, the value of c2 is undefined
```

what happens depends on the range of the values that the types permit:

- When we assign one of the nonbool arithmetic types to a `bool` object, the result is `false` if the value is `0` and `true` otherwise.
- When we assign a `bool` to one of the other arithmetic types, the resulting value is `1` if the `bool` is `true` and `0` if the `bool` is `false`.
- When we assign a floating-point value to an object of integral type, the value is truncated. The value that is stored is the part before the decimal point.
- When we assign an integral value to an object of floating-point type, the fractional part is zero. Precision may be lost if the integer has more bits than the floating-point object can accommodate.
- If we assign an out-of-range value to an object of unsigned type, the result is the remainder of the value modulo the number of values the target type can hold. For example, an 8-bit `unsigned char` can hold values from 0 through 255, inclusive. If we assign a value outside this range, the compiler assigns the remainder of that value modulo 256. Therefore, assigning –1 to an 8-bit `unsigned char` gives that object the value 255.
- If we assign an out-of-range value to an object of signed type, the result is **undefined**. The program might appear to work, it might crash, or it might produce garbage values.

ADVICE: AVOID UNDEFINED AND IMPLEMENTATION-DEFINED BEHAVIOR

Undefined behavior results from errors that the compiler is not required (and sometimes is not able) to detect. Even if the code compiles, a program that executes an undefined expression is in error.

Unfortunately, programs that contain undefined behavior can appear to execute correctly in some circumstances and/or on some compilers. There is no guarantee that the same program, compiled under a different compiler or even a subsequent release of the same compiler, will continue to run correctly. Nor is there any guarantee that what works with one set of inputs will work with another.

Similarly, programs usually should avoid implementation-defined behavior, such as assuming that the size of an `int` is a fixed and known value. Such programs are said to be *nonportable*. When the program is moved to another machine, code that relied on implementation-defined behavior may fail. Tracking down these sorts of problems in previously working programs is, mildly put, unpleasant.

The compiler applies these same type conversions when we use a value of one arithmetic type where a value of another arithmetic type is expected. For example, when we use a nonbool value as a condition (§ 1.4.1, p. 12), the arithmetic value is converted to `bool` in the same way that it would be converted if we had assigned that arithmetic value to a `bool` variable:

```
int i = 42;
if (i) //  condition will evaluate as true
    i = 0;
```

If the value is `0`, then the condition is `false`; all other (nonzero) values yield `true`.

By the same token, when we use a `bool` in an arithmetic expression, its value always converts to either `0` or `1`. As a result, using a `bool` in an arithmetic expression is almost surely incorrect.

Expressions Involving Unsigned Types

Although we are unlikely to intentionally assign a negative value to an object of unsigned type, we can (all too easily) write code that does so implicitly. For example, if we use both `unsigned` and `int` values in an arithmetic expression, the `int` value ordinarily is converted to `unsigned`. Converting an `int` to `unsigned` executes the same way as if we assigned the `int` to an `unsigned`:

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl;  //  prints -84
std::cout << u + i << std::endl;  //  if 32-bit ints, prints 4294967264
```

In the first expression, we add two (negative) `int` values and obtain the expected result. In the second expression, the `int` value `-42` is converted to `unsigned` before the addition is done. Converting a negative number to `unsigned` behaves exactly as if we had attempted to assign that negative value to an `unsigned` object. The value "wraps around" as described above.

Regardless of whether one or both operands are unsigned, if we subtract a value from an unsigned, we must be sure that the result cannot be negative:

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; //  ok: result is 32
std::cout << u2 - u1 << std::endl; //  ok: but the result will wrap around
```

The fact that an unsigned cannot be less than zero also affects how we write loops. For example, in the exercises to § 1.4.1 (p. 13), you were to write a loop that used the decrement operator to print the numbers from `10` down to `0`. The loop you wrote probably looked something like

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

We might think we could rewrite this loop using an `unsigned`. After all, we don't plan to print negative numbers. However, this simple change in type means that our loop will never terminate:

```
//  WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

Consider what happens when `u` is `0`. On that iteration, we'll print `0` and then execute the expression in the `for` loop. That expression, `--u`, subtracts `1` from `u`. That result, `-1`, won't fit in an `unsigned` value. As with any other out-of-range value, `-1` will be transformed to an `unsigned` value. Assuming 32-bit `int`s, the result of `--u`, when `u` is `0`, is `4294967295`.

One way to write this loop is to use a `while` instead of a `for`. Using a `while` lets us decrement before (rather than after) printing our value:

```
unsigned u = 11; //  start the loop one past the first element we want to print
while (u > 0) {
     --u;         //  decrement first, so that the last iteration will print 0
    std::cout << u << std::endl;
}
```

This loop starts by decrementing the value of the loop control variable. On the last iteration, `u` will be `1` on entry to the loop. We'll decrement that value, meaning that we'll print `0` on this iteration. When we next test `u` in the `while` condition, its value will be `0` and the loop will exit. Because we start by decrementing `u`, we have to initialize `u` to a value one greater than the first value we want to print. Hence, we initialize `u` to `11`, so that the first value printed is `10`.

CAUTION: DON'T MIX SIGNED AND UNSIGNED TYPES

Expressions that mix signed and unsigned values can yield surprising results when the signed value is negative. It is essential to remember that signed values are automatically converted to unsigned. For example, in an expression like `a * b`, if `a` is `-1` and `b` is `1`, then if both `a` and `b` are `int`s, the value is, as expected `-1`. However, if `a` is `int` and `b` is an `unsigned`, then the value of this expression depends on how many bits an `int` has on the particular machine. On our machine, this expression yields `4294967295`.

EXERCISES SECTION 2.1.2

Exercise 2.3: What output will the following code produce?

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;
int i = 10, i2 = 42;
std::cout << i2 - i << std::endl;
std::cout << i - i2 << std::endl;
std::cout << i - u << std::endl;
std::cout << u - i << std::endl;
```

Exercise 2.4: Write a program to check whether your predictions were correct. If not, study this section until you understand what the problem is.

2.1.3 Literals

A value, such as `42`, is known as a **literal** because its value self-evident. Every literal has a type. The form and value of a literal determine its type.

Integer and Floating-Point Literals

We can write an integer literal using decimal, octal, or hexadecimal notation. Integer literals that begin with `0` (zero) are interpreted as octal. Those that begin with either `0x` or `0X` are interpreted as hexadecimal. For example, we can write the value `20` in any of the following three ways:

```
20 /* decimal */  024 /* octal */  0x14 /* hexadecimal */
```

The type of an integer literal depends on its value and notation. By default, decimal literals are signed whereas octal and hexadecimal literals can be either signed or unsigned types. A decimal literal has the smallest type of `int`, `long`, or `long long` (i.e., the first type in this list) in which the literal's value fits. Octal and hexadecimal literals have the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which the literal's value fits. It is an error to use a literal that is too large to fit in the largest related type. There are no literals of type `short`. We'll see in Table 2.2 (p. 40) that we can override these defaults by using a suffix.

Although integer literals may be stored in signed types, technically speaking, the value of a decimal literal is never a negative number. If we write what appears to be a negative decimal literal, for example, `-42`, the minus sign is *not* part of the literal. The minus sign is an operator that negates the value of its (literal) operand.

Floating-point literals include either a decimal point or an exponent specified using scientific notation. Using scientific notation, the exponent is indicated by either `E` or `e`:

```
3.14159    3.14159E0    0.    0e0    .001
```

By default, floating-point literals have type `double`. We can override the default using a suffix from Table 2.2 (overleaf).

Character and Character String Literals

A character enclosed within single quotes is a literal of type `char`. Zero or more characters enclosed in double quotation marks is a string literal:

```
'a'   // character literal
"Hello World!"   // string literal
```

The type of a string literal is ***array*** of constant `char`s, a type we'll discuss in § 3.5.4 (p. 122). The compiler appends a null character (`'\0'`) to every string literal. Thus, the actual size of a string literal is one more than its apparent size. For example, the literal `'A'` represents the single character `A`, whereas the string literal `"A"` represents an array of two characters, the letter `A` and the null character.

Two string literals that appear adjacent to one another and that are separated only by spaces, tabs, or newlines are concatenated into a single literal. We use this form of literal when we need to write a literal that would otherwise be too large to fit comfortably on a single line:

```
// multiline string literal
std::cout << "a really, really long string literal "
             "that spans two lines" << std::endl;
```

Escape Sequences

Some characters, such as backspace or control characters, have no visible image. Such characters are **nonprintable**. Other characters (single and double quotation marks, question mark, and backslash) have special meaning in the language. Our programs cannot use any of these characters directly. Instead, we use an **escape sequence** to represent such characters. An escape sequence begins with a backslash. The language defines several escape sequences:

newline	`\n`	horizontal tab	`\t`	alert (bell)	`\a`
vertical tab	`\v`	backspace	`\b`	double quote	`\"`
backslash	`\\`	question mark	`\?`	single quote	`\'`
carriage return	`\r`	formfeed	`\f`		

We use an escape sequence as if it were a single character:

```
std::cout << '\n';          // prints a newline
std::cout << "\tHi!\n";     // prints a tab followd by "Hi!" and a newline
```

We can also write a generalized escape sequence, which is `\x` followed by one or more hexadecimal digits or a `\` followed by one, two, or three octal digits. The value represents the numerical value of the character. Some examples (assuming the Latin-1 character set):

```
\7 (bell)     \12 (newline)      \40 (blank)
\0 (null)     \115 ('M')         \x4d ('M')
```

As with an escape sequence defined by the language, we use these escape sequences as we would any other character:

```
std::cout << "Hi \x4dO\115!\n"; // prints Hi MOM! followed by a newline
std::cout << '\115' << '\n';    // prints M followed by a newline
```

Note that if a \ is followed by more than three octal digits, only the first three are associated with the \. For example, `"\1234"` represents two characters: the character represented by the octal value 123 and the character 4. In contrast, `\x` uses up all the hex digits following it; `"\x1234"` represents a single, 16-bit character composed from the bits corresponding to these four hexadecimal digits. Because most machines have 8-bit `chars`, such values are unlikely to be useful. Ordinarily, hexadecimal characters with more than 8 bits are used with extended characters sets using one of the prefixes from Table 2.2.

Specifying the Type of a Literal

We can override the default type of an integer, floating- point, or character literal by supplying a suffix or prefix as listed in Table 2.2.

```
L'a'      // wide character literal, type is wchar_t
u8"hi!"   // utf-8 string literal (utf-8 encodes a Unicode character in 8 bits)
42ULL     // unsigned integer literal, type is unsigned long long
1E-3F     // single-precision floating-point literal, type is float
3.14159L  // extended-precision floating-point literal, type is long double
```

When you write a `long` literal, use the uppercase `L`; the lowercase letter `l` is too easily mistaken for the digit 1.

Table 2.2: Specifying the Type of a Literal

Character and Character String Literals

Prefix	Meaning	Type
`u`	Unicode 16 character	`char16_t`
`U`	Unicode 32 character	`char32_t`
`L`	wide character	`wchar_t`
`u8`	utf-8 (string literals only)	`char`

Integer Literals		Floating-Point Literals	
Suffix	**Minimum Type**	**Suffix**	**Type**
`u` or `U`	`unsigned`	`f` or `F`	`float`
`l` or `L`	`long`	`l` or `L`	`long double`
`ll` or `LL`	`long long`		

We can independently specify the signedness and size of an integral literal. If the suffix contains a `U`, then the literal has an unsigned type, so a decimal, octal, or hexadecimal literal with a `U` suffix has the smallest type of `unsigned int`, `unsigned long`, or `unsigned long long` in which the literal's value fits. If the suffix contains an `L`, then the literal's type will be at least `long`; if the suffix contains `LL`, then the literal's type will be either `long long` or `unsigned long long`.

We can combine `U` with either `L` or `LL`. For example, a literal with a suffix of `UL` will be either `unsigned long` or `unsigned long long`, depending on whether its value fits in `unsigned long`.

Boolean and Pointer Literals

The words `true` and `false` are literals of type `bool`:

```
bool test = false;
```

The word `nullptr` is a pointer literal. We'll have more to say about pointers and `nullptr` in § 2.3.2 (p. 52).

Exercises Section 2.1.3

Exercise 2.5: Determine the type of each of the following literals. Explain the differences among the literals in each of the four examples:

(a) `'a', L'a', "a", L"a"`
(b) `10, 10u, 10L, 10uL, 012, 0xC`
(c) `3.14, 3.14f, 3.14L`
(d) `10, 10u, 10., 10e-2`

Exercise 2.6: What, if any, are the differences between the following definitions:

```
int month = 9, day = 7;
int month = 09, day = 07;
```

Exercise 2.7: What values do these literals represent? What type does each have?

(a) `"Who goes with F\145rgus?\012"`
(b) `3.14e1L` (c) `1024f` (d) `3.14L`

Exercise 2.8: Using escape sequences, write a program to print `2M` followed by a newline. Modify the program to print `2`, then a tab, then an `M`, followed by a newline.

2.2 Variables

A ***variable*** provides us with named storage that our programs can manipulate. Each variable in C++ has a type. The type determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as "variables" or "objects" interchangeably.

2.2.1 Variable Definitions

A simple variable definition consists of a **type specifier**, followed by a list of one or more variable names separated by commas, and ends with a semicolon. Each name

in the list has the type defined by the type specifier. A definition may (optionally) provide an initial value for one or more of the names it defines:

```
int sum = 0, value, //  sum, value, and units_sold have type int
    units_sold = 0; //  sum and units_sold have initial value 0
Sales_item item;    //  item has type Sales_item (see § 1.5.1 (p. 20))
//  string is a library type, representing a variable-length sequence of characters
std::string book("0-201-78345-X"); //  book initialized from string literal
```

The definition of `book` uses the `std::string` library type. Like `iostream` (§ 1.2, p. 7), `string` is defined in namespace `std`. We'll have more to say about the `string` type in Chapter 3. For now, what's useful to know is that a `string` is a type that represents a variable-length sequence of characters. The `string` library gives us several ways to initialize `string` objects. One of these ways is as a copy of a string literal (§ 2.1.3, p. 39). Thus, `book` is initialized to hold the characters `0-201-78345-X`.

TERMINOLOGY: WHAT IS AN OBJECT?

C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that can contain data and has a type.

Some use the term *object* only to refer to variables or values of class types. Others distinguish between named and unnamed objects, using the term *variable* to refer to named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and the term *value* for data that are read-only.

In this book, we'll follow the more general usage that an object is a region of memory that has a type. We will freely use the term *object* regardless of whether the object has built-in or class type, is named or unnamed, or can be read or written.

Initializers

An object that is **initialized** gets the specified value at the moment it is created. The values used to initialize a variable can be arbitrarily complicated expressions. When a definition defines two or more variables, the name of each object becomes visible immediately. Thus, it is possible to initialize a variable to the value of one defined earlier in the same definition.

```
//  ok: price is defined and initialized before it is used to initialize discount
double price = 109.99, discount = price * 0.16;
//  ok: call applyDiscount and use the return value to initialize salePrice
double salePrice = applyDiscount(price, discount);
```

Initialization in C++ is a surprisingly complicated topic and one we will return to again and again. Many programmers are confused by the use of the = symbol to initialize a variable. It is tempting to think of initialization as a form of assignment, but initialization and assignment are different operations in C++. This concept is particularly confusing because in many languages the distinction is irrelevant

and can be ignored. Moreover, even in C++ the distinction often doesn't matter. Nonetheless, it is a crucial concept and one we will reiterate throughout the text.

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

List Initialization

One way in which initialization is a complicated topic is that the language defines several different forms of initialization. For example, we can use any of the following four different ways to define an `int` variable named `units_sold` and initialize it to `0`:

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

The generalized use of curly braces for initialization was introduced as part of the new standard. This form of initialization previously had been allowed only in more restricted ways. For reasons we'll learn about in § 3.3.1 (p. 98), this form of initialization is referred to as **list initialization**. Braced lists of initializers can now be used whenever we initialize an object and in some cases when we assign a new value to an object.

C++ 11

When used with variables of built-in type, this form of initialization has one important property: The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information:

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; // error: narrowing conversion required
int c(ld), d = ld;   // ok: but value will be truncated
```

The compiler rejects the initializations of `a` and `b` because using a `long double` to initialize an `int` is likely to lose data. At a minimum, the fractional part of `ld` will be truncated. In addition, the integer part in `ld` might be too large to fit in an `int`.

As presented here, the distinction might seem trivial—after all, we'd be unlikely to directly initialize an `int` from a `long double`. However, as we'll see in Chapter 16, such initializations might happen unintentionally. We'll say more about these forms of initialization in § 3.2.1 (p. 84) and § 3.3.1 (p. 98).

Default Initialization

When we define a variable without an initializer, the variable is **default initialized**. Such variables are given the "default" value. What that default value is depends on the type of the variable and may also depend on where the variable is defined.

The value of an object of built-in type that is not explicitly initialized depends on where it is defined. Variables defined outside any function body are initialized to zero. With one exception, which we cover in § 6.1.1 (p. 205), variables of built-in

type defined inside a function are **uninitialized**. The value of an uninitialized variable of built-in type is undefined (§ 2.1.2, p. 36). It is an error to copy or otherwise try to access the value of a variable whose value is undefined.

Each class controls how we initialize objects of that class type. In particular, it is up to the class whether we can define objects of that type without an initializer. If we can, the class determines what value the resulting object will have.

Most classes let us define objects without explicit initializers. Such classes supply an appropriate default value for us. For example, as we've just seen, the library `string` class says that if we do not supply an initializer, then the resulting `string` is the empty string:

```
std::string empty;   //  empty implicitly initialized to the empty string
Sales_item item;     //  default-initialized Sales_item object
```

Some classes require that every object be explicitly initialized. The compiler will complain if we try to create an object of such a class with no initializer.

Uninitialized objects of built-in type defined inside a function body have undefined value. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

Exercises Section 2.2.1

Exercise 2.9: Explain the following definitions. For those that are illegal, explain what's wrong and how to correct it.

(a) `std::cin >> int input_value;`
(b) `int i = { 3.14 };`
(c) `double salary = wage = 9999.99;`
(d) `int i = 3.14;`

Exercise 2.10: What are the initial values, if any, of each of the following variables?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

2.2.2 Variable Declarations and Definitions

To allow programs to be written in logical parts, C++ supports what is commonly known as ***separate compilation***. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

When we separate a program into multiple files, we need a way to share code across those files. For example, code defined in one file may need to use a variable defined in another file. As a concrete example, consider `std::cout` and

Caution: Uninitialized Variables Cause Run-Time Problems

An uninitialized variable has an indeterminate value. Trying to use the value of an uninitialized variable is an error that is often hard to debug. Moreover, the compiler is not required to detect such errors, although most will warn about at least some uses of uninitialized variables.

What happens when we use an uninitialized variable is undefined. Sometimes, we're lucky and our program crashes as soon as we access the object. Once we track down the location of the crash, it is usually easy to see that the variable was not properly initialized. Other times, the program completes but produces erroneous results. Even worse, the results may appear correct on one run of our program but fail on a subsequent run. Moreover, adding code to the program in an unrelated location can cause what we thought was a correct program to start producing incorrect results.

We recommend initializing every object of built-in type. It is not always necessary, but it is easier and safer to provide an initializer until you can be certain it is safe to omit the initializer.

`std::cin`. These are objects defined somewhere in the standard library, yet our programs can use these objects.

To support separate compilation, C++ distinguishes between declarations and definitions. A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity.

A variable declaration specifies the type and name of a variable. A variable definition is a declaration. In addition to specifying the name and type, a definition also allocates storage and may provide the variable with an initial value.

To obtain a declaration that is not also a definition, we add the `extern` keyword and may not provide an explicit initializer:

```
extern int i;    // declares but does not define i
int j;           // declares and defines j
```

Any declaration that includes an explicit initializer is a definition. We can provide an initializer on a variable defined as `extern`, but doing so overrides the `extern`. An `extern` that has an initializer is a definition:

```
extern double pi = 3.1416; // definition
```

It is an error to provide an initializer on an `extern` inside a function.

Variables must be defined exactly once but can be declared many times.

The distinction between a declaration and a definition may seem obscure at this point but is actually important. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that variable in one—and only one—file. Other files that use that variable must declare—but not define—that variable.

We'll have more to say about how C++ supports separate compilation in § 2.6.3 (p. 76) and § 6.1.3 (p. 207).

Exercises Section 2.2.2

Exercise 2.11: Explain whether each of the following is a declaration or a definition:

(a) `extern int ix = 1024;`
(b) `int iy;`
(c) `extern int iz;`

Key Concept: Static Typing

C++ is a *statically typed* language, which means that types are checked at compile time. The process by which types are checked is referred to as *type checking*.

As we've seen, the type of an object constrains the operations that the object can perform. In C++, the compiler checks whether the operations we write are supported by the types we use. If we try to do things that the type does not support, the compiler generates an error message and does not produce an executable file.

As our programs get more complicated, we'll see that static type checking can help find bugs. However, a consequence of static checking is that the type of every entity we use must be known to the compiler. As one example, we must declare the type of a variable before we can use that variable.

2.2.3 Identifiers

Identifiers in C++ can be composed of letters, digits, and the underscore character. The language imposes no limit on name length. Identifiers must begin with either a letter or an underscore. Identifiers are case-sensitive; upper- and lowercase letters are distinct:

```
// defines four different int variables
int somename, someName, SomeName, SOMENAME;
```

The language reserves a set of names, listed in Tables 2.3 and Table 2.4, for its own use. These names may not be used as identifiers.

The standard also reserves a set of names for use in the standard library. The identifiers we define in our own programs may not contain two consecutive underscores, nor can an identifier begin with an underscore followed immediately by an uppercase letter. In addition, identifiers defined outside a function may not begin with an underscore.

Conventions for Variable Names

There are a number of generally accepted conventions for naming variables. Following these conventions can improve the readability of a program.

- An identifier should give some indication of its meaning.
- Variable names normally are lowercase—`index`, not `Index` or `INDEX`.
- Like `Sales_item`, classes we define usually begin with an uppercase letter.
- Identifiers with multiple words should visually distinguish each word, for example, `student_loan` or `studentLoan`, not `studentloan`.

Naming conventions are most useful when followed consistently.

Table 2.3: C++ Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Table 2.4: C++ Alternative Operator Names

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

EXERCISES SECTION 2.2.3

Exercise 2.12: Which, if any, of the following names are invalid?

(a) `int double = 3.14;` (b) `int _;`
(c) `int catch-22;` (d) `int 1_or_2 = 1;`
(e) `double Double = 3.14;`

2.2.4 Scope of a Name

At any particular point in a program, each name that is in use refers to a specific entity—a variable, function, type, and so on. However, a given name can be reused to refer to different entities at different points in the program.

A **scope** is a part of the program in which a name has a particular meaning. Most scopes in C++ are delimited by curly braces.

The same name can refer to different entities in different scopes. Names are visible from the point where they are declared until the end of the scope in which the declaration appears.

As an example, consider the program from § 1.4.2 (p. 13):

```
#include <iostream>
int main()
{
    int sum = 0;
    //  sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val;  //  equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

This program defines three names—`main`, `sum`, and `val`—and uses the namespace name `std`, along with two names from that namespace—`cout` and `endl`.

The name `main` is defined outside any curly braces. The name `main`—like most names defined outside a function—has **global scope**. Once declared, names at the global scope are accessible throughout the program. The name `sum` is defined within the scope of the block that is the body of the `main` function. It is accessible from its point of declaration throughout the rest of the `main` function but not outside of it. The variable `sum` has **block scope**. The name `val` is defined in the scope of the `for` statement. It can be used in that statement but not elsewhere in `main`.

ADVICE: DEFINE VARIABLES WHERE YOU FIRST USE THEM

It is usually a good idea to define an object near the point at which the object is first used. Doing so improves readability by making it easy to find the definition of the variable. More importantly, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.

Nested Scopes

Scopes can contain other scopes. The contained (or nested) scope is referred to as an **inner scope**, the containing scope is the **outer scope**.

Once a name has been declared in a scope, that name can be used by scopes nested inside that scope. Names declared in the outer scope can also be redefined in an inner scope:

```
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42;  // reused has global scope
int main()
{
    int unique = 0; // unique has block scope
    // output #1: uses global reused; prints 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << " " <<  unique << std::endl;
    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << " " <<  unique << std::endl;
    return 0;
}
```

Output #1 appears before the local definition of `reused`. Therefore, this output statement uses the name `reused` that is defined in the global scope. This statement prints `42 0`. Output #2 occurs after the local definition of `reused`. The local `reused` is now **in scope**. Thus, this second output statement uses the local object named `reused` rather than the global one and prints `0 0`. Output #3 uses the scope operator (§ 1.2, p. 8) to override the default scoping rules. The global scope has no name. Hence, when the scope operator has an empty left-hand side, it is a request to fetch the name on the right-hand side from the global scope. Thus, this expression uses the global `reused` and prints `42 0`.

It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

EXERCISES SECTION 2.2.4

Exercise 2.13: What is the value of `j` in the following program?

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
```

Exercise 2.14: Is the following program legal? If so, what values are printed?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
     sum += i;
std::cout << i << " " << sum << std::endl;
```

2.3 Compound Types

A **compound type** is a type that is defined in terms of another type. C++ has several compound types, two of which—references and pointers—we'll cover in this chapter.

Defining variables of compound type is more complicated than the declarations we've seen so far. In § 2.2 (p. 41) we said that simple declarations consist of a type followed by a list of variable names. More generally, a declaration is a **base type** followed by a list of **declarators**. Each declarator names a variable and gives the variable a type that is related to the base type.

The declarations we have seen so far have declarators that are nothing more than variable names. The type of such variables is the base type of the declaration. More complicated declarators specify variables with compound types that are built from the base type of the declaration.

2.3.1 References

The new standard introduced a new kind of reference: an "rvalue reference," which we'll cover in § 13.6.1 (p. 532). These references are primarily intended for use inside classes. Technically speaking, when we use the term *reference*, we mean "lvalue reference."

A **reference** defines an alternative name for an object. A reference type "refers to" another type. We define a reference type by writing a declarator of the form `&d`, where `d` is the name being declared:

```
int ival = 1024;
int &refVal = ival;   //  refVal refers to (is another name for) ival
int &refVal2;         //  error: a reference must be initialized
```

Ordinarily, when we initialize a variable, the value of the initializer is copied into the object we are creating. When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references *must* be initialized.

A Reference Is an Alias

A reference is not an object. Instead, a reference is *just another name for an already existing object*.

After a reference has been defined, *all* operations on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2;       //  assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal; //  same as ii = ival
```

When we assign to a reference, we are assigning to the object to which the reference is bound. When we fetch the value of a reference, we are really fetching the value of the object to which the reference is bound. Similarly, when we use a reference as an initializer, we are really using the object to which the reference is bound:

```
// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;
// initializes i from the value in the object to which refVal is bound
int i = refVal;  // ok: initializes i to the same value as ival
```

Because references are not objects, we may not define a reference to a reference.

Reference Definitions

We can define multiple references in a single definition. Each identifier that is a reference must be preceded by the & symbol:

```
int i = 1024, i2 = 2048;  // i and i2 are both ints
int &r = i, r2 = i2;      // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3;  // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2;   // both r3 and r4 are references
```

With two exceptions that we'll cover in § 2.4.1 (p. 61) and § 15.2.3 (p. 601), the type of a reference and the object to which the reference refers must match exactly. Moreover, for reasons we'll explore in § 2.4.1, a reference may be bound only to an object, not to a literal or to the result of a more general expression:

```
int &refVal4 = 10;    // error: initializer must be an object
double dval = 3.14;
int &refVal5 = dval; // error: initializer must be an int object
```

EXERCISES SECTION 2.3.1

Exercise 2.15: Which of the following definitions, if any, are invalid? Why?

```
(a) int ival = 1.01;      (b) int &rval1 = 1.01;
(c) int &rval2 = ival;    (d) int &rval3;
```

Exercise 2.16: Which, if any, of the following assignments are invalid? If they are valid, explain what they do.

```
int i = 0, &r1 = i; double d = 0, &r2 = d;
(a) r2 = 3.14159;     (b) r2 = r1;
(c) i = r2;           (d) r1 = d;
```

Exercise 2.17: What does the following code print?

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

2.3.2 Pointers

A **pointer** is a compound type that "points to" another type. Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

Pointers are often hard to understand. Debugging problems due to pointer errors bedevil even experienced programmers.

We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable:

```
int *ip1, *ip2;  // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

Taking the Address of an Object

A pointer holds the address of another object. We get the address of an object by usin the address-of operator (the **`&` operator**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
```

The second statement defines `p` as a pointer to `int` and initializes `p` to point to the `int` object named `ival`. Because references are not objects, they don't have addresses. Hence, we may not define a pointer to a reference.

With two exceptions, which we cover in § 2.4.2 (p. 62) and § 15.2.3 (p. 601), the types of the pointer and the object to which it points must match:

```
double dval;
double *pd = &dval;  // ok: initializer is the address of a double
double *pd2 = pd;    // ok: initializer is a pointer to double
int *pi = pd;  // error: types of pi and pd differ
pi = &dval;    // error: assigning the address of a double to a pointer to int
```

The types must match because the type of the pointer is used to infer the type of the object to which the pointer points. If a pointer addressed an object of another type, operations performed on the underlying object would fail.

Pointer Value

The value (i.e., the address) stored in a pointer can be in one of four states:

1. It can point to an object.
2. It can point to the location just immediately past the end of an object.
3. It can be a null pointer, indicating that it is not bound to any object.
4. It can be invalid; values other than the preceding three are invalid.

It is an error to copy or otherwise try to access the value of an invalid pointer. As when we use an uninitialized variable, this error is one that the compiler is unlikely to detect. The result of accessing an invalid pointer is undefined. Therefore, we must always know whether a given pointer is valid.

Although pointers in cases 2 and 3 are valid, there are limits on what we can do with such pointers. Because these pointers do not point to any object, we may not use them to access the (supposed) object to which the pointer points. If we do attempt to access an object through such pointers, the behavior is undefined.

Using a Pointer to Access an Object

When a pointer points to an object, we can use the dereference operator (the `*` **operator**) to access that object:

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p;     // * yields the object to which p points; prints 42
```

Dereferencing a pointer yields the object to which the pointer points. We can assign to that object by assigning to the result of the dereference:

```
*p = 0;       // * yields the object; we assign a new value to ival through p
cout << *p; // prints 0
```

When we assign to `*p`, we are assigning to the object to which `p` points.

We may dereference only a valid pointer that points to an object.

KEY CONCEPT: SOME SYMBOLS HAVE MULTIPLE MEANINGS

Some symbols, such as `&` and `*`, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

```
int i = 42;
int &r = i;    // & follows a type and is part of a declaration; r is a reference
int *p;        // * follows a type and is part of a declaration; p is a pointer
p = &i;        // & is used in an expression as the address-of operator
*p = i;        // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

In declarations, `&` and `*` are used to form compound types. In expressions, these same symbols are used to denote an operator. Because the same symbol is used with very different meanings, it can be helpful to ignore appearances and think of them as if they were different symbols.

Null Pointers

A **null pointer** does not point to any object. Code can check whether a pointer is null before attempting to use it. There are several ways to obtain a null pointer:

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0;       // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL;    // equivalent to int *p3 = 0;
```

C++ 11

The most direct approach is to initialize the pointer using the literal **`nullptr`**, which was introduced by the new standard. `nullptr` is a literal that has a special type that can be converted (§ 2.1.2, p. 35) to any other pointer type. Alternatively, we can initialize a pointer to the literal `0`, as we do in the definition of `p2`.

Older programs sometimes use a **preprocessor variable** named `NULL`, which the `cstdlib` header defines as `0`.

We'll describe the preprocessor in a bit more detail in § 2.6.3 (p. 77). What's useful to know now is that the preprocessor is a program that runs before the compiler. Preprocessor variables are managed by the preprocessor, and are not part of the `std` namespace. As a result, we refer to them directly without the `std::` prefix.

When we use a preprocessor variable, the preprocessor automatically replaces the variable by its value. Hence, initializing a pointer to `NULL` is equivalent to initializing it to `0`. Modern C++ programs generally should avoid using `NULL` and use `nullptr` instead.

It is illegal to assign an `int` variable to a pointer, even if the variable's value happens to be `0`.

```
int zero = 0;
pi = zero;          // error: cannot assign an int to a pointer
```

ADVICE: INITIALIZE ALL POINTERS

Uninitialized pointers are a common source of run-time errors.

As with any other uninitialized variable, what happens when we use an uninitialized pointer is undefined. Using an uninitialized pointer almost always results in a run-time crash. However, debugging the resulting crashes can be surprisingly hard.

Under most compilers, when we use an uninitialized pointer, the bits in the memory in which the pointer resides are used as an address. Using an uninitialized pointer is a request to access a supposed object at that supposed location. There is no way to distinguish a valid address from an invalid one formed from the bits that happen to be in the memory in which the pointer was allocated.

Our recommendation to initialize all variables is particularly important for pointers. If possible, define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to `nullptr` or zero. That way, the program can detect that the pointer does not point to an object.

Assignment and Pointers

Both pointers and references give indirect access to other objects. However, there are important differences in how they do so. The most important is that a reference

is not an object. Once we have defined a reference, there is no way to make that reference refer to a different object. When we use a reference, we always get the object to which the reference was initially bound.

There is no such identity between a pointer and the address that it holds. As with any other (nonreference) variable, when we assign to a pointer, we give the pointer itself a new value. Assignment makes the pointer point to a different object:

```
int i = 42;
int *pi = 0;    // pi is initialized but addresses no object
int *pi2 = &i; // pi2 initialized to hold the address of i
int *pi3;       // if pi3 is defined inside a block, pi3 is uninitialized
pi3 = pi2;          // pi3 and pi2 address the same object, e.g., i
pi2 = 0;            // pi2 now addresses no object
```

It can be hard to keep straight whether an assignment changes the pointer or the object to which the pointer points. The important thing to keep in mind is that assignment changes its left-hand operand. When we write

```
pi = &ival; // value in pi is changed; pi now points to ival
```

we assign a new value to `pi`, which changes the address that `pi` holds. On the other hand, when we write

```
*pi = 0;     // value in ival is changed; pi is unchanged
```

then `*pi` (i.e., the value to which `pi` points) is changed.

Other Pointer Operations

So long as the pointer has a valid value, we can use a pointer in a condition. Just as when we use an arithmetic value in a condition (§ 2.1.2, p. 35), if the pointer is 0, then the condition is `false`:

```
int ival = 1024;
int *pi = 0;       // pi is a valid, null pointer
int *pi2 = &ival; // pi2 is a valid pointer that holds the address of ival
if (pi)  // pi has value 0, so condition evaluates as false
    // ...
if (pi2) // pi2 points to ival, so it is not 0; the condition evaluates as true
    // ...
```

Any nonzero pointer evaluates as `true`

Given two valid pointers of the same type, we can compare them using the equality (`==`) or inequality (`!=`) operators. The result of these operators has type `bool`. Two pointers are equal if they hold the same address and unequal otherwise. Two pointers hold the same address (i.e., are equal) if they are both null, if they address the same object, or if they are both pointers one past the same object. Note that it is possible for a pointer to an object and a pointer one past the end of a different object to hold the same address. Such pointers will compare equal.

Because these operations use the value of the pointer, a pointer used in a condition or in a comparsion must be a valid pointer. Using an invalid pointer as a condition or in a comparison is undefined.

§ 3.5.3 (p. 117) will cover additional pointer operations.

`void*` Pointers

The type **`void*`** is a special pointer type that can hold the address of any object. Like any other pointer, a `void*` pointer holds an address, but the type of the object at that address is unknown:

```
double obj = 3.14, *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;  // obj can be an object of any type
pv = pd;          // pv can hold a pointer to any type
```

There are only a limited number of things we can do with a `void*` pointer: We can compare it to another pointer, we can pass it to or return it from a function, and we can assign it to another `void*` pointer. We cannot use a `void*` to operate on the object it addresses—we don't know that object's type, and the type determines what operations we can perform on the object.

Generally, we use a `void*` pointer to deal with memory as memory, rather than using the pointer to access the object stored in that memory. We'll cover using `void*` pointers in this way in § 19.1.1 (p. 821). § 4.11.3 (p. 163) will show how we can retrieve the address stored in a `void*` pointer.

Exercises Section 2.3.2

Exercise 2.18: Write code to change the value of a pointer. Write code to change the value to which the pointer points.

Exercise 2.19: Explain the key differences between pointers and references.

Exercise 2.20: What does the following program do?

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

Exercise 2.21: Explain each of the following definitions. Indicate whether any are illegal and, if so, why.

```
int i = 0;
(a) double* dp = &i;  (b) int *ip = i; (c) int *p = &i;
```

Exercise 2.22: Assuming `p` is a pointer to `int`, explain the following code:

```
if (p) // ...
if (*p) // ...
```

Exercise 2.23: Given a pointer `p`, can you determine whether `p` points to a valid object? If so, how? If not, why not?

Exercise 2.24: Why is the initialization of `p` legal but that of `lp` illegal?

```
int i = 42;    void *p = &i;     long *lp = &i;
```

2.3.3 Understanding Compound Type Declarations

As we've seen, a variable definition consists of a base type and a list of declarators. Each declarator can relate its variable to the base type differently from the other declarators in the same definition. Thus, a single definition might define variables of different types:

```
// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
```

Many programmers are confused by the interaction between the base type and the type modification that may be part of a declarator.

Defining Multiple Variables

It is a common misconception to think that the type modifier (`*` or `&`) applies to all the variables defined in a single statement. Part of the problem arises because we can put whitespace between the type modifier and the name being declared:

```
int* p;   // legal but might be misleading
```

We say that this definition might be misleading because it suggests that `int*` is the type of each variable declared in that statement. Despite appearances, the base type of this declaration is `int`, not `int*`. The `*` modifies the type of `p`. It says nothing about any other objects that might be declared in the same statement:

```
int* p1, p2; // p1 is a pointer to int; p2 is an int
```

There are two common styles used to define multiple variables with pointer or reference type. The first places the type modifier adjacent to the identifier:

```
int *p1, *p2; // both p1 and p2 are pointers to int
```

This style emphasizes that the variable has the indicated compound type.

The second places the type modifier with the type but defines only one variable per statement:

```
int* p1;      // p1 is a pointer to int
int* p2;      // p2 is a pointer to int
```

This style emphasizes that the declaration defines a compound type.

There is no single right way to define pointers or references. The important thing is to choose a style and use it consistently.

In this book we use the first style and place the `*` (or the `&`) with the variable name.

Pointers to Pointers

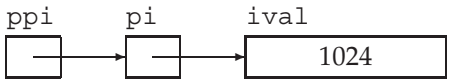
In general, there are no limits to how many type modifiers can be applied to a declarator. When there is more than one modifier, they combine in ways that are logical but not always obvious. As one example, consider a pointer. A pointer is

an object in memory, so like any object it has an address. Therefore, we can store the address of a pointer in another pointer.

We indicate each pointer level by its own `*`. That is, we write `**` for a pointer to a pointer, `***` for a pointer to a pointer to a pointer, and so on:

```
int ival = 1024;
int *pi = &ival;    // pi points to an int
int **ppi = &pi;    // ppi points to a pointer to an int
```

Here `pi` is a pointer to an `int` and `ppi` is a pointer to a pointer to an `int`. We might represent these objects as



Just as dereferencing a pointer to an `int` yields an `int`, dereferencing a pointer to a pointer yields a pointer. To access the underlying object, we must dereference the original pointer twice:

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

This program prints the value of `ival` three different ways: first, directly; then, through the pointer to `int` in `pi`; and finally, by dereferencing `ppi` twice to get to the underlying value in `ival`.

References to Pointers

A reference is not an object. Hence, we may not have a pointer to a reference. However, because a pointer is an object, we can define a reference to a pointer:

```
int i = 42;
int *p;          // p is a pointer to int
int *&r = p;     // r is a reference to the pointer p
r = &i; // r refers to a pointer; assigning &i to r makes p point to i
*r = 0; // dereferencing r yields i, the object to which p points; changes i to 0
```

The easiest way to understand the type of `r` is to read the definition right to left. The symbol closest to the name of the variable (in this case the `&` in `&r`) is the one that has the most immediate effect on the variable's type. Thus, we know that `r` is a reference. The rest of the declarator determines the type to which `r` refers. The next symbol, `*` in this case, says that the type `r` refers to is a pointer type. Finally, the base type of the declaration says that `r` is a reference to a pointer to an `int`.

It can be easier to understand complicated pointer or reference declarations if you read them from right to left.

EXERCISES SECTION 2.3.3

Exercise 2.25: Determine the types and values of each of the following variables.

(a) `int* ip, i, &r = i;` (b) `int i, *ip = 0;` (c) `int* ip, ip2;`

2.4 const Qualifier

Sometimes we want to define a variable whose value we know cannot be changed. For example, we might want to use a variable to refer to the size of a buffer size. Using a variable makes it easy for us to change the size of the buffer if we decided the original size wasn't what we needed. On the other hand, we'd also like to prevent code from inadvertently giving a new value to the variable we use to represent the buffer size. We can make a variable unchangeable by defining the variable's type as **const**:

```
const int bufSize = 512;     // input buffer size
```

defines `bufSize` as a constant. Any attempt to assign to `bufSize` is an error:

```
bufSize = 512; // error: attempt to write to const object
```

Because we can't change the value of a `const` object after we create it, it must be initialized. As usual, the initializer may be an arbitrarily complicated expression:

```
const int i = get_size();  // ok: initialized at run time
const int j = 42;          // ok: initialized at compile time
const int k;               // error: k is uninitialized const
```

Initialization and const

As we have observed many times, the type of an object defines the operations that can be performed by that object. A `const` type can use most but not all of the same operations as its nonconst version. The one restriction is that we may use only those operations that cannot change an object. So, for example, we can use a `const int` in arithmetic expressions in exactly the same way as a plain, nonconst `int`. A `const int` converts to `bool` the same way as a plain `int`, and so on.

Among the operations that don't change the value of an object is initialization—when we use an object to initialize another object, it doesn't matter whether either or both of the objects are `const`s:

```
int i = 42;
const int ci = i;     // ok: the value in i is copied into ci
int j = ci;           // ok: the value in ci is copied into j
```

Although `ci` is a `const int`, the value in `ci` is an `int`. The `const`ness of `ci` matters only for operations that might change `ci`. When we copy `ci` to initialize `j`, we don't care that `ci` is a `const`. Copying an object doesn't change that object. Once the copy is made, the new object has no further access to the original object.

By Default, `const` Objects Are Local to a File

When a `const` object is initialized from a compile-time constant, such as in our definition of `bufSize`:

```
const int bufSize = 512;     // input buffer size
```

the compiler will usually replace uses of the variable with its corresponding value during compilation. That is, the compiler will generate code using the value `512` in the places that our code uses `bufSize`.

To substitute the value for the variable, the compiler has to see the variable's initializer. When we split a program into multiple files, every file that uses the `const` must have access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value (§ 2.2.2, p. 45). To support this usage, yet avoid multiple definitions of the same variable, `const` variables are defined as local to the file. When we define a `const` with the same name in multiple files, it is as if we had written definitions for separate variables in each file.

Sometimes we have a `const` variable that we want to share across multiple files but whose initializer is not a constant expression. In this case, we don't want the compiler to generate a separate variable in each file. Instead, we want the `const` object to behave like other (nonconst) variables. We want to define the `const` in one file, and declare it in the other files that use that object.

To define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration(s):

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();

// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

In this program, `file_1.cc` defines and initializes `bufSize`. Because this declaration includes an initializer, it is (as usual) a definition. However, because `bufSize` is `const`, we must specify `extern` in order for `bufSize` to be used in other files.

The declaration in `file_1.h` is also `extern`. In this case, the `extern` signifies that `bufSize` is not local to this file and that its definition will occur elsewhere.

To share a `const` object among multiple files, you must define the variable as `extern`.

Exercises Section 2.4

Exercise 2.26: Which of the following are legal? For those that are illegal, explain why.

(a) `const int buf;` (b) `int cnt = 0;`
(c) `const int sz = cnt;` (d) `++cnt; ++sz;`

2.4.1 References to const

As with any other object, we can bind a reference to an object of a const type. To do so we use a **reference to const**, which is a reference that refers to a const type. Unlike an ordinary reference, a reference to const cannot be used to change the object to which the reference is bound:

```
const int ci = 1024;
const int &r1 =  ci;   // ok: both reference and underlying object are const
r1 = 42;               // error: r1 is a reference to const
int &r2 = ci;          // error: nonconst reference to a const object
```

Because we cannot assign directly to ci, we also should not be able to use a reference to change ci. Therefore, the initialization of r2 is an error. If this initialization were legal, we could use r2 to change the value of its underlying object.

TERMINOLOGY: const REFERENCE IS A REFERENCE TO const

C++ programmers tend to abbreviate the phrase "reference to const" as "const reference." This abbreviation makes sense—if you remember that it is an abbreviation.

Technically speaking, there are no const references. A reference is not an object, so we cannot make a reference itself const. Indeed, because there is no way to make a reference refer to a different object, in some sense all references are const. Whether a reference refers to a const or nonconst type affects what we can do with that reference, not whether we can alter the binding of the reference itself.

Initialization and References to const

In § 2.3.1 (p. 51) we noted that there are two exceptions to the rule that the type of a reference must match the type of the object to which it refers. The first exception is that we can initialize a reference to const from any expression that can be converted (§ 2.1.2, p. 35) to the type of the reference. In particular, we can bind a reference to const to a nonconst object, a literal, or a more general expression:

```
int i = 42;
const int &r1 = i;       // we can bind a const int& to a plain int object
const int &r2 = 42;      // ok: r1 is a reference to const
const int &r3 = r1 * 2;  // ok: r3 is a reference to const
int &r4 = r * 2;         // error: r4 is a plain, nonconst reference
```

The easiest way to understand this difference in initialization rules is to consider what happens when we bind a reference to an object of a different type:

```
double dval = 3.14;
const int &ri = dval;
```

Here ri refers to an int. Operations on ri will be integer operations, but dval is a floating-point number, not an integer. To ensure that the object to which ri is bound is an int, the compiler transforms this code into something like

```
const int temp = dval;    // create a temporary const int from the double
const int &ri = temp;     // bind ri to that temporary
```

In this case, `ri` is bound to a **temporary** object. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression. C++ programmers often use the word temporary as an abbreviation for temporary object.

Now consider what could happen if this initialization were allowed but `ri` was not `const`. If `ri` weren't `const`, we could assign to `ri`. Doing so would change the object to which `ri` is bound. That object is a temporary, not `dval`. The programmer who made `ri` refer to `dval` would probably expect that assigning to `ri` would change `dval`. After all, why assign to `ri` unless the intent is to change the object to which `ri` is bound? Because binding a reference to a temporary is almost surely *not* what the programmer intended, the language makes it illegal.

A Reference to `const` May Refer to an Object That Is Not `const`

It is important to realize that a reference to `const` restricts only what we can do through that reference. Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`. Because the underlying object might be nonconst, it might be changed by other means:

```
int i = 42;
int &r1 = i;           // r1 bound to i
const int &r2 = i;     // r2 also bound to i; but cannot be used to change i
r1 = 0;                // r1 is not const; i is now 0
r2 = 0;                // error: r2 is a reference to const
```

Binding `r2` to the (nonconst) `int` `i` is legal. However, we cannot use `r2` to change `i`. Even so, the value in `i` still might change. We can change `i` by assigning to it directly, or by assigning to another reference bound to `i`, such as `r1`.

2.4.2 Pointers and `const`

As with references, we can define pointers that point to either `const` or nonconst types. Like a reference to `const`, a **pointer to `const`** (§ 2.4.1, p. 61) may not be used to change the object to which the pointer points. We may store the address of a `const` object only in a pointer to `const`:

```
const double pi = 3.14;    // pi is const; its value may not be changed
double *ptr = &pi;         // error: ptr is a plain pointer
const double *cptr = &pi;  // ok: cptr may point to a double that is const
*cptr = 42;                // error: cannot assign to *cptr
```

In § 2.3.2 (p. 52) we noted that there are two exceptions to the rule that the types of a pointer and the object to which it points must match. The first exception is that we can use a pointer to `const` to point to a nonconst object:

```
double dval = 3.14;        // dval is a double; its value can be changed
cptr = &dval;              // ok: but can't change dval through cptr
```

Like a reference to const, a pointer to const says nothing about whether the object to which the pointer points is const. Defining a pointer as a pointer to const affects only what we can do with the pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to const won't change.

It may be helpful to think of pointers and references to const as pointers or references "that *think* they point or refer to const."

const Pointers

Unlike references, pointers are objects. Hence, as with any other object type, we can have a pointer that is itself const. Like any other const object, a **const pointer** must be initialized, and once initialized, its value (i.e., the address that it holds) may not be changed. We indicate that the pointer is const by putting the const after the *. This placement indicates that it is the pointer, not the pointed-to type, that is const:

```
int errNumb = 0;
int *const curErr = &errNumb;  // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
```

As we saw in § 2.3.3 (p. 58), the easiest way to understand these declarations is to read them from right to left. In this case, the symbol closest to curErr is const, which means that curErr itself will be a const object. The type of that object is formed from the rest of the declarator. The next symbol in the declarator is *, which means that curErr is a const pointer. Finally, the base type of the declaration completes the type of curErr, which is a const pointer to an object of type int. Similarly, pip is a const pointer to an object of type const double.

The fact that a pointer is itself const says nothing about whether we can use the pointer to change the underlying object. Whether we can change that object depends entirely on the type to which the pointer points. For example, pip is a const pointer to const. Neither the value of the object addressed by pip nor the address stored in pip can be changed. On the other hand, curErr addresses a plain, nonconst int. We can use curErr to change the value of errNumb:

```
*pip = 2.72;      // error: pip is a pointer to const
// if the object to which curErr points (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
}
```

2.4.3 Top-Level const

As we've seen, a pointer is an object that can point to a different object. As a result, we can talk independently about whether a pointer is const and whether

EXERCISES SECTION 2.4.2

Exercise 2.27: Which of the following initializations are legal? Explain why.

```
(a) int i = -1, &r = 0;          (b) int *const p2 = &i2;
(c) const int i = -1, &r = 0;    (d) const int *const p3 = &i2;
(e) const int *p1 = &i2;         (f) const int &const r2;
(g) const int i2 = i, &r = i;
```

Exercise 2.28: Explain the following definitions. Identify any that are illegal.

```
(a) int i, *const cp;        (b) int *p1, *const p2;
(c) const int ic, &r = ic;   (d) const int *const p3;
(e) const int *p;
```

Exercise 2.29: Uing the variables in the previous exercise, which of the following assignments are legal? Explain why.

```
(a) i = ic;                  (b) p1 = p3;
(c) p1 = &ic;                (d) p3 = &ic;
(e) p2 = p1;                 (f) ic = *p3;
```

the objects to which it can point are `const`. We use the term **top-level `const`** to indicate that the pointer itself is a `const`. When a pointer can point to a `const` object, we refer to that `const` as a **low-level `const`**.

More generally, top-level `const` indicates that an object itself is `const`. Top-level `const` can appear in any object type, i.e., one of the built-in arithmetic types, a class type, or a pointer type. Low-level `const` appears in the base type of compound types such as pointers or references. Note that pointer types, unlike most other types, can have both top-level and low-level `const` independently:

```
int i = 0;
int *const p1 = &i;   // we can't change the value of p1; const is top-level
const int ci = 42;    // we cannot change ci; const is top-level
const int *p2 = &ci; // we can change p2; const is low-level
const int *const p3 = p2; // right-most const is top-level, left-most is not
const int &r = ci;   // const in reference types is always low-level
```

The distinction between top-level and low-level matters when we copy an object. When we copy an object, top-level `const`s are ignored:

```
i = ci;   // ok: copying the value of ci; top-level const in ci is ignored
p2 = p3; // ok: pointed-to type matches; top-level const in p3 is ignored
```

Copying an object doesn't change the copied object. As a result, it is immaterial whether the object copied from or copied into is `const`.

On the other hand, low-level `const` is never ignored. When we copy an object, both objects must have the same low-level `const` qualification or there must be a conversion between the types of the two objects. In general, we can convert a nonconst to `const` but not the other way round:

```
int *p = p3; // error: p3 has a low-level const but p doesn't
p2 = p3;     // ok: p2 has the same low-level const qualification as p3
p2 = &i;     // ok: we can convert int* to const int*
int &r = ci; // error: can't bind an ordinary int& to a const int object
const int &r2 = i; // ok: can bind const int& to plain int
```

p3 has both a top-level and low-level const. When we copy p3, we can ignore its top-level const but not the fact that it points to a const type. Hence, we cannot use p3 to initialize p, which points to a plain (nonconst) int. On the other hand, we can assign p3 to p2. Both pointers have the same (low-level const) type. The fact that p3 is a const pointer (i.e., that it has a top-level const) doesn't matter.

EXERCISES SECTION 2.4.3

Exercise 2.30: For each of the following declarations indicate whether the object being declared has top-level or low-level const.

```
const int v2 = 0;    int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

Exercise 2.31: Given the declarations in the previous exercise determine whether the following assignments are legal. Explain how the top-level or low-level const applies in each case.

```
r1 = v2;
p1 = p2;    p2 = p1;
p1 = p3;    p2 = p3;
```

2.4.4 constexpr and Constant Expressions

A **constant expression** is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A const object that is initialized from a constant expression is also a constant expression. As we'll see, there are several contexts in the language that require constant expressions.

Whether a given object (or expression) is a constant expression depends on the types and the initializers. For example:

```
const int max_files = 20;     // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27;        // staff_size is not a constant expression
const int sz = get_size(); // sz is not a constant expression
```

Although staff_size is initialized from a literal, it is not a constant expression because it is a plain int, not a const int. On the other hand, even though sz is a const, the value of its initializer is not known until run time. Hence, sz is not a constant expression.

`constexpr` Variables

In a large system, it can be difficult to determine (for certain) that an initializer is a constant expression. We might define a `const` variable with an initializer that we think is a constant expression. However, when we use that variable in a context that requires a constant expression we may discover that the initializer was not a constant expression. In general, the definition of an object and its use in such a context can be widely separated.

C++ 11

Under the new standard, we can ask the compiler to verify that a variable is a constant expression by declaring the variable in a **`constexpr`** declaration. Variables declared as `constexpr` are implicitly `const` and must be initialized by constant expressions:

```
constexpr int mf = 20;          //  20 is a constant expression
constexpr int limit = mf + 1; //  mf + 1 is a constant expression
constexpr int sz = size();     //  ok only if size is a constexpr function
```

Although we cannot use an ordinary function as an initializer for a `constexpr` variable, we'll see in § 6.5.2 (p. 239) that the new standard lets us define certain functions as `constexpr`. Such functions must be simple enough that the compiler can evaluate them at compile time. We can use `constexpr` functions in the initializer of a `constexpr` variable.

Generally, it is a good idea to use `constexpr` for variables that you intend to use as constant expressions.

Literal Types

Because a constant expression is one that can be evaluated at compile time, there are limits on the types that we can use in a `constexpr` declaration. The types we can use in a `constexpr` are known as "literal types" because they are simple enough to have literal values.

Of the types we have used so far, the arithmetic, reference, and pointer types are literal types. Our `Sales_item` class and the library IO and `string` types are not literal types. Hence, we cannot define variables of these types as `constexpr`s. We'll see other kinds of literal types in § 7.5.6 (p. 299) and § 19.3 (p. 832).

Although we can define both pointers and reference as `constexpr`s, the objects we use to initialize them are strictly limited. We can initialize a `constexpr` pointer from the `nullptr` literal or the literal (i.e., constant expression) `0`. We can also point to (or bind to) an object that remains at a fixed address.

For reasons we'll cover in § 6.1.1 (p. 204), variables defined inside a function ordinarily are not stored at a fixed address. Hence, we cannot use a `constexpr` pointer to point to such variables. On the other hand, the address of an object defined outside of any function is a constant expression, and so may be used to initialize a `constexpr` pointer. We'll see in § 6.1.1 (p. 205), that functions may define variables that exist across calls to that function. Like an object defined outside any function, these special local objects also have fixed addresses. Therefore, a `constexpr` reference may be bound to, and a `constexpr` pointer may address, such variables.

Pointers and `constexpr`

It is important to understand that when we define a pointer in a `constexpr` declaration, the `constexpr` specifier applies to the pointer, not the type to which the pointer points:

```
const int *p = nullptr;     // p is a pointer to a const int
constexpr int *q = nullptr; // q is a const pointer to int
```

Despite appearances, the types of `p` and `q` are quite different; `p` is a pointer to `const`, whereas `q` is a constant pointer. The difference is a consequence of the fact that `constexpr` imposes a top-level `const` (§ 2.4.3, p. 63) on the objects it defines.

Like any other constant pointer, a `constexpr` pointer may point to a `const` or a nonconst type:

```
constexpr int *np = nullptr; // np is a constant pointer to int that is null
int j = 0;
constexpr int i = 42;  // type of i is const int
// i and j must be defined outside any function
constexpr const int *p = &i; // p is a constant pointer to the const int i
constexpr int *p1 = &j;      // p1 is a constant pointer to the int j
```

EXERCISES SECTION 2.4.4

Exercise 2.32: Is the following code legal or not? If not, how might you make it legal?

```
int null = 0, *p = null;
```

2.5 Dealing with Types

As our programs get more complicated, we'll see that the types we use also get more complicated. Complications in using types arise in two different ways. Some types are hard to "spell." That is, they have forms that are tedious and error-prone to write. Moreover, the form of a complicated type can obscure its purpose or meaning. The other source of complication is that sometimes it is hard to determine the exact type we need. Doing so can require us to look back into the context of the program.

2.5.1 Type Aliases

A **type alias** is a name that is a synonym for another type. Type aliases let us simplify complicated type definitions, making those types easier to use. Type aliases also let us emphasize the purpose for which a type is used.

We can define a type alias in one of two ways. Traditionally, we use a **typedef**:

```
typedef double wages;     // wages is a synonym for double
typedef wages base, *p;   // base is a synonym for double, p for double*
```

The keyword `typedef` may appear as part of the base type of a declaration (§ 2.3, p. 50). Declarations that include `typedef` define type aliases rather than variables. As in any other declaration, the declarators can include type modifiers that define compound types built from the base type of the definition.

C++ 11

The new standard introduced a second way to define a type alias, via an **alias declaration**:

```
using SI = Sales_item;    // SI is a synonym for Sales_item
```

An alias declaration starts with the keyword `using` followed by the alias name and an =. The alias declaration defines the name on the left-hand side of the = as an alias for the type that appears on the right-hand side.

A type alias is a type name and can appear wherever a type name can appear:

```
wages hourly, weekly;     // same as double hourly, weekly;
SI item;                  // same as Sales_item item
```

Pointers, `const`, and Type Aliases

Declarations that use type aliases that represent compound types and `const` can yield surprising results. For example, the following declarations use the type `pstring`, which is an alias for the the type `char*`:

```
typedef char *pstring;
const pstring cstr = 0; //  cstr is a constant pointer to char
const pstring *ps;      //  ps is a pointer to a constant pointer to char
```

The base type in these declarations is `const pstring`. As usual, a `const` that appears in the base type modifies the given type. The type of `pstring` is "pointer to `char`." So, `const pstring` is a constant pointer to `char`—not a pointer to `const char`.

It can be tempting, albeit incorrect, to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type:

```
const char *cstr = 0;   // wrong interpretation of const pstring cstr
```

However, this interpretation is wrong. When we use `pstring` in a declaration, the base type of the declaration is a pointer type. When we rewrite the declaration using `char*`, the base type is `char` and the `*` is part of the declarator. In this case, `const char` is the base type. This rewrite declares `cstr` as a pointer to `const char` rather than as a `const` pointer to `char`.

2.5.2 The `auto` Type Specifier

It is not uncommon to want to store the value of an expression in a variable. To declare the variable, we have to know the type of that expression. When we write a program, it can be surprisingly difficult—and sometimes even impossible—to determine the type of an expression. Under the new standard, we can let the compiler figure out the type for us by using the **`auto`** type specifier. Unlike type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce

C++ 11

the type from the initializer. By implication, a variable that uses `auto` as its type specifier must have an initializer:

```
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1 + val2
```

Here the compiler will deduce the type of `item` from the type returned by applying `+` to `val1` and `val2`. If `val1` and `val2` are `Sales_item` objects (§ 1.5, p. 19), `item` will have type `Sales_item`. If those variables are type `double`, then `item` has type `double`, and so on.

As with any other type specifier, we can define multiple variables using `auto`. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other:

```
auto i = 0, *p = &i;      // ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14;   // error: inconsistent types for sz and pi
```

Compound Types, `const`, and `auto`

The type that the compiler infers for `auto` is not always exactly the same as the initializer's type. Instead, the compiler adjusts the type to conform to normal initialization rules.

First, as we've seen, when we use a reference, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r;  // a is an int (r is an alias for i, which has type int)
```

Second, `auto` ordinarily ignores top-level `const`s (§ 2.4.3, p. 63). As usual in initializations, low-level `const`s, such as when an initializer is a pointer to `const`, are kept:

```
const int ci = i, &cr = ci;
auto b = ci;  // b is an int (top-level const in ci is dropped)
auto c = cr;  // c is an int (cr is an alias for ci whose const is top-level)
auto d = &i;  // d is an int* (& of an int object is int*)
auto e = &ci; // e is const int* (& of a const object is low-level const)
```

If we want the deduced type to have a top-level `const`, we must say so explicitly:

```
const auto f = ci;  // deduced type of ci is int; f has type const int
```

We can also specify that we want a reference to the `auto`-deduced type. Normal initialization rules still apply:

```
auto &g = ci;        // g is a const int& that is bound to ci
auto &h = 42;        // error: we can't bind a plain reference to a literal
const auto &j = 42; // ok: we can bind a const reference to a literal
```

When we ask for a reference to an `auto`-deduced type, top-level `const`s in the initializer are not ignored. As usual, `const`s are not top-level when we bind a reference to an initializer.

When we define several variables in the same statement, it is important to remember that a reference or pointer is part of a particular declarator and not part of the base type for the declaration. As usual, the initializers must provide consistent `auto`-deduced types:

```
auto k = ci, &l = i;     // k is int; l is int&
auto &m = ci, *p = &ci; // m is a const int&; p is a pointer to const int
// error: type deduced from i is int; type deduced from &ci is const int
auto &n = i, *p2 = &ci;
```

EXERCISES SECTION 2.5.2

Exercise 2.33: Using the variable definitions from this section, determine what happens in each of these assignments:

```
a = 42;   b = 42;   c = 42;
d = 42;   e = 42;   g = 42;
```

Exercise 2.34: Write a program containing the variables and assignments from the previous exercise. Print the variables before and after the assignments to check whether your predictions in the previous exercise were correct. If not, study the examples until you can convince yourself you know what led you to the wrong conclusion.

Exercise 2.35: Determine the types deduced in each of the following definitions. Once you've figured out the types, write a program to see whether you were correct.

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```

2.5.3 The `decltype` Type Specifier

Sometimes we want to define a variable with a type that the compiler deduces from an expression but do not want to use that expression to initialize the variable. For such cases, the new standard introduced a second type specifier, **`decltype`**, which returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression:

C++ 11

```
decltype(f()) sum = x; // sum has whatever type f returns
```

Here, the compiler does not call `f`, but it uses the type that such a call would return as the type for `sum`. That is, the compiler gives `sum` the same type as the type that would be returned if we were to call `f`.

The way `decltype` handles top-level `const` and references differs subtly from the way `auto` does. When the expression to which we apply `decltype` is a vari-

able, `decltype` returns the type of that variable, including top-level `const` and references:

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; //  x has type const int
decltype(cj) y = x; //  y has type const int& and is bound to x
decltype(cj) z;     //  error: z is a reference and must be initialized
```

Because `cj` is a reference, `decltype(cj)` is a reference type. Like any other reference, `z` must be initialized.

It is worth noting that `decltype` is the *only* context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.

`decltype` and References

When we apply `decltype` to an expression that is not a variable, we get the type that that expression yields. As we'll see in § 4.1.1 (p. 135), some expressions will cause `decltype` to yield a reference type. Generally speaking, `decltype` returns a reference type for expressions that yield objects that can stand on the left-hand side of the assignment:

```
//  decltype of an expression can be a reference type
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;  //  ok: addition yields an int; b is an (uninitialized) int
decltype(*p) c;     //  error: c is int& and must be initialized
```

Here `r` is a reference, so `decltype(r)` is a reference type. If we want the type to which `r` refers, we can use `r` in an expression, such as `r + 0`, which is an expression that yields a value that has a nonreference type.

On the other hand, the dereference operator is an example of an expression for which `decltype` returns a reference. As we've seen, when we dereference a pointer, we get the object to which the pointer points. Moreover, we can assign to that object. Thus, the type deduced by `decltype(*p)` is `int&`, not plain `int`.

Another important difference between `decltype` and `auto` is that the deduction done by `decltype` *depends on the form of its given expression*. What can be confusing is that enclosing the name of a variable in parentheses affects the type returned by `decltype`. When we apply `decltype` to a variable without any parentheses, we get the type of that variable. If we wrap the variable's name in one or more sets of parentheses, the compiler will evaluate the operand as an expression. A variable is an expression that can be the left-hand side of an assignment. As a result, `decltype` on such an expression yields a reference:

```
//  decltype of a parenthesized variable is always a reference
decltype((i)) d;    //  error: d is int& and must be initialized
decltype(i) e;      //  ok: e is an (uninitialized) int
```

Remember that `decltype((`*variable*`))` (note, double parentheses) is always a reference type, but `decltype(`*variable*`)` is a reference type only if *variable* is a reference.

Exercises Section 2.5.3

Exercise 2.36: In the following code, determine the type of each variable and the value each variable has when the code finishes:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

Exercise 2.37: Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if `i` is an `int`, then the type of the expression `i = x` is `int&`. Using that knowledge, determine the type and value of each variable in this code:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

Exercise 2.38: Describe the differences in type deduction between `decltype` and `auto`. Give an example of an expression where `auto` and `decltype` will deduce the same type and an example where they will deduce differing types.

2.6 Defining Our Own Data Structures

At the most basic level, a data structure is a way to group together related data elements and a strategy for using those data. As one example, our `Sales_item` class groups an ISBN, a count of how many copies of that book had been sold, and the revenue associated with those sales. It also provides a set of operations such as the `isbn` function and the `>>`, `<<`, `+`, and `+=` operators.

In C++ we define our own data types by defining a class. The library types `string`, `istream`, and `ostream` are all defined as classes, as is the `Sales_item` type we used in Chapter 1. C++ support for classes is extensive—in fact, Parts III and IV are largely devoted to describing class-related features. Even though the `Sales_item` class is pretty simple, we won't be able to fully define that class until we learn how to write our own operators in Chapter 14.

2.6.1 Defining the `Sales_data` Type

Although we can't yet write our `Sales_item` class, we can write a more concrete class that groups the same data elements. Our strategy for using this class is that users will be able to access the data elements directly and must implement needed operations for themselves.

Because our data structure does not support any operations, we'll name our version `Sales_data` to distinguish it from `Sales_item`. We'll define our class as follows:

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Our class begins with the keyword **`struct`**, followed by the name of the class and a (possibly empty) class body. The class body is surrounded by curly braces and forms a new scope (§ 2.2.4, p. 48). The names defined inside the class must be unique within the class but can reuse names defined outside the class.

The close curly that ends the class body must be followed by a semicolon. The semicolon is needed because we can define variables after the class body:

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;
// equivalent, but better way to define these objects
struct Sales_data { /* ... */ };
Sales_data accum, trans, *salesptr;
```

The semicolon marks the end of the (usually empty) list of declarators. Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures the code by combining the definitions of two different entities—the class and a variable—in a single statement.

It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

Class Data Members

The class body defines the **members** of the class. Our class has only **data members**. The data members of a class define the contents of the objects of that class type. Each object has its own copy of the class data members. Modifying the data members of one object does not change the data in any other `Sales_data` object.

We define data members the same way that we define normal variables: We specify a base type followed by a list of one or more declarators. Our class has three data members: a member of type `string` named `bookNo`, an `unsigned` member named `units_sold`, and a member of type `double` named `revenue`. Each `Sales_data` object will have these three data members.

Under the new standard, we can supply an **in-class initializer** for a data member. When we create objects, the in-class initializers will be used to initialize the data members. Members without an initializer are default initialized (§ 2.2.1, p. 43). Thus, when we define `Sales_data` objects, `units_sold` and `revenue` will be initialized to 0, and `bookNo` will be initialized to the empty string.

C++ 11

In-class initializers are restricted as to the form (§ 2.2.1, p. 43) we can use: They must either be enclosed inside curly braces or follow an = sign. We may not specify an in-class initializer inside parentheses.

In § 7.2 (p. 268), we'll see that C++ has a second keyword, `class`, that can be used to define our own data structures. We'll explain in that section why we use `struct` here. Until we cover additional class-related features in Chapter 7, you should use `struct` to define your own data structures.

Exercises Section 2.6.1

Exercise 2.39: Compile the following program to see what happens when you forget the semicolon after a class definition. Remember the message for future reference.

```
struct Foo { /* empty */ } // Note: no semicolon
int main()
{
    return 0;
}
```

Exercise 2.40: Write your own version of the `Sales_data` class.

2.6.2 Using the `Sales_data` Class

Unlike the `Sales_item` class, our `Sales_data` class does not provide any operations. Users of `Sales_data` have to write whatever operations they need. As an example, we'll write a version of the program from § 1.5.2 (p. 23) that printed the sum of two transactions. The input to our program will be transactions such as

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

Each transaction holds an ISBN, the count of how many books were sold, and the price at which each book was sold.

Adding Two `Sales_data` Objects

Because `Sales_data` provides no operations, we will have to write our own code to do the input, output, and addition operations. We'll assume that our `Sales_data` class is defined inside `Sales_data.h`. We'll see how to define this header in § 2.6.3 (p. 76).

Because this program will be longer than any we've written so far, we'll explain it in separate parts. Overall, our program will have the following structure:

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // code to read into data1 and data2
    // code to check whether data1 and data2 have the same ISBN
    //      and if so print the sum of data1 and data2
}
```

As in our original program, we begin by including the headers we'll need and define variables to hold the input. Note that unlike the `Sales_item` version, our new program includes the `string` header. We need that header because our code will have to manage the `bookNo` member, which has type `string`.

Reading Data into a `Sales_data` Object

Although we won't describe the library `string` type in detail until Chapters 3 and 10, we need to know only a little bit about `string`s in order to define and use our ISBN member. The `string` type holds a sequence of characters. Its operations include the `>>`, `<<`, and `==` operators to read, write, and compare `string`s, respectively. With this knowledge we can write the code to read the first transaction:

```
double price = 0;   // price per book, used to calculate total revenue
// read the first transactions: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;
// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;
```

Our transactions contain the price at which each book was sold but our data structure stores the total revenue. We'll read the transaction data into a `double` named `price`, from which we'll calculate the `revenue` member. The input statement

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

uses the dot operator (§ 1.5.2, p. 23) to read into the `bookNo` and `units_sold` members of the object named `data1`.

The last statement assigns the product of `data1.units_sold` and `price` into the `revenue` member of `data1`.

Our program will next repeat the same code to read data into `data2`:

```
// read the second transaction
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

Printing the Sum of Two `Sales_data` Objects

Our other task is to check that the transactions are for the same ISBN. If so, we'll print their sum, otherwise, we'll print an error message:

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // print: ISBN, total sold, total revenue, average price per book
    std::cout << data1.bookNo << " " << totalCnt
              << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout  << "(no sales)" << std::endl;
    return 0;  // indicate success
} else {  // transactions weren't for the same ISBN
    std::cerr << "Data must refer to the same ISBN"
              << std::endl;
    return -1; // indicate failure
}
```

In the first `if` we compare the `bookNo` members of `data1` and `data2`. If those members are the same ISBN, we execute the code inside the curly braces. That code adds the components of our two variables. Because we'll need to print the average price, we start by computing the total of `units_sold` and `revenue` and store those in `totalCnt` and `totalRevenue`, respectively. We print those values. Next we check that there were books sold and, if so, print the computed average price per book. If there were no sales, we print a message noting that fact.

Exercises Section 2.6.2

Exercise 2.41: Use your `Sales_data` class to rewrite the exercises in § 1.5.1 (p. 22), § 1.5.2 (p. 24), and § 1.6 (p. 25). For now, you should define your `Sales_data` class in the same file as your `main` function.

2.6.3 Writing Our Own Header Files

Although as we'll see in § 19.7 (p. 852), we can define a class inside a function, such classes have limited functionality. As a result, classes ordinarily are not defined inside functions. When we define a class outside of a function, there may be only one definition of that class in any given source file. In addition, if we use a class in several different files, the class' definition must be the same in each file.

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of the class. For example, the `string` library type is defined in the `string` header. Similarly, as we've already seen, we will define our `Sales_data` class in a header file named `Sales_data.h`.

Headers (usually) contain entities (such as class definitions and `const` and `constexpr` variables (§ 2.4, p. 60)) that can be defined only once in any given file. However, headers often need to use facilities from other headers. For example, because our `Sales_data` class has a `string` member, `Sales_data.h` must `#include` the `string` header. As we've seen, programs that use `Sales_data` also need to include the `string` header in order to use the `bookNo` member. As a result, programs that use `Sales_data` will include the `string` header twice: once directly and once as a side effect of including `Sales_data.h`. Because a header might be included more than once, we need to write our headers in a way that is safe even if the header is included multiple times.

Whenever a header is updated, the source files that use that header must be recompiled to get the new or changed declarations.

A Brief Introduction to the Preprocessor

The most common technique for making it safe to include a header multiple times relies on the **preprocessor**. The preprocessor—which C++ inherits from C—is a

program that runs before the compiler and changes the source text of our programs. Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

C++ programs also use the preprocessor to define **header guards**. Header guards rely on preprocessor variables (§ 2.3.2, p. 53). Preprocessor variables have one of two possible states: defined or not defined. The **`#define`** directive takes a name and defines that name as a preprocessor variable. There are two other directives that test whether a given preprocessor variable has or has not been defined: **`#ifdef`** is true if the variable has been defined, and **`#ifndef`** is true if the variable has *not* been defined. If the test is true, then everything following the `#ifdef` or `#ifndef` is processed up to the matching **`#endif`**.

We can use these facilities to guard against multiple inclusion as follows:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

The first time `Sales_data.h` is included, the `#ifndef` test will succeed. The preprocessor will process the lines following `#ifndef` up to the `#endif`. As a result, the preprocessor variable `SALES_DATA_H` will be defined and the contents of `Sales_data.h` will be copied into our program. If we include `Sales_data.h` later on in the same file, the `#ifndef` directive will be false. The lines between it and the `#endif` directive will be ignored.

Preprocessor variable names do not respect C++ scoping rules.

Preprocessor variables, including names of header guards, must be unique throughout the program. Typically we ensure uniqueness by basing the guard's name on the name of a class in the header. To avoid name clashes with other entities in our programs, preprocessor variables usually are written in all uppercase.

Headers should have guards, even if they aren't (yet) included by another header. Header guards are trivial to write, and by habitually defining them you don't need to decide whether they are needed.

Exercises Section 2.6.3

Exercise 2.42: Write your own version of the `Sales_data.h` header and use it to rewrite the exercise from § 2.6.2 (p. 76).

Chapter Summary

Types are fundamental to all programming in C++.

Each type defines the storage requirements and the operations that may be performed on objects of that type. The language provides a set of fundamental built-in types such as `int` and `char`, which are closely tied to their representation on the machine's hardware. Types can be non`const` or `const`; a `const` object must be initialized and, once initialized, its value may not be changed. In addition, we can define compound types, such as pointers or references. A compound type is one that is defined in terms of another type.

The language lets us define our own types by defining classes. The library uses the class facility to provide a set of higher-level abstractions such as the IO and `string` types.

Defined Terms

address Number by which a byte in memory can be found.

alias declaration Defines a synonym for another type: `using` *name* = *type* declares *name* as a synonym for the type *type*.

arithmetic types Built-in types representing boolean values, characters, integers, and floating-point numbers.

array Data structure that holds a collection of unnamed objects that are accessed by an index. Section 3.5 covers arrays in detail.

auto Type specifier that deduces the type of a variable from its initializer.

base type type specifier, possibly qualified by `const`, that precedes the declarators in a declaration. The base type provides the common type on which the declarators in a declaration can build.

bind Associating a name with a given entity so that uses of the name are uses of the underlying entity. For example, a reference is a name that is bound to an object.

byte Smallest addressable unit of memory. On most machines a byte is 8 bits.

class member Part of a class.

compound type A type that is defined in terms of another type.

const Type qualifier used to define objects that may not be changed. `const` objects must be initialized, because there is no way to give them a value after they are defined.

const pointer Pointer that is `const`.

const reference Colloquial synonym for reference to `const`.

constant expression Expression that can be evaluated at compile time.

constexpr Variable that represents a constant expression. § 6.5.2 (p. 239) covers `constexpr` functions.

conversion Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types.

data member Data elements that constitute an object. Every object of a given class has its own copies of the class' data members. Data members may be initialized when declared inside the class.

declaration Asserts the existence of a variable, function, or type defined elsewhere. Names may not be used until they are defined or declared.

declarator The part of a declaration that includes the name being defined and an optional type modifier.

decltype Type specifier that deduces the type of a variable or an expression.

default initialization How objects are initialized when no explicit initializer is given. How class type objects are initialized is controlled by the class. Objects of built-in type defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values.

definition Allocates storage for a variable of a specified type and optionally initializes the variable. Names may not be used until they are defined or declared.

escape sequence Alternative mechanism for representing characters, particularly for those without printable representations. An escape sequence is a backslash followed by a character, three or fewer octal digits, or an `x` followed by a hexadecimal number.

global scope The scope that is outside all other scopes.

header guard Preprocessor variable used to prevent a header from being included more than once in a single file.

identifier Sequence of characters that make up a name. Identifiers are case-sensitive.

in-class initializer Initializer provided as part of the declaration of a class data member. In-class initializers must follow an `=` symbol or be enclosed inside curly braces.

in scope Name that is visible from the current scope.

initialized A variable given an initial value when it is defined. Variables usually should be initialized.

inner scope Scope that is nested inside another scope.

integral types See arithmetic type.

list initialization Form of initialization that uses curly braces to enclose one or more initializers.

literal A value such as a number, a character, or a string of characters. The value cannot be changed. Literal characters are enclosed in single quotes, literal strings in double quotes.

local scope Colloquial synonym for block scope.

low-level const A `const` that is not top-level. Such `const`s are integral to the type and are never ignored.

member Part of a class.

nonprintable character A character with no visible representation, such as a control character, a backspace, newline, and so on.

null pointer Pointer whose value is 0. A null pointer is valid but does not point to any object.

nullptr Literal constant that denotes the null pointer.

object A region of memory that has a type. A variable is an object that has a name.

outer scope Scope that encloses another scope.

pointer An object that can hold the address of an object, the address one past the end of an object, or zero.

pointer to const Pointer that can hold the address of a `const` object. A pointer to `const` may not be used to change the value of the object to which it points.

preprocessor Program that runs as part of compilation of a C++ program.

preprocessor variable Variable managed by the preprocessor. The preprocessor replaces each preprocessor variable by its value before our program is compiled.

reference An alias for another object.

reference to `const` A reference that may not change the value of the object to which it refers. A reference to `const` may be bound to a `const` object, a nonconst object, or the result of an expression.

scope The portion of a program in which names have meaning. C++ has several levels of scope:

> **global**—names defined outside any other scope
>
> **class**—names defined inside a class
>
> **namespace**—names defined inside a namespace
>
> **block**—names defined inside a block

Scopes nest. Once a name is declared, it is accessible until the end of the scope in which it was declared.

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

signed Integer type that holds negative or positive values, including zero.

string Library type representing variable-length sequences of characters.

struct Keyword used to define a class.

temporary Unnamed object created by the compiler while evaluating an expression. A temporary exists until the end of the largest expression that encloses the expression for which it was created.

top-level const The `const` that specifies that an object may not be changed.

type alias A name that is a synonym for another type. Defined through either a `typedef` or an alias declaration.

type checking Term used to describe the process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

type specifier The name of a type.

typedef Defines an alias for another type. When `typedef` appears in the base type of a declaration, the names defined in the declaration are type names.

undefined Usage for which the language does not specify a meaning. Knowingly or unknowingly relying on undefined behavior is a great source of hard-to-track run-time errors, security problems, and portability problems.

uninitialized Variable defined without an initial value. In general, trying to access the value of an uninitialized variable results in undefined behavior.

unsigned Integer type that holds only values greater than or equal to zero.

variable A named object or reference. In C++, variables must be declared before they are used.

void* Pointer type that can point to any nonconst type. Such pointers may not be dereferenced.

void type Special-purpose type that has no operations and no value. It is not possible to define a variable of type `void`.

word The natural unit of integer computation on a given machine. Usually a word is large enough to hold an address. On a 32-bit machine a word is typically 4 bytes.

`&` operator Address-of operator. Yields the address of the object to which it is applied.

`*` operator Dereference operator. Dereferencing a pointer returns the object to which the pointer points. Assigning to the result of a dereference assigns a new value to the underlying object.

#define Preprocessor directive that defines a preprocessor variable.

#endif Preprocessor directive that ends an `#ifdef` or `#ifndef` region.

#ifdef Preprocessor directive that determines whether a given variable is defined.

#ifndef Preprocessor directive that determines whether a given variable is not defined.