Constants

Constants are expressions with a fixed value.

常量是具有固定值的表达式。

Literals

Literals are the most obvious kind of constants. They are used to express particular values within the source code of a program. We have already used some in previous chapters to give specific values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

字面量是最明显的一种常量。它们用于表示程序源代码中的特定值。在前几章中，我们已经使用了一些方法来给变量赋值或表示我们希望程序输出的消息，例如:

```
a=5
```

The `5` in this piece of code was a *literal constant*.

这段代码中的' 5 '是一个*文字常量*。

Literal constants can be classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.

文字常量可以分为:整型、浮点型、字符型、字符串型、布尔型、指针型和用户定义的文字型。

Integer Numerals

```
1776
707
-273
```

These are numerical constants that identify integer values. Notice that they are not enclosed in quotes or any other special character; they are a simple succession of digits representing a whole number in decimal base; for example, `1776` always represents the value *one thousand seven hundred seventy-six*.

这些是标识整数值的数值常量。注意，它们没有被括在引号或任何其他特殊字符中;它们是一个简单的连续数字表示一个十进制的整数;例如，' 1776 '总是表示值 1776。

In addition to decimal numbers (those that most of us use every day), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literal constants. For octal literals, the digits are preceded with a `0` (zero) character. And for hexadecimal, they are preceded by the characters `0x` (zero, x). For example, the following literal constants are all equivalent to each other:

除了十进制数(我们大多数人每天都在使用)之外，c++还允许使用八进制(以8为基数)和十六进制数(以16为基数)作为字面常量。对于八进制字面量，数字前面有一个' 0 '(零)字符。对于十六进制，它们的前面要有字符' 0x ' (0, x)。例如，下面的字面值常量都是等价的:

```
75         // decimal
0113       // octal
0x4b       // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

所有这些都表示相同的数字:75(75)分别表示为以10为基数的数字、八进制数字和十六进制数字。

These literal constants have a type, just like variables. By default, integer literals are of type `int`. However, certain suffixes may be appended to an integer literal to specify a different integer type:

这些文字常量有一个类型，就像变量一样。默认情况下，整型字面值的类型是' int '。但是，某些后缀可以附加到整数字面量以指定不同的整数类型:

Suffix	Type modifier
`u` *or* `U`	`unsigned`
`l` *or* `L`	`long`
`ll` *or* `LL`	`long long`

Unsigned may be combined with any of the other two in any order to form `unsigned long` or `unsigned long long`.

Unsigned可以与其他两种中的任何一种以任意顺序组合成 unsigned long 或 unsigned long long 。

For example:

```
75         // int
75u        // unsigned int
75l        // long
75ul       // unsigned long
75lu       // unsigned long
```

In all the cases above, the suffix can be specified using either upper or lowercase letters.

在上述所有情况下，可以使用大写字母或小写字母指定后缀。

Floating Point Numerals

They express real values, with decimals and/or exponents. They can include either a decimal point, an `e` character (that expresses *"by ten at the Xth height"*, where *X* is an integer value that follows the `e` character), or both a decimal point and an `e` character:

它们用小数和/或指数表示实数。它们可以包括一个小数点，一个' e '字符(表示*"在第Xth高度处乘10 "*，其中*X*是' e '字符后面的一个整数值)，或者一个小数点和一个' e '字符:

```
3.14159    // 3.14159
6.02e23    // 6.02 x 10^23
1.6e-19    // 1.6 x 10^-19
3.0        // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated-, and the last one is the number *three* expressed as a floating-point numeric literal.

这是用c++表示的四个有效的带小数的数字。第一个数字是PI，第二个是阿伏伽德罗的数字，第三个是电子的电荷(一个非常小的数字)——它们都是近似的——最后一个是用浮点数字文字表示的数字3。

The default type for floating-point literals is `double`. Floating-point literals of type `float` or `long double` can be specified by adding one of the following suffixes:

浮点字面值的默认类型是' double '。类型为' float '或' long double '的浮点字面值可以通过添加以下后缀之一来指定:

Suffix	Type
`f` *or* `F`	`float`
`l` *or* `L`	`long double`

For example:

```
3.14159L   // long double
6.02e23f   // float
```

Any of the letters that can be part of a floating-point numerical constant (`e`, `f`, `l`) can be written using either lower or uppercase letters with no difference in meaning.

任何可以作为浮点数字常量一部分的字母(' e ', ' f ', ' l ')都可以用小写或大写字母书写，意义上没有区别。

Character and string literals

Character and string literals are enclosed in quotes:

字符和字符串文字用引号括起来:

```
'z'
'p'
"Hello world"
"How do you do?"
```

The first two expressions represent *single-character literals*, and the following two represent *string literals* composed of several characters. Notice that to represent a single character, we enclose it between single quotes (`'`), and to express a string (which generally consists of more than one character), we enclose the characters between double quotes (`"`).

前两个表达式表示*单字符字面值*，下面两个表示由几个字符组成的*字符串字面值*。请注意，为了表示单个字符，我们将其括在单引号(" ')之间，而为了表示字符串(通常包含多个字符)，我们将字符括在双引号(' " ')之间。

Both single-character and string literals require quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

单字符和字符串字面值都需要引号将其与可能的变量标识符或保留关键字区分开来。注意这两个表达之间的区别:

`x`

`'x'`

Here, `x` alone would refer to an identifier, such as the name of a variable or a compound type, whereas `'x'` (enclosed within single quotation marks) would refer to the character literal `'x'` (the character that represents a lowercase *x*letter).

在这里，x单独指的是一个标识符，比如变量名或复合类型，而‘ x' (用单引号括起来)指的是字符字面量'x'(表示小写*x*字母的字符)。

Character and string literals can also represent special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). These special characters are all of them preceded by a backslash character (`\`).

字符和字符串文字也可以表示在程序源代码中难以或不可能以其他方式表示的特殊字符，如换行符(\n)或制表符(\t)。这些特殊字符前面都有一个反斜杠字符()。

Here you have a list of the single character escape codes:

这里你有一个单字符转义代码的列表:

Escape code	Description
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\b`	backspace
`\f`	form feed (page feed)
`\a`	alert (beep)
`\'`	single quote (`'`)
`\"`	double quote (`"`)
`\?`	question mark (`?`)
`\\`	backslash (`\`)

For example:

`'\n''\t'"Left \t Right""one\ntwo\nthree"`

Internally, computers represent characters as numerical codes: most typically, they use one extension of the ASCIIcharacter encoding system (see ASCII code for more info). Characters can also be represented in literals using its numerical code by writing a backslash character (`\`) followed by the code expressed as an octal (base-8) or hexadecimal (base-16) number. For an octal value, the backslash is followed directly by the digits; while for hexadecimal, an x character is inserted between the backslash and the hexadecimal digits themselves (for example: `\x20` or `\x4A`).

在内部，计算机将字符表示为数字代码:最典型的是，它们使用[ASCII]的一个扩展(http://www.cplusplus.com/ascii)character编码系统(参见ASCII code获取更多信息)。字符也可以使用数字代码以文字形式表示，即输入一个反斜杠字符(' \ ')，后跟表示为八进制(base-8)或十六进制数(base-16)的代码。对于八进制值，反斜杠后面直接跟着数字;对于十六进制，在反斜杠和十六进制数字之间插入一个' x '字符(例如:' \x20 '或' \x4A ')。

Several string literals can be concatenated to form a single string literal simply by separating them by one or more blank spaces, including tabs, newlines, and other valid blank characters. For example:

只需用一个或多个空格(包括制表符、换行符和其他有效的空白字符)将多个字符串字面值连接起来，就可以形成单个字符串字面值。例如:

```
"this forms" "a single"     " string "
"of characters"
```

The above is a string literal equivalent to:

上面的字符串字面量相当于:

```
"this formsa single string of characters"
```

Note how spaces within the quotes are part of the literal, while those outside them are not.

注意，引号内的空格是文字的一部分，而引号外的空格则不是。

Some programmers also use a trick to include long string literals in multiple lines: In C++, a backslash (`\`) at the end of line is considered a *line-continuation* character that merges both that line and the next into a single line. Therefore the following code:

一些程序员还使用一种技巧在多行中包含长字符串:在c++中，行尾的反斜杠(' \ ')被认为是一个*行延续*字符，它将该行和下一行合并为一行。因此，以下代码:

```
x = "string expressed in \
two lines"
```

is equivalent to:

```
x = "string expressed in two lines"
```

All the character literals and string literals described above are made of characters of type `char`. A different character type can be specified by using one of the following prefixes:

上面描述的所有字符字面值和字符串字面值都由' char '类型的字符组成。可以使用下列前缀之一指定不同的字符类型:

Prefix	Character type
`u`	`char16_t`
`U`	`char32_t`
`L`	`wchar_t`

Note that, unlike type suffixes for integer literals, these prefixes are *case sensitive*: lowercase for `char16_t` and uppercase for `char32_t` and `wchar_t`.

For string literals, apart from the above `u`, `U`, and `L`, two additional prefixes exist:

注意，与整型字面值的类型后缀不同，这些前缀是*区分大小写的*:小写代表' char16_t '，大写代表' char32_t '和' wchar_t '。

对于字符串字面量，除了上面的' u '， ' u '和' L '，还有两个额外的前缀:

Prefix	Description
`u8`	The string literal is encoded in the executable using UTF-8
`R`	The string literal is a raw string

In raw strings, backslashes and single and double quotes are all valid characters; the content of the literal is delimited by an initial R"*sequence*(and a final)*sequence*", where *sequence* is any sequence of characters (including an empty sequence). The content of the string is what lies inside the parenthesis, ignoring the delimiting sequence itself. For example:

在原始字符串中，反斜杠、单引号和双引号都是有效字符;字面量的内容由开头的R"序列(和最后的)序列"分隔，其中序列是任何字符序列(包括空序列)。字符串的内容位于括号内，忽略分隔序列本身。例如:

```
R"(string with \backslash)"
R"&%$(string with \backslash)&%$"
```

Both strings above are equivalent to "string with \backslash". The R prefix can be combined with any other prefixes, such as `u`, `L` or `u8`.

上面的两个字符串都等价于"带有\反斜杠的字符串"。前缀R可以与任何其他前缀组合，如u、L或u8。

Other literals

Three keyword literals exist in C++: `true`, `false` and `nullptr`:

c++中有三个关键字字面量:' true ', ' false '和' nullptr ':

- `true` and `false` are the two possible values for variables of type `bool`.
- `nullptr` is the *null pointer* value.

```
bool foo = true;
bool bar = false;
int* p = nullptr;
```

Typed constant expressions

Sometimes, it is just convenient to give a name to a constant value:

有时候，给一个常量赋一个名字是很方便的:

```
const double pi = 3.1415926;
const char tab = '\t';
```

We can then use these names instead of the literals they were defined to:

然后我们可以使用这些名称而不是定义它们的字面量:

```
#include <iostream>
using namespace std;

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

Preprocessor definitions (#define)

Another mechanism to name constant values is the use of preprocessor definitions. They have the following form:

另一种命名常量的机制是使用预处理器定义。它们有以下形式:

```
#define identifier replacement
```

After this directive, any occurrence of `identifier` in the code is interpreted as `replacement`, where replacement is any sequence of characters (until the end of the line). This replacement is performed by the preprocessor, and happens before the program is compiled, thus causing a sort of blind replacement: the validity of the types or syntax involved is not checked in any way.

在这个指令之后，代码中任何' identifier '的出现都被解释为' replacement '，替换是任何字符序列(直到行尾)。这种替换由预处理器执行，并且发生在程序编译之前，因此导致了一种盲替换:所涉及的类型或语法的有效性不会以任何方式进行检查。

For example:

```
#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;                // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

}
```

Note that the `#define` lines are preprocessor directives, and as such are single-line instructions that -unlike C++ statements- do not require semicolons (;) at the end; the directive extends automatically until the end of the line. If a semicolon is included in the line, it is part of the replacement sequence and is also included in all replaced occurrences.

请注意，#define行是预处理器指令，因此是单行指令——不像c++语句——不需要在末尾加上分号(;);指令会自动扩展到行尾。如果分号包含在行中，则它是替换序列的一部分，并且也包含在所有替换的出现中。