Dan Saks

Numeric Literals

The C language provides several different kinds of constants: *integer constants* such as `10` and `0x1C`, *floating constants* such as `1.0` and `6.022e+23`, and *character constants* such as `'a'` and `'\x10'`. C also provides *string literals* such as `"ouch!"` and `"\n"`. C++ provides the same kinds of constants and literals, except the C++ standard just calls all of them *literals* instead. I do, too.

Every literal in C and C++ has a type. For example, `10` is obviously an integer. But is it an `int` (which is `signed` by default) an `unsigned int`, or a `short` or `long` variation thereof? Do you even care?

If you're a C programmer, you might not care. Part of me hopes you'd care, but I'm not absolutely sure you need to. Many C programmers apparently get by quite well without knowing the exact types of constants.

If you're a C++ programmer, you really should care. C++ supports function name overloading, and the exact type of a literal just might determine which function is the best match for a particular call. For example, given:

```
int f(int);
unsigned int f(unsigned int);
long int f(long int);
```

which of these functions does `f(10)` call? The answer is:

```
int f(int);
```

because the precise type of `10` is `(signed) int`.

Forms of integer literals

An integer literal takes different forms:

- A decimal integer literal (base 10) is a non-zero digit followed by zero or more decimal digits
- A hexadecimal integer literal (base sixteen) is `0x` or `0X` followed by a sequence of one or more hexadecimal digits
- An octal integer literal (base eight) is the digit `0` followed by a sequence of zero or more octal digits

For example, the decimal integer literal `63` can also be expressed as the hexadecimal integer literal `0x3F` or as the octal integer literal `077`.

Any integer literal may have a suffix that influences its type:

- `U` or `u` specifies that the literal has an `unsigned` type
- `L` or `l` specifies that the literal has a `long` type
- `LL` or `ll` specifies that the literal has a `long long` type. (This is available in the most recent edition of Standard C, C99, but not in earlier versions of C or in C++)

For example, `63u` has type `unsigned int`. `0x3FL` has type `(signed) long int`. An integer literal suffix may combine `U` with either `L` or `LL` (in either upper or lower case and in either order), so that both `63ul` and `0x3FLU` have type `unsigned long int`.

An integer literal's suffix influences the literal's type, but does not determine it. The type also depends on the literal's value.

Every literal has a type. It may not be obvious, and it may vary across platforms, but the standard does specify it precisely.

Types of integer literals

In general, an integer literal's type is the smallest integer type no smaller than `int` that can hold the literal's value and still satisfy the constraints imposed by the form and suffix. For example, the type of a decimal integer literal with no suffix is the smallest (the first) of the following types that can represent its value: `int`, `long int`, and in C99, `long long int`. Because the maximum values of the types vary across platforms, the exact types for many integer literals also vary across platforms.

Both the C and C++ standards allow considerable slack in the range of values for integer types. The maximum value for an `int` must be at least 2^{15}–1, and the maximum value for a `long int` must be at least 2^{31}–1. In practice, this means that an `int` must

In C++, the only form of a floating literal is a decimal floating literal, in which all of the digits (in both the significant part and the exponent part) are decimal digits. C99 also provides hexadecimal floating literals, in which the digits of the significant part are hexadecimal digits.

TABLE 1 The type of integer literals in C99 and C++

Suffix	Decimal Literal	Hexadecimal Literal
None	int	int
		unsigned int
	long int	long int
		unsigned long int
	long long int	long long int
		unsigned long long int
U or u	unsigned int	unsigned int
	unsigned long int	unsigned long int
	unsigned long long int	unsigned long long int
L or l	long int	long int
		unsigned long int
	long long int	long long int
		unsigned long long int
LL or ll	long long int	long long int
		unsigned long long int
Both U and L (in either case and either order)	unsigned long int	unsigned long int
	unsigned long long int	unsigned long long int
Both U and LL (in either case and either order)	unsigned long long int	unsigned long long int

Lighter areas apply only to C99

occupy at least 16 bits and a `long int` must occupy at least 32 bits.

On any platform, `int` and `long int` may occupy more than their minimum storage requirement, as long as a `long int` occupies at least as much storage as an `int`. An unsigned integer type must occupy the same amount of storage as its corresponding signed integer type. For instance, a 16-bit platform might use 16 bits for `int` and `unsigned int` and 32 bits for `long int` and `unsigned long int`. A 32-bit platform might use 32 bits for all four types, `int`, `unsigned int`, `long int`, and `unsigned long int`.

Table 1 describes the rules for determining the type of an integer literal in C99 and C++. A literal whose value can't be represented in any of the listed types produces a compile error.

The rules in Table 1 lead to some potential portability problems. A decimal integer literal whose value is not greater than 32,767 (= 2^{15}–1) has type `int` on every standard C and C++ environment. No surprise there. However, any decimal integer literal whose value is greater than 2^{15}-1 but not greater than 2^{31}-1 has type `long int` on a 16-bit platform, but just plain `int` on a 32-bit platform.

For example, given the overloaded functions:

```
int f(int);
unsigned int f(unsigned int);
long int f(long int);
unsigned long int
    f(unsigned long int);
```

calling `f(32768)` calls `f(long int)` when compiled using C++ for a 16-bit platform, but it calls `f(int)` when compiled for a 32-bit platform. On the other hand, if you write the call as `f(0x8000)` (`0x8000` is 32,768 as a hexadecimal literal), it calls `f(unsigned int)` on every platform.

Floating literals

Floating literals offer a few little surprises, too. The type of a floating literal such as `1.0` or `6.022e+23` is `double`, not `float`. Appending the letter `F` or `f` to a floating literal makes its type `float`. Appending the letter `L` or `l` to a floating literal makes its type `long double`. For example, `1.0F` has type `float`, and `6.022e+23L` has type `long double`. Combining `F` and `L` (in either case and in either order) in a floating literal is a syntax error.

In C++, the only form of a floating literal is a decimal floating literal, in which all of the digits (in both the significant part and the exponent part) are decimal digits. C99 also provides hexadecimal floating literals, in which the digits of the significant part are hexadecimal digits.

In a decimal floating literal, the exponent part is optional. If present, it begins with `E` or `e` followed by decimal digits representing a power of 10. In a hexadecimal floating literal, the exponent part is mandatory. Since `E` and `e` are valid hexadecimal digits, a hexadecimal floating literal uses `P` or `p` to mark the beginning of the exponent part, following by decimal digits representing a power of two. For example, `0x1.FFFFFEp127f` is a floating constant whose value is `0x1.FFFFFE` multiplied by 2^{127} (decimal). The `f` at the end of the literal specifies that the literal's type is `float`.

Neither the form nor value of a floating literal affects its type. The suffix letter (or absence thereof) completely determines the type.

Character and string literals

Integer and floating literals are essentially the same in both C and C++. The differences lie in C's added support for `long long` integer types and hexadecimal floating literals. In contrast, character and string literals behave slightly differently between C and C++. Those differences are my topic for next time. **esp**

Dan Saks is the president of Saks & Associates, a C/C++ training and consulting company. He is also a consulting editor for the C/C++ Users Journal. *He served for many years as secretary of the C++ standards committee. With Thomas Plum, he wrote* C++ Programming Guidelines *(Plum Hall). You can write to him at dsaks@wittenberg.edu.*