C Programming Language Overview

Leonardo Resende Lopes

"[C has] the power of assembly language and the
convenience of ... assembly language." - Dennis Ritchie

1 Introduction

C is a general-purpose programming language. Many of the important ideas of C stem from the language BCPL, developed by Martin richards. C provides a variety of data types. In addition, there is a hierarchy of derived data types created with **pointers, arrays, structures and unions**.

Expressions are formed from **operators and operands**; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well-structured programs: **statement grouping**, decision making **(if-else)**, selecting one of a set of possible cases **(switch)**, looping with the termination test at the **top (while, for)** or at the **bottom (do)**, and early loop exit **(break)**.

2 Types, Operators and Expressions

2.1 Variable Names

Names are made up of letters and digits; the first character must be a letter. Uppercase and lowercase letters are distinct.

Traditional C practice is to use lower case for variable names, and all uppercase for symbolic constants.

2.2 Data Types and Sizes

There are only a few basic data types in C:

- **char**

  A single byte, capable of holding one character in the local character set.

- **int**

  An integer, typically reflecting the natural size of integers on the host machine.

- **float**

  Single-precision floating point.

- **double**

  Double-precision floating point.

In addition, there are a number of qualifiers that can be applied to these basic types. **short** and **long** apply to integers.

The intent is that short and long should provide different lengths of integers where practical. Int will normally be the natural size for a particular machine. **short** is often 16 bits, **long** 32 bits, and **int** either 16 or 32 bits.

Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long.

The qualifier **signed** or **unsigned** may be applied to char or any integer.

Unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2^n, where n is the number of bits in the type. Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive.

The type **long double** specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; **float**, **double** and **long double** could represent one, two or three distinct sizes.1

2.3 Constants

An **integer constant** like 1234 is an int. A **long constant** is written with a terminal "l" or "L" in 123456789L; An integer too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal "u" or "U", and the suffix "ul" or "UL" indicates unsigned long.

Floating-point constants contain a decimal point (123.4), or an exponent (1e-2) or both; their type is double, unless suffixed. The suffix "f" or "F" indicate a float constant; "l" or "L" indicate a **long double**.

The **value of an integer can be specified in octal or hexadecimal** instead of decimal. A leading zero on an integer constant means octal; a leading '0x' or '0X' means hexadecimal.

A **character constant** is an integer, written as one character within single quotes, such as 'x'. The value of a character constant is the numeric value of the character in the machine's character set.

Certain characters can be represented in character and **string constants** by escape sequences like '\n' (newline). In addition, an arbitrary byte-sized bit pattern can be specified by '\ooo', where **ooo** is one to three octal digits, or by '\xhh', where **hh** is one or more hexadecimal digits. Id est, we might write:

```
1 #define VTAB '\013' /* ASCII vertical tab   */
2 #define BELL '\007' /* ASCII bell character */
```

or, in hexadecimal:

```
1 #define VTAB '\xb' /* ASCII vertical tab    */
2 #define BELL '\x7' /* ASCII bell character  */
```

The complete set of escape sequences is:

\a	Alert(bell) character
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal Tab
\v	Vertical Tab
\ \	Backslash
\?	Question mark
\'	Single quote
\"	Double quote
\ooo	Octal number
\xhh	Hexadecimal number

The character constant **'\0'** represents the character with value zero, the null character. The '\0' is foten written instead of 0 to emphasize the character nature of some expression, but the **numeric value is just 0**.

A **constant expression** is an expression that involves only constants.

A **string constant**, or **string literal**, is is a sequence of zero or more characters surrounded by double quotes. String constants can be concatenated at compile time:

```
"hello," "world"
```

is equivalent to:

```
"hello, world"
```

This is useful for splitting long strings across several source lines.

Technically, a string constant is an array of characters. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function **strlen(s)** returns the length of its character string argument **s**, excluding the terminal '\0'.

Be careful to distinguish between a character constant and a string that contains a single character.

An enumeration is a list of constant integer values, as in:

```
enum boolean { NO, YES};
```

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. Names in different enumerations must be distinct. Values need not to be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to '**#define**' with the advantage that the values can be generated for you.

Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, **enumeration variables offer the chance of checking and so are often better than '#defines'**.

2.4 Declarations

All variables must be declared before use, although certain declarations can be made implicitly by context. A declaration specifies a type, and contains a list of one or more variables of that type, as in:

```
int lower, upper, step;
char c, line[1000];
```

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expressions serves as an initializer, as in:

```
char esc = '\\';
int i = 0;
int limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptually before the program starts executing, and the **initializer must be a constant expression**. An explicitly initialized automatic variable is initialized each time the function or block it is in entered; the initializer may be any expression. **External and static variables are initialized to zero by default**. Automatic variables for which there is no explicit initializer have undefined (i.e., garbage) values.

The qualifier **const** can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a const.

2.5 Arithmetic Operators

The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression:

```
x % y
```

produces the remainder when x is divided by y, and thus is zero when y divides x exactly.

The % operator cannot be applied to **float** or **double**. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as in the action taken on **overflow and underflow**.

The binary + and - operators have the same precedence, which is lower than the precedence of *, /, and %, which is in turn lower than unary + and -.

Arithmetic operators associate left to right.

2.6 Relational and Logical Operators

The relational operators are >, >=, < and <=.

They all have the same precedence. Just below them in precedence are the equality operators: == and ! =.

Relational operators have lower precedence than arithmetic operators, so an expression like $i < lim - 1$ is taken as $i < (lim - 1)$, as would be expected.

More interesting are the logical operators && and ||. Expressions connect by && or || are evaluated left to right, and evaluation stops as soon as the truth or falsehood of the result is known.

The precedence of && is higher than that of ||, and both are lower than relational and equality operators.

By definition, the numeric value of a relational or logical expression is 1 if the realtion is true, and 0 if the relation is false.

The unary negation operator ! converts a **non-zero operand** into 0, and a **zero operand** into 1.

2.7 Type conversions

2.8 Increment and Decrement Operators

2.9 Bitwise Operators

2.10 Assignment Operators and Expressions

2.11 Conditional Expressions

2.12 Precedence and Order of Evaluation