Types, Operators and Expressions

Devendra Tewari

2024-10-09

Introduction

- The basic C data objects are
  - Variables
  - Constants
- Operators act on data objects
- Expressions are composed of data objects and operators

Variable Names

- Names are composed of letters and digits
  - Underscore is treated as a letter
- Must start with a letter
- Are case sensitive
- Have a size limit of 31 characters
- As a convention lower case is used for variable names

Keywords

- Symbols reserved by C
- Cannot be used as variable names

auto	default	for	return	switch	while
break	double	goto	short	typedef	
case	else	if	signed	union	
char	enum	int	sizeof	unsigned	
const	extern	long	static	void	
continue	float	register	struct	volatile	

Basic Data Types

- Integer
  - char, short int, int, long int
  - signed and unsigned types
- Floating point
  - float
  - double, long double
- Size is implementation dependent
  - `<limits.h>` and `<float.h>` contain constants for sizes

Integer types

- int is normally the natural size for a machine
- unsigned types store negative values in two's complement form
- char meant for holding single byte character code
  - insufficient for Unicode and other codes

	bits	unsigned	signed
long	32	0 to 2^32 - 1	-2^31 to 2^31 - 1
int	32	0 to 2^32 - 1	-2^31 to 2^31 - 1
short	16	0 to 2^16 - 1	-2^15 to 2^15 - 1
char	8	0 to 2^8 - 1	-2^7 to 2^7 - 1

Floating point types

- Maximum and minimum values implementation dependent
- Decimal precision is limited

	Range	Precision
float	1E-37 to 1E+37	6
double	1E-308 to 1E+308	15
long double	1E-308 to 1E+308	15

Constants

- Following types
  - Numeric
  - Character
  - String
  - Enumeration

Numeric constants

- Integer constants
  - A sequence of numbers (4567) is an `int`
  - Signed long integers specified by suffixing `l` or `L` (`4567L`) and unsigned long integers by suffixing `ul` or `UL` (`4567UL`)
  - Octal values indicated with a leading `0`
  - Hexadecimal values indicated with leading `0x`
- Floating-point constants
  - A sequence of numbers with a decimal point (1.23) or an exponent (123E-2) or both (12.3E-1) is a double
  - Long doubles are indicated by suffixing `l` or `L`

Character constants

- A character between single quotes (`'0'`) is a `char`
- A character constant represents the integer value of the character (`'0'` = 48 in ASCII character set)
- A character constant is more portable

String constants

- A sequence of characters delimited by double quotes (`"hello world\n"`)
- Strings constants separated by white-spaces are concatenated at compile time
  - `"hello " "world\n"` becomes `"hello world\n"`
- Internally a string constant is terminated by a `'\0'` (null) character
- Function `strlen(s)` in `<string.h>` returns the size of a string
- A string is actually an array of char (`char []`)

Escape sequences

- Some characters are hard to represent in character constants and string constants
- Escape sequences are used to represent such characters

`\a`	alert (beep)	`\\`	backslash
`\b`	backspace	`\?`	question mark
`\f`	formfeed	`\'`	single quote
`\n`	newline	`\"`	double quote
`\r`	carriage return	`\nnn`	octal number
`\t`	horizontal tab	`\xhh`	hexadecimal number
`\v`	vertical tab	`\0`	null character

Constant expression

- Expressions involving only constants
- May be evaluated at compile time
- Can be used in the place of a constant

```
#define MAX 100
int i = MAX;
```

Enumeration constants

- A list of constant integers
- Values can be specified or generated

```
enum colors { RED = 'r', BLUE = 'b', GREEN = 'g'};

enum dow { SUN = 1, MON, TUE, WED, THU, FRI, SAT};
```

- Variables of enum types can be declared

```
enum colors c = RED;
```

- DDD shows values of enum variables as symbols

Variable Declarations

- All variables must be declared before use
- A declaration only specifies the nature of a variable (i.e. type)
- A declaration contains a type followed by a list of one or more comma separated names

```
char c, name [50];
```

- A variable may be initialized in its declaration

```
int i = MAX + 1, j = i;
```

- By prefixing `const` to a declaration, a variable can be declared as unchangeable

```
const double pi = 3.14;
```

Arithmetic Operators

- Binary operators

  `+ - * / %`

- Unary operators

  `++ -- + -`

- Postfix / prefix unary operator usage

```
int i = 0, j;
j = i++; /* j = 0, i = 1 */
j = ++i; /* j = 2, i = 2 */
```

Relational operators

- Equal and not equal

  `==` `!=`

- Less than and less than or equal to

  `<` `<=`

- Greater than and greater than or equal to

  `>` `>=`

- No boolean type, a value of 0 represents FALSE, any other value is TRUE

Logic operators

- Logical negation

  !

- Logical AND

  &&

- Logical OR

  ||

Bitwise operators

- Bitwise complement

  `~`

- Bitwise left and right shift

  `<< >>`

- Bitwise AND, OR, and exclusive OR (XOR)

  `& | ^`

Assignment operators

- Normal assignment

  `=`

- Arithmetic

  `+= -= *= /= %=`

- Bitwise assignment operators

  `&= ^= |= <<= >>=`

Expressions

- Most expressions are assignments or functions calls
- If an expression is missing the statement is called a null statement
  - Can be used to supply an empty body for an iteration or loop
- All side effects from the expressions are completed before the next statement

Side effects

- Side effects are unpredictable assignment to variables resulting from undefined order of evaluation of an expression
  - In function calls

    ```
    printf("%d, %d", i + 1, i = j + 2 );
    printf("%d, %d, %d", i++, i++, i++);
    ```

  - Nested assignments

    ```
    c = getchar() != EOF
    ```

  - Increment and decrement operators

    ```
    a[i] = i++;
    ```

- Avoid side effects, don't depend on the results from your compiler

Operator Precedence

`() [] -> .`	left to right
`! ~ ++ -- + - * &` `(type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %=` `&= ^= |= <<= >>=`	right to left
`,`	left to right

Conditional expressions

- `lvalue = expr1 ? expr2 : expr3`

  `lvalue` is the value of the expression
  `expr1` is evaluated first
  `expr2` is evaluated if `expr1` is not 0 (i.e. true)
  `expr3` is evaluated if `expr1` is 0 (i.e. false)

- This is equivalent to

```
if (expr1)
  lvalue = expr2;
else
  lvalue = expr3;
```

Automatic type conversion

- Wider conversions are automatic, e.g. `char` to `int`
- Beware mixing signed and unsigned values, `-1L > 1UL`

  `1UL`
  `00000000000000000000000000000001`

  `1L` as two's complement of `1L` (calculated as `~1+1` or `2^32 - 1`)
  `11111111111111111111111111111111`

Automatic type conversion rules

For an operator that takes two operands

- If one is `long double`, convert other to `long double`
- Else, if one is `double`, convert other to `double`
- Else, if one is `float`, convert other to `float`
- Else, if one is `unsigned long int`, convert other to `unsigned long int`
- Else, if one is `long int` and other is `unsigned int`
  - If a `long int` can represent all values of an `unsigned int`, convert both to `long int`
  - Else convert both to `unsigned long int`
- Else, if one is `long int`, convert other to `long int`
- Else, if one is `unsigned int`, convert other to `unsigned int`
- Else, convert both to `int`

Type casting

- Forced type conversions (coercion)

  (*type-name*) *expression*
- Required for *narrow* conversions
- Can result in loss of data
  - Wide integers to narrow integers
  - Float or double to integers