Lectures: C Programming Language

Programming Systems and Tools

State University of New York at Binghamton

January 25, 2024

Binghamton University | Thomas J. Watson College of Engineering and Applied Science

Types, Operators & Expressions

Outline

1 Types, Operators & Expressions
- Variable Names
- Types
- Operators
- Expressions

Tokenization

C source code is comprised of *tokens*

Tokens

A *string* which has an assigned meaning, comprised of a token name, and a token value.

identifier	names the programmer chooses: `foo`, `color`, `PI`
keywords	names in the programming language: `if`, `while`, `return`, `int`, `char`
separators	punctuation of the language: `()`, `;`
operators	symbols that operate on values: `+`,`=`
literal	fixed numeric and logical values: `true`, `3.14159`, `'token'`
comment	programmer-readable annotations ignored by the compiler: `/* block comment */`, `// inline comment`

Tokenization

- C is whitespace insensitive: `x = 2 + y` is same as mintinlinec$x = 2 + y$
- C is case sensitive: `foo` is not the same as `FOO`
- These rules are the source of a lot of bugs!
  - `foo` is not the same as `FOO`
  - → Choose a whitespace style an identifier convention . . . and be consistent!

Variables

Variable names in C must follow the following rules:

- Composed of letters, digits, and underscore
- Begin with a letter or underscore (not digits)
- Case Sensitive

Remember, variables:

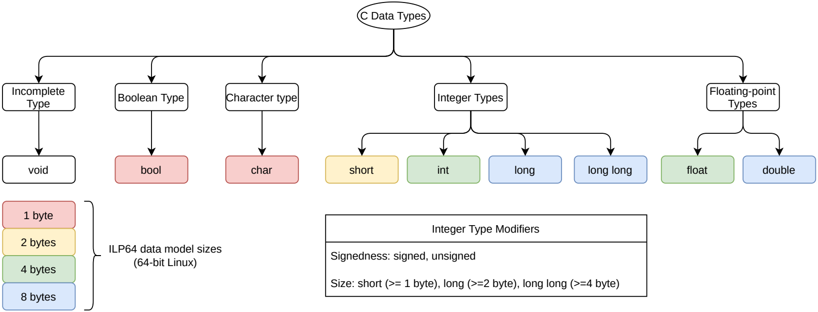
- Must be *declared* with a data type →reserves the necessary memory for the variable
- Must be *initialized* before use →assigns a value

```c
// Declaration
int x;
// Initialization
x = 42;
// Combined declaration and initialization
int z = 0;
// Multiple declarations and initialization
int a,b = 32; // OK
int c,d = 48,64; // not OK
```

Types

- Every variable in C must have a type, which tells the compiler how much data is to be interpreted (and how much memory it needs)
- Technically C has unlimited types because of user-defined types
- C has a fundamental type hierarchy with five classes and nine types
- All other types are defined from a composition of these types

Types


C Data Types
Incomplete Type
Boolean Type
Character type
Integer Types
Floating-point Types
void
bool
char
short
int
long
long long
float
double
1 byte
2 bytes
4 bytes
8 bytes
ILP64 data model sizes (64-bit Linux)
Integer Type Modifiers
Signedness: signed, unsigned
Size: short (>= 1 byte), long (>=2 byte), long long (>=4 byte)

Constants and Literals

C has constant values of certain types. Examples:

- **Binary** `0b01010101` with leading `0b`
- **Decimal** `1234` (int), `123456789L` (long int), `12345U` (unsigned int)
- **Octal** `01234567` with leading `0`
- **Hexadecimal** `0x123abc`, `0X123DEF` with leading `0x` or `0X`
- **Character** `'x'`, `\170`, `\x78`
- **Float** 123.4, 123e-4 with decimal point and/or scientific notation exponent
- **Boolean** `true`, `false`
- **Null Pointer** `nullptr`

Character Literals

Certain characters don’t have a printable glyph or a special meaning in a string, but are still in the ASCII character set. We can access them two ways, defining constants or escape sequences:

```
#define FF '\x0c'   /* ASCII form feed */
#define VTAB '\013' /* ASCII vertical tab */
/* ... etc  */
```

Escape	**Meaning**	**Escape**	**Meaning**
\a	alert (bell)	\\	backslash
\b	backspace	\?	question mark
\f	form feed	\’	single quote
\n	newline	\”	double quote
\r	carriage return	\ooo	octal literal
\t	horizontal tab	\xhh	hexadecimal literal
\v	vertical tab	\0	null terminator

Enumerations

enum

A list of constant integer values associated with a name, substituted with the value at compile time.

- Once declared, the values may not change (enums aren't variables).
- The first value in an enum has the value 0 unless an explicit value is specified.
- Unspecified values continue the progression of natural integers from the last specified value.
- Names in enumerations must be distinct, however values need not be distinct within the same enumeration.

Enumerations

Enumeration Benefits

Enumerations provide a convenient alternative to `#define`, with added benefits:

- Values can be generated automatically
- The compiler can perform type and name checking
- The debugger can print enumeration values symbolically

Enumeration Examples

```c
enum months { JAN = 1, FEB/*=2*/, MAR=/*=3 etc.*/,
              APR, MAY, JUN,
              JUL, AUG, SEP,
              OCT, NOV, DEC };
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
enum color { RED=0xFF0000, GREEN=0x00FF00, BLUE=0x0000FF };
```

- C operations are unary (1 operand), binary (2 operand), or ternary (conditional)

Common operators

assignment	increment decrement	arithmetic	logical	comparison	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
						`a(...)`
						`a, b`
						`(type) a`
						`? :`
						`sizeof`
						`_Alignof` (since C11)

`https://en.cppreference.com/w/c/language/expressions`

Arithmetic Operators

- *Promotion, Negation*: Unary $+, -$ before a signed variable make it positive or negative (not specified means assumed positive).
- If there are mixed types, smaller type is *promoted* to larger type.
  - e.g. integral and float addition: `(int) 1 + (float) 4.3 == 5.3`
- Two operators for modular division, which must take integral types for arguments:
  - Integer division returns quotient: `5 / 3 == 1`
  - Modulo division returns remainder: `5 % 3 == 2`
  - Always: `(a/b)*b + a%b == a`
- Remember: you must assign the result or it is discarded (arithmetic operators don't modify operands by themselves!)

Logical Operators

- In C, zero values are `false`, all others are `true`
- Three operators, which return `true`/`false`:
  - Logical NOT (negation): `!expr`
  - Logical AND (conjunction): `lhs && rhs`
  - Logical OR (Disjunction): `lhs || rhs`
- Logical AND will *short circuit* if the left-hand value is `false`.
- Logical OR will *short circuit* if the left hand value is `true`.
- You must assign the result or it is discarded!
- Definition of the `bool` type and `false`, `true` values are implementation-defined in **stdbool.h**.

Type Conversion

- Once a variable is declared, its type cannot be changed
- But what if you want to assign its value to a variable of a different size? We must *cast* it. Casting doesn't change the type of the old variable!
- We can always cast from a smaller type to a larger type. It will be *promoted* automatically. But if we cast from larger to smaller, information will be lost.
- Conversion can be *implicit* (performed by the compiler to create an equivalent value) or *explicit* (the raw representation of the source is copied to the destination and interpreted according to the destination type).

```c
char c = 'A';
int chr = (int) c;

// prints "c: A (0x41), chr: 65"
printf("c: %c (%#X), chr: %i\n", c, c, chr);
```

Increment and Decrement

- Two operators: increment (++) and decrement(--)
- Two forms: prefix and postfix
  - Prefix yields the value then changes it
  - Postfix changes the value then yields it
- You must assign the result or it is discarded!

```
// What are the values printed on each line?
// What is the value of ans after each line?
int ans = 42;
printf("%d\n", ans);
printf("postincrement: %d\n", ans++);
printf("postdecrement: %d\n", ans--);
printf("preincrement: %d\n", ++ans);
printf("predecrement: %d\n", --ans);
```

Binary: What and Why?

- Everything in the computer is binary: text, media, code
- Why do computers use binary and not base-10?
  - Computers are built out of switches
  - These switches detect a voltage relative to a threshold
  - Base-10 systems would require 10 different discrete voltage levels to encode!
- *Note that no matter what representation of a number is displayed, it's still binary to the computer!*

How Does Binary Work?

- $2_{10} = 10_2$
- Binary powers of two work like decimal places:
  - Shifting left one place is like multiplying by 2.
  - Shifting right one place is like dividing by 2.
  - We can do place expansion to convert between the two bases.

Converting from Binary to Decimal

short ans = 42;															
MSB 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	LSB 0
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

$$0b0000000000101010$$
$$= (0 \cdot 2^{15}) + (0 \cdot 2^{14}) + \ldots (1 \cdot 2^{5}) + (0 \cdot 2^{4}) + (1 \cdot 2^{3}) + (0 \cdot 2^{2}) + (1 \cdot 2^{1}) + (0$$
$$= (0) + (0) + \ldots (1 \cdot 32) + (0) + (1 \cdot 8) + (0) + (1 \cdot 2) + (0)$$
$$= 32 + 8 + 2$$
$$= 42$$

Converting from Decimal to Binary

short ans = 42;															
MSB 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	LSB 0
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0

$$
\begin{aligned}
42/2 = 21, 42\%2 = 0 &\rightarrow 0b0 \\
21/2 = 10, 21\%2 = 1 &\rightarrow 0b10 \\
10/2 = 5, 10\%2 = 0 &\rightarrow 0b010 \\
5/2 = 2, 5\%2 = 1 &\rightarrow 0b1010 \\
2/2 = 1, 2\%2 = 0 &\rightarrow 0b01010 \\
1/2 = 0, 1\%2 = 1 &\rightarrow 0b101010
\end{aligned}
$$

Note that we are constructing the binary representation from lower order to higher order.

Negative Numbers

We need to be able to represent negative numbers ...

- Binary bits don't have a sign (i.e they can't represent -1, 0, +1)

→ We reserve the Most Significant Bit (MSB) for the sign in signed numbers

→ If MSB is set, then the number is negative.

- But reserving a bit reduces the maximum value we can express by a factor of two

→ So C types can be modified to be `signed` (use the MSB for sign) or `unsigned` (use the MSB for the value)

- If not specified, the default is to treat it as signed.

→ But what happens if we later cast from unsigned to signed?

One's Complement

- *One's Complement* is the most intuitive way to represent a negative number.
- Reserve the last bit as a signed bit, and then flip remaining bits for negative numbers.

$\rightarrow$ To get the positive value, we can just invert the bits:

$$
\begin{aligned}
0b11111110& \\
0b00000001& \rightarrow \text{flip the bits (inversion)} \\
-0d1& \rightarrow \text{use negative sign (negation)}
\end{aligned}
$$

One's Complement

- *One's Complement* is the most intuitive way to represent a negative number.
- Reserve the last bit as a signed bit, and then flip remaining bits for negative numbers.

$\rightarrow$ To get the positive value, we can just invert the bits:

$$
\begin{aligned}
0b11111110 & \\
0b00000001 & \rightarrow \text{flip the bits (inversion)} \\
-0d1 & \rightarrow \text{use negative sign (negation)}
\end{aligned}
$$

But this creates the problem of having two representations for zero: 0b. . . 1111 = -0

Two's Complement

- *Two's Complement* resolves the problem of having two representations of zero.
- Same procedure as *One's Complement*, except after inversion, we will also add 1

$$
\begin{aligned}
&0b11111110 \\
&0b00000001 \rightarrow \text{flip the bits (inversion)} \\
&0b00000010 \rightarrow \text{add 1} \\
&\qquad -0d2 \rightarrow \text{use negative sign (negation)}
\end{aligned}
$$

Bitwise Operators

Operator	Operator Name	Example	Result
`~`	Bitwise NOT	`~a`	bitwise inversion of a
`&`	Bitwise AND	`a & b`	bitwise conjunction of a and b
`|`	Bitwise OR	`a | b`	bitwise disjunction of a and b
`^`	Bitwise XOR	`a ^ b`	bitwise exclusive disjunction of a and b
`<<`	Bitwise SHL	`a << b`	a is bitwise shifted left by b places
`>>`	Bitwise SHR	`a >> b`	a is bitwise shifted right by b places

Bitwise Inversion - NOT

NOT	0	1
	1	0

- Equivalent to 1's Complement

Example:

	0	1	0	1	1	0	1	0
NOT	1	0	1	0	0	1	0	1

Bitwise Conjunction- AND

AND	0	1
0	0	0
1	0	1

Example:

	0	1	0	1	1	0	1	0
	0	0	0	0	1	1	1	1
AND								
	0	0	0	0	1	0	1	0

Bitwise Disjunction - OR

OR	0	1
0	0	1
1	1	1

Example:

	0	1	0	1	1	0	1	0
	0	0	0	0	1	1	1	1
OR	0	1	0	1	1	1	1	1

Bitwise Exclusive Disjunction - XOR

XOR	0	1
0	0	1
1	1	0

Example:

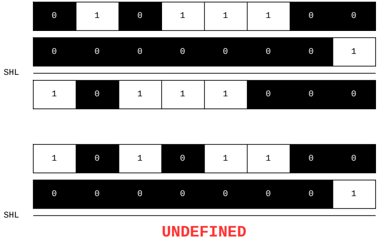
	0	1	0	1	1	0	1	0
	0	0	0	0	1	1	1	1
XOR								
	0	1	0	1	0	1	0	1

Bitwise Shift Left - SHL

SHL	$A_{n-1} = 0$	$A_{n-1} = 1$
$A_0(LSB)$	0	0
A_n	0	1

- SHL is undefined if rhs is negative or is greater or equal the number of bits in the promoted lhs.
- For unsigned lhs, the value of `LHS << RHS` is the value of $lhs \cdot 2^{rhs}$
- For signed lhs, the value of `LHS << RHS` is the value of $lhs \cdot 2^{rhs}$

Example:



Note: signed values shown.

Bitwise Shift Right - SHR

SHR	$A_{n+1} = 0$	$A_{n+1} = 1$
A_{MSB}	A_{MSB}	A_{MSB}
A_n	0	1

- For unsigned lhs and for signed lhs with non-negative values, the value of `LHS >> RHS` is the integer part of $lhs/2^{rhs}$
- For signed LHS with negative values, the value of `LHS >> RHS` is implementation-defined (in x86, the result remains negative).

Example:

	0	1	0	1	1	1	0	0
SHR	0	0	0	0	0	0	0	1
	0	0	1	0	1	1	1	0

	1	0	1	0	1	1	0	0
SHR	0	0	0	0	0	0	0	1
	1	1	0	1	0	1	1	0

Note: signed values shown.

Comparing Bitwise Arithmetic and Logical Operators

What is the output of this code?

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  // Note: no such thing as %b binary format specifier!
  // Note: how many binary digits in each number?
  printf("0d3 = 0b011, 0d7 = 0b111\n");

  // Compare the outputs
  printf("AND: arithmetic(3 & 7): %d, logical(3 && 7): %d\n",
         3 & 7, 3 && 7);
  printf("OR: arithmetic(3 | 7): %d, logical(3 || 7): %d\n",
         3 | 7, 3 || 7);
  printf("NOT: arithmetic(~3): %d, logical(!3): %d\n",
         ~3, !3);
  printf("NOT: arithmetic(~7): %u, logical(!7): %u\n",
         ~7, !7);

  return 0;
}
```

Lab: Set, Clear and Toggle

Implement the following three functions:

```
unsigned long long setBit(unsigned long long n, int k);
unsigned long long clearBit(unsigned long long n, int k);
unsigned long long toggleBit(unsigned long long n, int k);
```

- Setting a bit means that if k-th bit is 0, then set it to 1 and if it is 1 then leave it unchanged.
- Clearing a bit means that if k-th bit is 1, then clear it to 0 and if it is 0 then leave it unchanged.
- Toggling a bit means that if k-th bit is 1, then change it to 0 and if it is 0 then change it to 1.
- Hints:
  - Think about the NOT, AND, OR, and XOR truth tables.

Lab: Set, Clear and Toggle

- Can I choose a known bit value such that the truth table results in setting, clearing or toggling a second unknown bit value?
- I can move my bit into the correct location for masking using SHL and SHR.
- How do arguments n and k relate to each of the hints above?

Lab: Set, Clear and Toggle

Examples:

```
// 0d5 = 0b101
printf("setBit(5,1): %d\n", setBit(5ULL, 1)); // 7
printf("clearBit(5,1): %d\n", clearBit(5ULL, 1)); // 5
printf("toggleBit(5,1): %d\n", toggleBit(5ULL, 1)); // 7

// 0d7 = 0b111
printf("setBit(7,2): %d\n", setBit(7ULL, 2)); // 7
printf("clearBit(7,2): %d\n", clearBit(7ULL, 2)); // 3
printf("toggleBit(7,2): %d\n", toggleBit(7ULL, 2)); // 3

// 0d10 = 0b1010
printf("setBit(10,2): %d\n", setBit(10ULL, 2)); // 14
printf("clearBit(10,2): %d\n", clearBit(10ULL, 2)); // 10
printf("toggleBit(10,2): %d\n", toggleBit(10ULL, 2)); // 14
```

Memory Access Operators

More on these soon!

Operator	Operator name	Example	Description
[]	array subscript	a[b]	access the **b**th element of array **a**
*	pointer dereference	*a	dereference the pointer **a** to access the object or function it refers to
&	address of	&a	create a pointer that refers to the object or function **a**
.	member access	a.b	access member **b** of struct or union **a**
->	member access through pointer	a->b	access member **b** of struct or union pointed to by **a**

Other Operators

Operator	Operator name	Example	Description
`(...)`	function call	`f(...)`	call the function **f**(), with zero or more arguments
`,`	comma operator	`a, b`	evaluate expression **a**, disregard its return value and complete any side-effects, then evaluate expression **b**, returning the type and the result of this evaluation
`(type)`	type cast	`(type)a`	cast the type of **a** to **type**
`? :`	conditional operator	`a ? b : c`	if **a** is logically true (does not evaluate to zero) then evaluate expression **b**, otherwise evaluate expression **c**
`sizeof`	sizeof operator	`sizeof a`	the size in bytes of **a**
`_Alignof` (since C11)	_Alignof operator	`_Alignof(type)`	the alignment required of **type**

Expressions

Expression

A sequence of operators and their operands that specify a computation.

Expressions can:

- produce a result: `2 + 2` produces the integer 4
- generate side-effects: `printf("%c\n, 52)` prints the character '4' to standard output
- designate objects or functions: in `func(a,b)`, `func` is the function designator
- be comprised of primary expressions and subexpressions: `1 + 2 * 3` has a primary expression `1` and a subexpression `2 * 3` as operands to the operator `+`

Assignment Operators

- Basic assignment is `lhs = rhs`
- Each arithmetic operation has an associated compound form `lhs op= rhs`, where the operation is both performed and assigned to the left-hand value: `lhs = lhs op rhs`!
  - Example: `lhs = lhs + rhs;` can also be written `lhs += rhs;`
  - Read as "lhs becomes equal to the addition of lhs and rhs."

Conditional Expression

To conditionally execute a statement, or choose between more than one statement, use *selection statements*: `expr1 ? expr2 : expr3`

1. First evaluate expr1.
2. If expr1 is non-zero (true), evaluate expr2
3. If expr1 is zero (false), evaluate expr3
4. The value of the right hand side of `?` is the result of the conditional expression

Because there are three clauses, the conditional expression is sometimes called a *ternary operator*.

Comparison Operators

- C has the usual assortment of comparison operators.
- Notably missing an explicit identity operator (e.g. `===` or `is`). Instead, take the address of two objects and compare.
- The type of a comparison operator is int and has a value of 1,0 for `true, false`.

Operator	Operator name	Example	Description
`==`	equal to	`a == b`	**a** is equal to **b**
`!=`	not equal to	`a != b`	**a** is not equal to **b**
`<`	less than	`a < b`	**a** is less than **b**
`>`	greater than	`a > b`	**a** is greater than **b**
`<=`	less than or equal to	`a <= b`	**a** is less than or equal to **b**
`>=`	greater than or equal to	`a >= b`	**a** is greater than or equal to **b**

Comparison Operators

Beware Comparing Floats

```c
#include <assert.h>
int main(void) {
        // some float comparisions make sense
        assert (2 + 2 == 4.0); // ints promote to float

        // some floats can't be compared at all
        double d = 0.0/0.0; // NaN
        assert( !(d < d) );
        assert( !(d > d) );
        assert( !(d <= d) );
        assert( !(d >= d) );
        assert( !(d == d) );

        // some floats compare counterintuitively
        // because of their precisions
        float f = 0.1; // f = 0.100000001490116119384...
        double g = 0.1; // g = 0.1000000000000000555...
        assert(f > g); // different values
}
```

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right