C Primer (2) – Expressions and Basic Data Types

Ion Mandoiu
Laurent Michel
Revised by M. Khan, J. Shi and W. Wei

Expressions in C

- Similar to expressions in Java/C++/Python
- Expressions are inductively defined:
  - **Constants**, **variables,** and function calls (covered later)
  - Combining expressions using parentheses and **operators**
- All C expressions have a **type**
  - Constants have a type
  - Variables have a type
  - Function return values have a type
  - Every sub-expression of a larger expression has a type
- Adding a semicolon to an expression makes it a statement

A Few Basic Data Types

- int
  - An integer
- char
  - A single byte that can store a character in ASCII
  - An 8-bit integer
- float
  - Floating point numbers

More on basic data types later…

Constants (of basic types)

```
// Constants cannot be changed
// char
'a', 'b', '\n'

// integer (note that compiler stores them in binary)
200, -34
0x7fffFFFF // hex
07112      // octal

// floating point numbers
3.1415, -0.34, 1.3E20
```

Variables

- **All variables must be declared and initialized before use**
- Variable declarations specify the type and name
    - Compiler allocates memory based on type
    - Valid names consist of letters (case sensitive!), digits, and '_', but cannot start with digits
    - Multiple variables of the same type can be declared together
    - Variables can be initialized when declared (“variable definition”) or using separate assignments

Examples:

```
char c;
   int  i, j, k = 1;
   float f;
```

Operators

- Conventional arithmetic, bitwise, and logical operators

```
+  -  *  /  %
&  |  ~  ^  <<  >>
&& || !
```

- Pre/post increment/decrement (as in Java, C++, etc.)

```
i++  ++i  j--  --j
c = i++;    //  c will be (i - 1)
c = ++i;    //  c will be the same as i
```

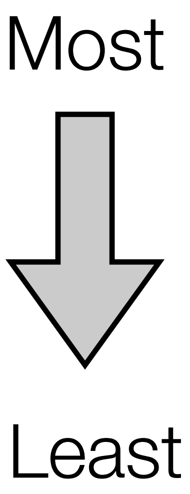
- Simple and compound assignment operators

More to come!

Precedence and associativity

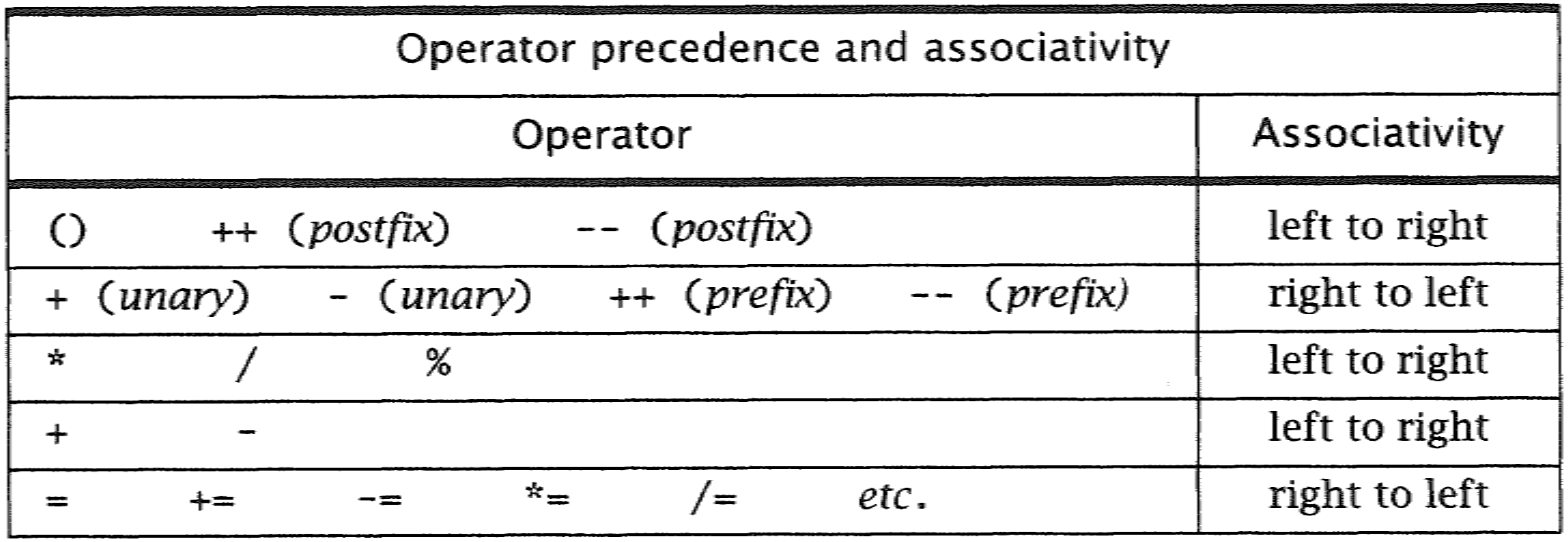
- Precedence determines which operation is done first
  - If operators have the same precedence, use associativity
  - Use parentheses

```
i + j * 10 - k / 20
(i + (j * 10)) - (k / 20)
```



Most

Least



Operator precedence and associativity

Operator	Associativity
() ++ *(postfix)* -- *(postfix)*	left to right
+ *(unary)* - *(unary)* ++ *(prefix)* -- *(prefix)*	right to left
* / %	left to right
+ -	left to right
= += -= *= /= *etc.*	right to left

Assignment operators

- Assignment operator

```
LHS = Expression
```

  - LHS (Left Hand Side) is something that can be ***written to*** (e.g, a variable)
  - LHS and Expression have “compatible” types
  - The value of Expression is assigned to LHS and becomes the value of the assignment operation

- Compound assignment operators (+=, *=, …)

```
var op= expr    ⟺    var = var op expr
```

Examples:

```
a = x + y;      b = c = d = 0;
i += 10;        // i = i + 10
j *= 5;         // j = j * 5
```

Assignments ARE NOT Statements

- Assignments are ***expressions*** and “=” is an operator
  - You can chain them!
  - You can use them inside larger expressions

```
int a,b,c;
a = b = c = 10;
```

```
int a,b,c;
a = (b = (c = 10));
c = 10;
b = 10;
a = 10;
```

```
int a,b,c;
a = (b = 2) + (c = 3);
```

```
int a,b,c;
b = 2;
c = 3;
a = b + c;
```

Integer Data Types

`char`		
`short int`	←→	`short`
`int`		
`long int`	←→	`long`
`long long int`	←→	`long long`

And unsigned versions like “unsigned char”, “unsigned short”, etc.

- How many bytes does each take?
  - Depends on CPU architecture and compiler

Integer Data Types

- Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 .. 127	unsigned char 0..255
16	short -32768..32767	unsigned short 0..65535
32	int $- 2^{31} .. 2^{31} - 1$	unsigned int $0..2^{32}-1$
64	long $- 2^{63} .. 2^{63} - 1$	unsigned long $0..2^{64} -1$
64	long long $- 2^{63} .. 2^{63} - 1$	unsigned long long $0..2^{64} -1$

Integer Data Types

- Consider i386 (32-bit architecture)

size (in bits)	signed	unsigned
8	`char` -128 .. 127	`unsigned char` 0..255
16	`short` -32768..32767	`unsigned short` 0..65535
32	`int` $- 2^{31} .. 2^{31} - 1$	`unsigned int` $0..2^{32}-1$
32	`long` $- 2^{31} .. 2^{31} - 1$	`unsigned long` $0..2^{32}-1$
64	`long long` $- 2^{63} .. 2^{63} - 1$	`unsigned long long` $0..2^{64} -1$

How much space?

- How to determine the amount of space for some type?
- Operator sizeof gives the number of bytes needed for a type or a variable
  - You will need this later to dynamically allocate space!

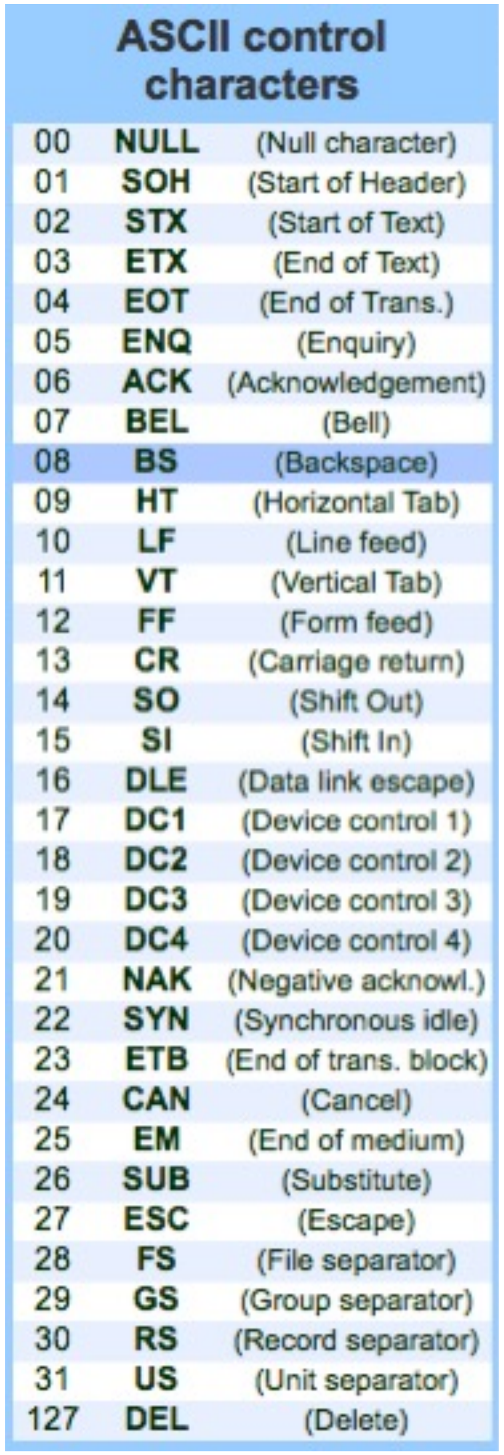
```
sizeof(T)
```

Examples:

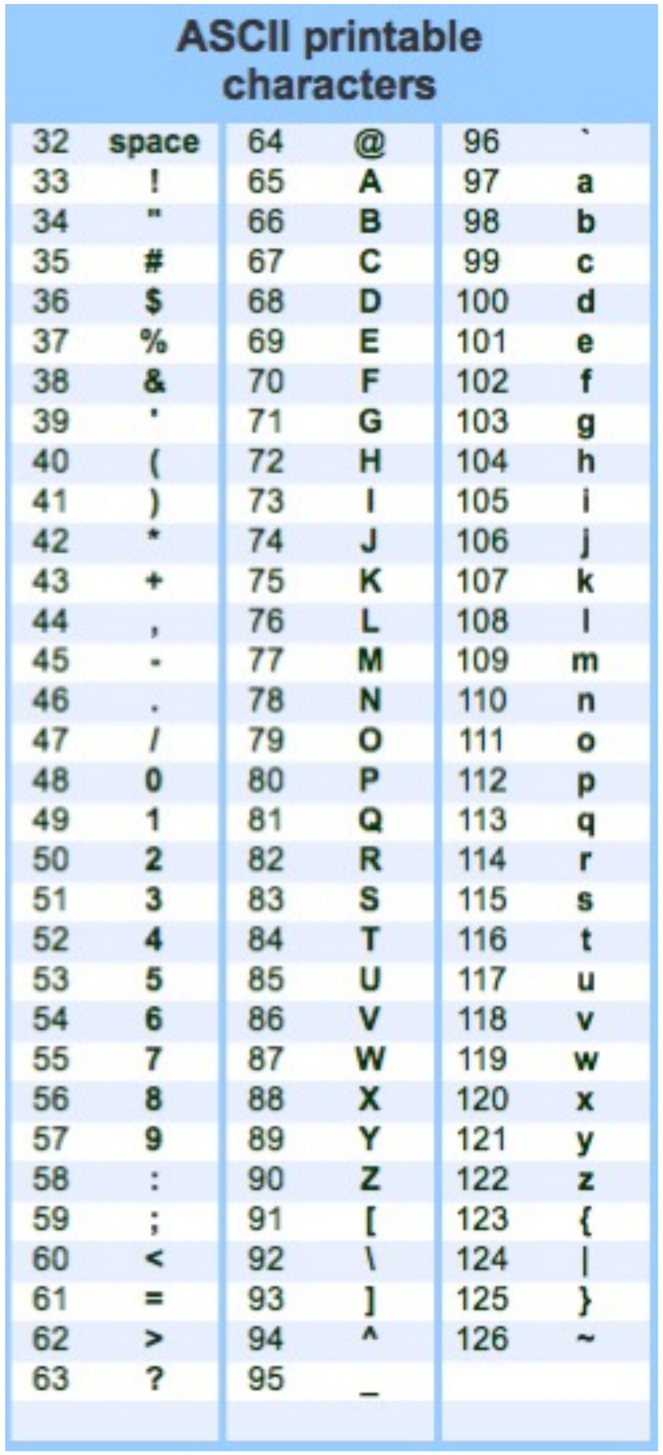
```
int i;
sizeof(int);  sizeof(i);
// 4 on the machines we use in this course
```

Character (char) Data Type

- char has 8 bits (a byte)
- ASCII code
  - Characters are mapped to an integer in 0..127
  - An ASCII character can be stored in char
- Classes in ASCII
  - 0..31: “Control” character (aka, non-printable)
  - 48..57: Digits
  - 65..90: Upper case letters
  - 97..122: Lower case letters



ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)



ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

So…

- The character ‘H’ is none other than …. 72

```
char h1 = 'H', h2 = 72; // h1 and h2 have the same value
```

- Observe how…
  - ‘0’ through ‘9’ are consecutive!
  - ‘A’ through ‘Z’ are consecutive!
  - ‘a’ through ‘z’ are consecutive!

Want to see ASCII table in your terminal?

```
man ascii
```

```
char ch = '8';
int  x = ch - '0';    // What is the value of x?
```

Non-printable characters?

- These are sometimes useful
  - Showing the constant (literal)

`'\n'`	newline
`'\r'`	carriage-return
`'\f'`	form-feed
`'\t'`	tabulation
`'\b'`	backspace
`'\x7'`	audible bell (x indicates hexadecimal)
`'\07'`	audible bell (0 indicates octal)

Basic Data Types: Floating Point

- A few ***floating point*** types
  - Consider x86_64 again

size (in bits)	size (bytes)	Name & Range
32	4	**float** 1.17 * 10^{-38} to 3.40 * 10^{+38}
64	8	**double** 2.22 * 10^{-308} to 1.79 * 10^{+308}
80/128	16	**long double** 3.65 * 10^{-4951} to 1.18 * 10^{+4932}

Automatic Type Conversion

- When an operator has operands of different types, the operands are **automatically** converted to a common type by the compiler
  - In general, a lower rank operand is converted into the type of the higher rank one, where

    char < short < int < long < long long < float < double < long double

  - E.g., 1 gets converted to double before performing the addition in the expression 1 + 2.5
- Automatic conversion can also occur across assignments
  - The value of the expression on right hand side may be **widened** to the type of the LHS, e.g., double d = 1;
  - Or **narrowed** (possibly with information loss), e.g., int i = 2.5;
- Read the book for more details!

Type Casting: Explicit Type Conversion

- Useful to convert an operand to another type before doing arithmetic

```
(<Type>)<expression>
```

Example: integer or double?

```
int x = 10;
int y = 3;

double z = x / y;
```

```
int x = 10;
int y = 3;

double z = (double)x / y;

// the following doesn't work
// z = (double)(x / y)
```

What About Booleans?

- K&R and C89/C90 do not have a Boolean data type
  - 0 “means” FALSE and anything else “means” TRUE
  - Common to use int or char to store Boolean values and define convenience macros

```
#define BOOL    char
#define TRUE    1
#define FALSE   0
```

- C99 introduced _Bool
  - A variable of _Bool type is either 0 or 1

Be Mindful…

- Sometimes the results may not be as expected!
  - What is the size (in bits) of each operand?
  - Are your operands signed or unsigned ?

Examples

```
unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
```

```
_Bool b1;
char  b2, b3;
int   i = 256; // 0x100

b1 = i;
b2 = i;
b3 = i != 0;
```

Be Mindful…

- Sometimes the results may not be as expected!
  - What is the size (in bits) of each operand?
  - Are your operands signed or unsigned ?

Examples

```
unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
```

z holds the binary representation of -4, but reading it as an unsigned int yields a very different value!

```
_Bool b1;
char  b2, b3;
int   i = 256; // 0x100

b1 = i;
b2 = i;
b3 = i != 0;
```

b1 is 1 because i is not 0
b2 is 0 because lowest 8 bits in i are 0
b3 is 1 because i is not 0

Do you want b2 or b3?

- Study remaining slides yourself

Examples: char constants

```
// single quotation marks, only one character


'h'
'\n'
'\007'      // octal.
'\xAA'      // hex. 170 = 0xAA
'\''        // single quotation mark
'\\'        // back slash
'"'         // no need to escape double quotation mark here
```

Examples: integer and floating-point constants

```
200
-300
0x7fffFFFFu  // hex. unsigned int. case insensitive
0123456      // octal. starting with 0!
0x12345678L  // hex. long int
123UL        // unsigned long
123LL        // long long
12345678901234567890ull // unsigned long long
3.14f          // float literals
3.14L          // long double literals
```

Integral promotion (ABC 3.11)

- Integer types smaller than int, for example, char or short, are promoted to int or unsigned int when an operation is performed on them

```
// If there is no integral promotion,
// c1 * c2 would not have 600 as result
char r, c1, c2, c3;
c1 = 100;
c2 = 6;
c3 = 8;
r = c1 * c2 / c3;
```

Bitwise Operators (ABC 7.1)

op.	Description	Example
&	bitwise AND	Set bits to 0. Mask out bits
|	bitwise OR	Set bits to 1
^	bitwise XOR	Flip some bits (using masks)
~	1’s complement	Flip all bits
<<	Shift left	Move bits to left.
>>	Shift right	Move bits to right (pay attention to the sign)

All these operators can be suffixed with =
For instance a &= b; is the same as a = a & b;

Example: bitwise operations

```
11010011
    &
10001100
-------------
10000000
```

```
11010011
    |
10001100
-------------
11011111
```

```
11010011
    ^
10001100
-------------
01011111
```

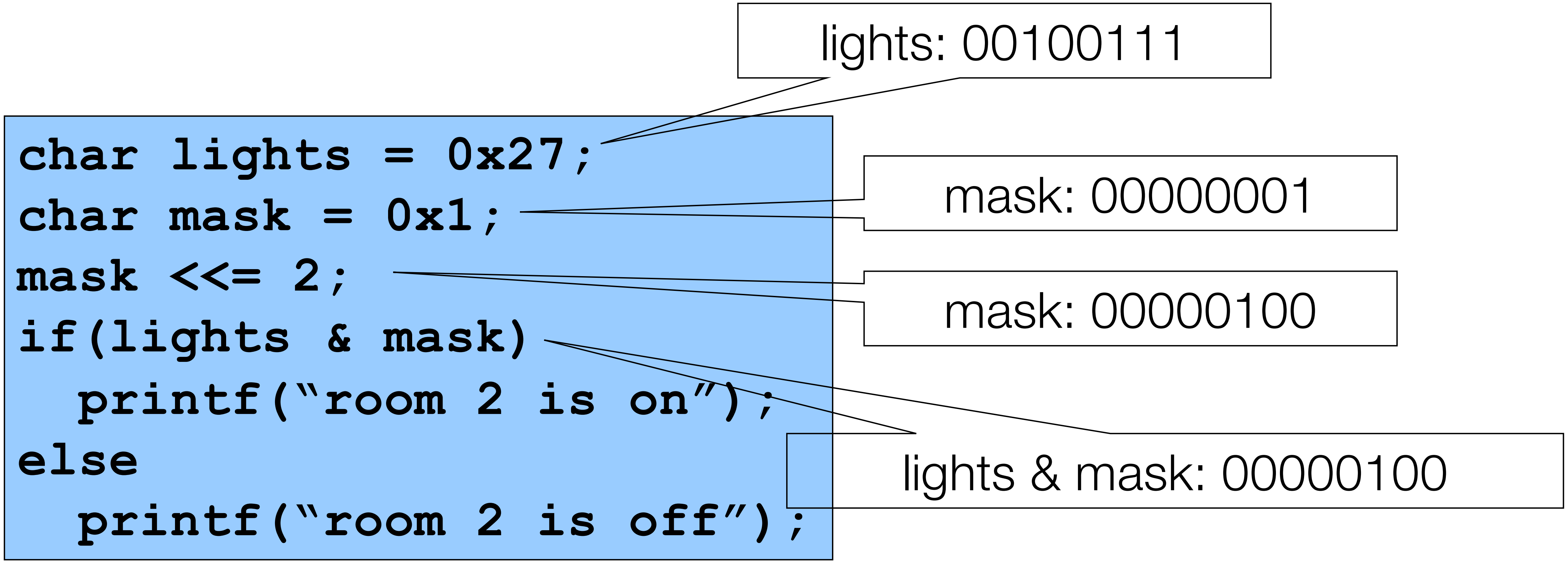
```
~11010011
-------------
 00101100
```

```
11010011<<3
-------------
   10011000
```

```
01010011>>3
-------------
   00001010
```

Example: getting bits

- Suppose bit 2 (the third bit from the right) of lights indicates if the light is on (if bit 2 is 1) or off (if bit 2 is 0)
- Hexadecimal representations are (shorter and) easier to read



```
char lights = 0x27;
char mask = 0x1;
mask <<= 2;
if(lights & mask)
  printf(“room 2 is on”);
else
  printf(“room 2 is off”);
```

lights: 00100111

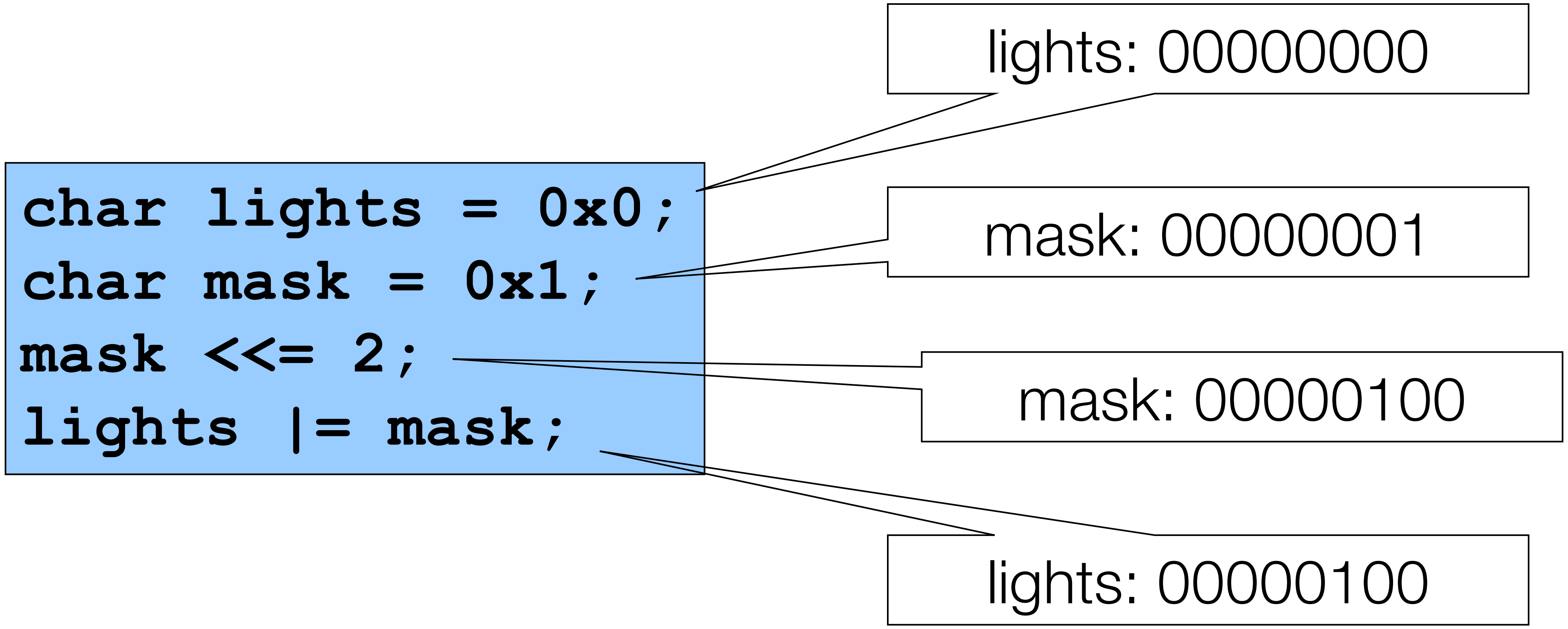
mask: 00000001

mask: 00000100

lights & mask: 00000100

Example: setting bits

- Set bit 2 in lights to 1



```
char lights = 0x0;
char mask = 0x1;
mask <<= 2;
lights |= mask;
```

lights: 00000000

mask: 00000001

mask: 00000100

lights: 00000100

Integers of specific sizes (C99)

```
#include <stdint.h>
int8_t          // signed 8 bits integers
int16_t
int32_t
int64_t
uint8_t         // unsigned 8 bits integers
uint16_t
uint32_t
uint64_t
// Many projects have their own *standard* types
```