KUAS
KYOTO UNIVERSITY OF ADVANCED SCIENCE
京都発世界人財

京都先端科学大学

Information Processing 2 05

declarations
operators
precedence

Ian Piumarta

Faculty of Engineering, KUAS

last week: types and expressions; declarations and initialisation

variable name rules
`char int float double short long`
integer constants, characters
escape sequences,
constant expressions
string constants
enumeration constants, `enum`
advantages of `enum` over `#define`
type conversions, coercion, and cast

this week: declarations, operators, precedence

declarations and initialisation
operators: arithmetic, relational
increment and decrement operators
bitwise logical operators
assignment operators
conditional expressions
operator precedence and associativity
sequence points during expression evaluation
ambiguous (undefined) expressions

declarations and initialisation

all variables must be declared before use

type-name list-of-variables ;

```
int  lower, upper, step, numDigits[10];
char c, line[1000];
```

variables can be declared and initialised at the same time

```
int  i = 0, limit = LINEMAX - 1;
```

the qualifier `const` indicates that a variable cannot be modified

```
const double e = 2.71828182845905;
const int    linemax = 1000;
```

initialisation of arrays

arrays are initialised by writing a list of elements in curly braces

```
int numDigits[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

the number of elements must not exceed the size of the array

when the array contains characters it can be initialised with a string

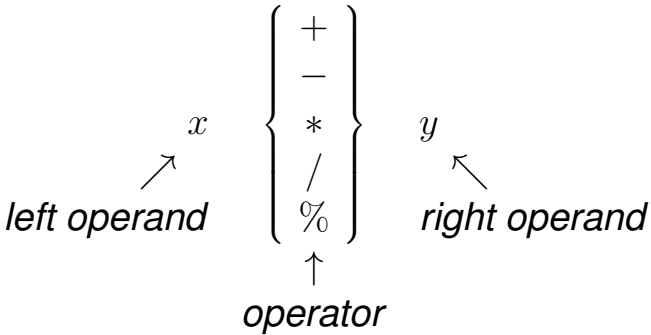
```
char m[6] = "hello"; // includes nul terminator (a string)
char n[5] = "hello"; // no nul terminator (not a string)
```

the size of an array can be inferred from the number of initialiser elements

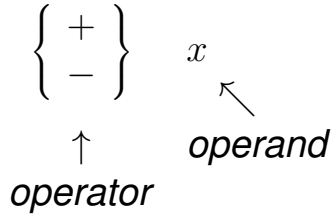
```
int  pow[] = { 1, 10, 1000 };  // int  pow[3]
char str[] = "bye";            // char str[4]  <- NOTE
```

arithmetic operators

binary operators



unary operators



arithmetic operators

work on any numbers: `int long float double`

- except modulus (%) which only works on integers

+	addition	also unary
−	subtraction	also unary (negation)
*	multiplication	
/	division (quotient)	
%	modulus (remainder)	integer operands only

common pattern: `if (num % den == 0) // num is a multiple of den`

relational operators

logical (true/false) values are represented as integers

- the value `0` is false
- any other value is true (but conventionally `1` represents ‘true’)

relational operators compare two things to produce a logical result

- their result is `0` if the relation is false
- their result is `1` if the relation is true

`<`	less
`<=`	less or equal
`>=`	greater or equal
`>`	greater
`==`	equal
`!=`	not equal

operations on logical (Boolean) values

a *truth table* describes how a logical operator behaves

if we write false as `0` and true as `1`, then:

not

x	!x
0	1
1	0

and

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

or

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

exclusive or

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

logical operators

logical operators combine logical values with ‘and’ and ‘or’

`&&`	and
`||`	or

`x && y` is true only if both `x` **and** `y` are true

`x || y` is true only if either `x` **or** `y` is true

logical expressions are evaluated left-to-right (`&&` has higher precedence than `||`)

```
#define LINEMAX 10

  char line[LINEMAX], i = 0;

  while (i < LINEMAX - 1 && (c = getchar()) != EOF && c != '\n') {
    line[i] = c;
    ++i;
  }
  line[i] = '\0';
```

logical operators

evaluation of logical operators stops as soon as the result (true/false) is known

since ‘false and *anything’* is **always** false:

$$x \;\&\&\; y \;\&\&\; z$$
$$\uparrow \qquad \uparrow \qquad \uparrow$$

if all values are non-`0` (true) then the overall result is `1` (true), otherwise:

evaluation stops at the first value that is `0` (false) and the result is `0` (false)

since ‘true or *anything’* is **always** true:

$$x \;||\; y \;||\; z$$
$$\uparrow \qquad \uparrow \qquad \uparrow$$

if all values are `0` (false) then the overall result is `0` (false), otherwise:

evaluation stops at the first value that is non-`0` (true) and the result is `1` (true)

logical operators

there is one unary logical operator: ! not

! converts: false (0) → true (1)
true (non-zero) → false (0)

the result of ! is always either 0 or 1

useful pattern: !!x is 0 if x is 0, 1 if x is non-zero

note: `if (result == 0)` and `if (!result)` mean the same thing

- choose the one that makes the program easier to understand
- the second form is more usual when `result` is a logical value

increment and decrement operators

we have already used:

```
++i   increment i by 1
--i   decrement i by 1
```

you can also write:

```
i++   increment i by 1
i--   decrement i by 1
```

when used only for their *side effect* (modifying `i`) they are equivalent

when used in an expression for their *value*, they are different

name	*notation*	*meaning*
pre-increment	`++i`	the value of `i` after incrementing it
pre-decrement	`--i`	the value of `i` after decrementing it
post-increment	`i++`	the value of `i` before incrementing it
post-decrement	`i--`	the value of `i` before incrementing it

compare: `while (--i > 0)` with `while (i-- > 0)`

common pattern: `while (i < linemax) line[i++] = '\0'`

beware: order of increment and decrement operations

be very careful when using increment/decrement operators

C does not define *when* the updated value is written back into the variable

given

```
int array[10], i = 0;
```

the following operations are **undefined** (the compiler should warn you about this)

```
printf("%d\n", i + i++);  // printed value is undefined

array[i] = array[++i];
//     ↑
//     index value is undefined
```

bitwise operators

six operators are provided that manipulate individual bits within integers

	binary		**unary**
&	and	~	not
|	or		
^	exclusive or		
<<	left shift		
>>	right shift		

these operators only work on integers

bitwise and

performs ‘and’ between each bit of an integer and the corresponding bit in another

```
          0x33 = 0b 0 0 1 1 0 0 1 1
                   & & & & & & & &
          0x55 = 0b 0 1 0 1 0 1 0 1
                   ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
0x33 & 0x55 = 0b 0 0 0 1 0 0 0 1
```

bitwise and

in other words, calculates which bits are 1 in **both** operands

```
int A =          0b00001111;
int B =          0b00110011;
int C = A & B; // 00000011
```

and

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

& can be used to *clear* (force to 0) some subset of bits in an integer

```
x = y & 0b00001111; // clear all but the LS four bits of y
```

useful pattern: `x & 1` is equal to the least significant bit of `x`

bitwise or

performs ‘or’ between each bit of an integer and the corresponding bit in another

```
          0x33 =  0b 0  0  1  1  0  0  1  1
                     |  |  |  |  |  |  |  |
          0x55 =  0b 0  1  0  1  0  1  0  1
                     ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
0x33 | 0x55 =  0b 0  1  1  1  0  1  1  1
```

bitwise or

in other words, calculates which bits are 1 in **either** operand

```
int A =          0b00001111;
int B =          0b00110011;
int C = A | B; // 00111111
```

or

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

| can be used to *set* (force to 1) some subset of bits in an integer

```
x = y | 0b00001111; // set the LS four bits of y
```

bitwise exclusive or

performs ‘exclusive or’ between corresponding bits of two integers

```
          0x33 =  0b 0  0  1  1  0  0  1  1
                     ^  ^  ^  ^  ^  ^  ^  ^
          0x55 =  0b 0  1  0  1  0  1  0  1
                     ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
0x33 ^ 0x55 =  0b 0  1  1  0  0  1  1  0
```

bitwise exclusive or

in other words, calculates which bits are different in the operands

```
int A =          0b00001111;
int B =          0b00110011;
int C = A ^ B; // 00111100
```

exclusive or

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

^ can be used to *toggle* (invert) some subset of bits in an integer

```
x = y ^ 0b00001111; // invert the LS four bits of y
```

bitwise not (one's complement)

invert all the bits in an integer operand

not

a	~a
0	1
1	0

```
unsigned char A =      0b00001111;
unsigned char B = ~A; // 11110000
```

left and right shift

shift operators move all the bits in an integer to the left or right

the right hand operand is the distance (number of bit positions) to move the integer

`x << y` is the value of `x` with all the bits moved left `y` places
`x >> y` is the value of `x` with all the bits moved right `y` places

```
int A =           0b00111100;
int B = A << 2; // 11110000
int C = A >> 2; // 00001111
```

useful pattern: << and >> can be used to multiply or divide by a power of 2

```
x = y << n; // multiply y by 2^n (move binary point right)
x = y >> n; // divide y by 2^n (move binary point left)
```

combining shifts and bitwise operators

useful pattern: `1 << n` is equal to 2^n

```
int i = 1 << 5;  // i == 0b00100000 == 32 == 2^5
```

useful pattern: `(1 << n) - 1` has

- the `n` least significant (LS) bits set to `1`
- all other bits set to `0`

```
int i = (1 << 5) - 1;       // i = 0b00011111
int k = j & (1 << n) - 1;   // get the n LS bits of j
```

(note: `(1 << n)` is a `1` with `n` zeros after it; `(1 << n) - 1` is `0` with `n` ones after it)

assignment operators and expressions

assignment expressions of the form

`x = x` *binary-operator* `(`*expression*`)`

which repeat the same variable `x` on the left and right have a shorthand form

`x` *binary-operator* `=` *expression*

this shorthand works for these binary operators: `+ - * / % << >> & | ^`

shorthand form	*equivalent to*
`i += 2`	`i = i + (2)`
`i -= j + k`	`i = i - (j + k)`
`i *= b - 2 * a * c`	`i = i * (b - 2 * a * c)`
etc...	

note: the left operand can be any 'assignable' value, for example, an array element

```
for (i = 0;  i < N;  ++i)
  a[i] += b[i]; // add array b to array a
```

conditional expression

a conditional statement chooses between two statements based on a condition

```
if (a < b)
  minimum = a;
else
  minimum = b;
```

a *conditional expression* chooses between two expressions based on a condition

```
minimum = (a < b) ? a : b;     // minimum of a and b

magnitude = (n < 0) ? -n : n; // absolute value of n

int shift(int i, int n) { // shift i by n, left or right
  return (n > 0) ? i << n : i >> n;
}
```

comma operator

left-operand , *right-operand*

the comma operator

- evaluates the left operand
- discards the result
- evaluates the right operand

the result is the value of the right operand

the comma operator has the lowest precedence of all

useful in loops:

```
while ((n = from[n]))
  to[n++] = c, ++n;    // do two things in one statement

return to[n] = 0, n;   // number of bytes copied
```

precedence of all operators (including some we have not seen)

	operators	*associativity*
highest precedence	`()  []  ->  .`	left to right
(unary)	`! ~ ++ -- + - * & (`*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
lowest precedence	`,`	left to right

order of evaluation

C does not specify the order of evaluation of operands

- except for `&&`, `||`, `?:`, and `,`

example: in '`x = f() + g();`' the function `g` might be called before or after `f`

the value of `x` will be *undefined* if

- either `g` or `f` modify a global variable, and
- the other function depends on the value of that variable

never use a variable twice in an expression if anything might modify it

```
a[i] = i++;                          // the index in a[i] is undefined
a[i++] = i;                          // the value assigned to a[i] is undefined
printf("%d %d\n", n++, power(2, n)); // 2nd argument of power is undefined
i = i - i++;                         // undefined: (i+1)-i or i-i
```

the compiler will usually warn about these situations

next week: control flow

semicolons and expression statements
`if`, `else`, `else if`; binary search
`switch`, `break`; count digits, whitespace, others
`while` and `for`; infinite loops
nested loops; sorting, sieve for primes
`reverse()` using comma operator
`do` loop
`break` and `continue`
simplifying loop conditions with `break`
labels and `goto`

exercises

please download assignment from

MS Team “Information Processing 2”, “General” channel

exercises

converting a string to lower case

counting ‘1’ bits

removing characters from strings

detecting characters in strings

removing sets of characters from strings

getting a single bit from an integer

getting multiple bits from an integer

setting a single bit in an integer

clearing a single bit in an integer

setting multiple bits in an integer

multiple-precision arithmetic

12:40	5	last week: variables, constants, `enum`
12:45	5	declaration and initialisation
12:50	5	initialisation of arrays
12:55	5	operators: binary, unary
13:00	5	arithmetic operators
13:05	5	relational operators
13:10	5	logical operators
13:15	5	increment and decrement operators
13:20	10	bitwise operators
13:30	5	left and right shift
13:35	10	combining bitwise operators and shifts
13:45	5	assignment operators
13:50	5	conditional expression
13:55	5	comma operator
14:00	5	precedence of all operators
14:05	5	order of evaluation
14:10	10	*break*
14:20	90	exercises
15:50	00	end