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

京都先端科学大学

Information Processing 2 04

number representations
unsigned and signed binary numbers
data types and sizes, number range
constants, type conversions

Ian Piumarta

Faculty of Engineering, KUAS

last week: functions, line input, character arrays

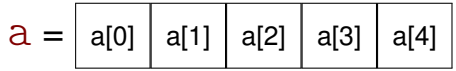
1.7 Functions
1.8 Arguments-Call by Value
1.9 Character Arrays
1.10 External Variables and Scope

functions, `return`, call-by-value
character arrays
`getchars` and character array `copy`
print longest line

character arrays

character arrays store characters

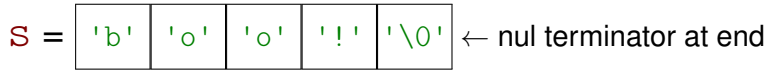
```
char a[5];  // an array containing 5 characters
```



a string is a kind of character array

- it contains one character per element
- the last element **must** be a 'nul' character with value 0 (often written `'\0'`)
- the nul character *terminates* the string (indicates the end of the string)
- a character array can be initialised with a string (including nul, if there is space)

```
char s[5] = "boo!";  // a holds a string of 4 characters
```



functions

return-type function-name (*parameter-declarations*)
{
  declarations
  statements
}

return-type *function-name* *parameter-declarations*

```
int power(int base, int exponent)
{
  int result = 1;

  while (exponent > 0) {
    result *= base;
    --exponent;
  }
  return result;
}
```

← *declarations*

} *statements*

this 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

variable names

variable names contain one or more letters and digits

first character must be a letter

- the underscore character '_' counts a letter

variable names cannot be keywords

- `if`, `else`, `return`, `int`, `char`, etc.

`y`

`_`

`2`

`xy`

`x_`

`x2`

`yz`

`_z`

`2z`

positional number representations

each digit position in a number has a different significance or *weight*

each weight is related to the next one by a constant multiplier

- the multiplier is called the base or radix r of the number
- position 0, immediately before the radix point, has weight 1 (r^0)
- weight is multiplied by r when moving left, divided by r when moving right

$$r^2 \quad r^1 \quad r^0 \quad . \quad r^{-1} \quad r^{-2}$$

radix point
(or *'separator'*)

the value of the digit d_i in position i is

- the digit multiplied by the weight of its position, $d_i \times r^i$

the value of a number is the sum of the values of all the digits: $\sum d_i \times r^i$

decimal system (base 10)

decimal numbers have base (radix) $r = 10$

- in Latin, “decima” means “one tenth part”

there are 10 symbols for the digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9

position weights are powers of 10

$$\begin{array}{ccccccccccc} \ldots & 10^{3} & 10^{2} & 10^{1} & 10^{0} & . & 10^{-1} & 10^{-2} & 10^{-3} & \ldots \\ & 1000 & 100 & 10 & 1 & & \frac{1}{10} & \frac{1}{100} & \frac{1}{1000} & \end{array}$$

decimal point

the decimal radix is implied (or denoted by the subscript 10 when ambiguous)

e.g., converting the decimal number 36.25_{10} to decimal:

$$\begin{array}{ccccccccc} 3 & & 6 & . & 2 & & 5 & & \\ 10^{1} & & 10^{0} & & 10^{-1} & & 10^{-2} & & \\ 3 \times 10 & + & 6 \times 1 & + & 2 \times 0.1 & + & 5 \times 0.01 & = & 36.25 \end{array}$$

octal system (base 8)

octal numbers have radix $r = 8$

- in Latin, “octo” means eight

8 symbols are used for the digits: 0, 1, 2, 3, 4, 5, 6 and 7

position weights are powers of 8

$$\begin{array}{ccccccccccc} \ldots & 8^3 & 8^2 & 8^1 & 8^0 & . & 8^{-1} & 8^{-2} & 8^{-3} & \ldots \\ & 512 & 64 & 8 & 1 & & \frac{1}{8} & \frac{1}{64} & \frac{1}{512} & \end{array}$$

octal point

the octal radix is denoted by the subscript 8

e.g., converting the octal number 44.2_8 to decimal:

$$\begin{array}{ccccccc} 4 & & 4 & . & 2 & & \\ 8^1 & & 8^0 & & 8^{-1} & & \\ 4 \times 8 & + & 4 \times 1 & + & 2 \times \frac{1}{8} & = & 36.25 \end{array}$$

hexadecimal system (base 16)

hexadecimal numbers have radix $r = 16$

- in Greek, “hexa” means six (plus Latin “decima” meaning tenth)

16 symbols are used for the digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F

position weights are powers of 16

$$\begin{array}{cccccccccc} \ldots & 16^3 & 16^2 & 16^1 & 16^0 & . & 16^{-1} & 16^{-2} & 16^{-3} & \ldots \\ & 4096 & 256 & 16 & 1 & & \frac{1}{16} & \frac{1}{256} & \frac{1}{4096} & \end{array}$$

hexadecimal point

the hexadecimal radix is denoted by the subscript 16

e.g., converting the hexadecimal number 24.4_{16} to decimal:

$$\begin{array}{ccccccc} \mathsf{2} & & \mathsf{4} & \mathsf{.} & \mathsf{4} & & \\ 16^1 & & 16^0 & & 16^{-1} & & \\ 2 \times 16 & + & 4 \times 1 & + & 4 \times \frac{1}{16} & = & 36.25 \end{array}$$

hexadecimal	*decimal*
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

binary system (base 2)

binary numbers have radix $r = 2$

- in Latin, “bini” means “two together”

there are 2 symbols for the digits: 0 and 1

position weights are powers of 2

each binary digit is called a *bit*

a 32-bit number contains 32 binary digits

$$\begin{array}{ccccccccccc} \ldots & 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} & \ldots \\ & 8 & 4 & 2 & 1 & & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \end{array}$$

binary point

the binary radix is denoted by the subscript 2

e.g., converting the binary number 100100.01_2 to decimal:

$$\begin{array}{ccccccccc} 1 & 0 & 0 & 1 & 0 & 0 & . & 0 & 1 \\ 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & 2^{-1} & 2^{-2} \end{array}$$

$$1 \times 32 \; + \; 0 \times 16 \; + \; 0 \times 8 \; + \; 1 \times 4 \; + \; 0 \times 2 \; + \; 0 \times 1 \; + \; 0 \times \tfrac{1}{2} \; + \; 1 \times \tfrac{1}{4} \quad = 36.25$$

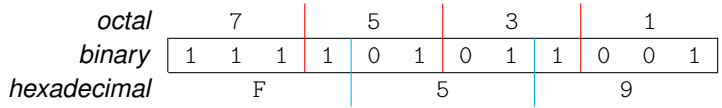
radix conversion: binary, octal and hexadecimal

one octal digit represents exactly three binary digits ($8^1 = 2^3$)

one hexadecimal digit represents four binary digits ($16^1 = 2^4$)

converting between binary and either octal or hexadecimal is easy

- octal: convert bits in groups of three
- hexadecimal: convert bits in groups of four



to convert between octal and hexadecimal

- first convert to binary

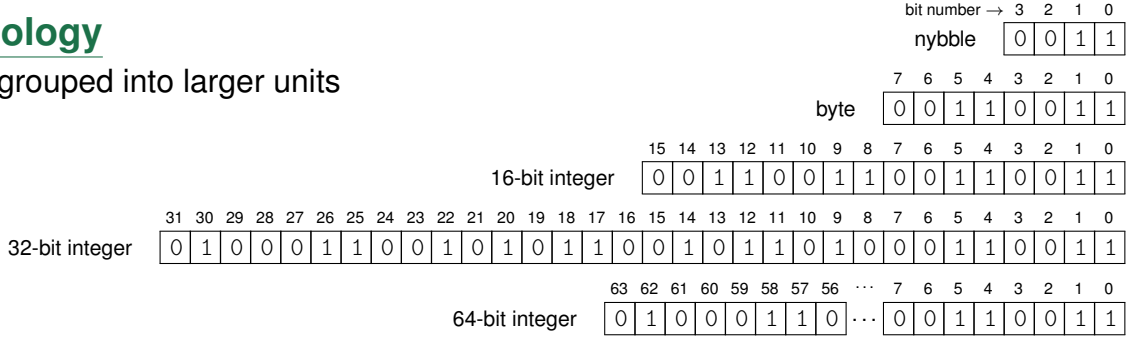
recommendation:

- keep practising conversions between binary and hexadecimal (:-()
- until you can do it *without looking at the table* (:-))

octal	*binary*	*hexadecimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
	1000	8
	1001	9
	1010	A
	1011	B
	1100	C
	1101	D
	1110	E
	1111	F

terminology

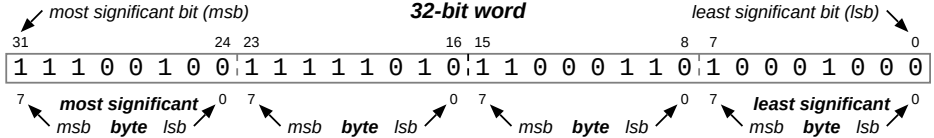
bits are grouped into larger units



an `int` is usually the natural size of the computer hardware's integers

the leftmost bit is the *most significant bit* (msb)

the rightmost bit is the *least significant bit* (lsb)



signed numbers

in an `unsigned` binary number, all weights are positive

- a 4-bit unsigned number has weights: $2^3\ 2^2\ 2^1\ 2^0$

in a `signed` binary number, the largest weight (leftmost bit) is negated

- a 4-bit signed number has weights: $-2^3\ 2^2\ 2^1\ 2^0$

for example, if $n = 8$ then

- *unsigned* $10001001_2 = \quad 2^7 + 2^3 + 2^0 = \quad 128 + 8 + 1 = \quad 136_{10}$
- *signed* $10001001_2 = -2^7 + 2^3 + 2^0 = -128 + 8 + 1 = -119_{10}$

the most significant (leftmost) bit indicates the *sign* of the number

- sign bit $0 \Rightarrow$ positive number
- sign bit $1 \Rightarrow$ negative number

data types and sizes

`char`	one byte (8 bits) capable of holding one character
`int`	integer of ‘natural’ size (according to the computer) typically 32 bits
`float`	low-precision floating point number, typically 32 bits
`double`	high-precision floating point number, typically 64 bits

`int` can also be modified with qualifier `short` or `long`

`short int`	between `char` and `int`, typically 16 bits
`long int`	larger than `int`, typically 64 bits

(you can omit the word ‘`int`’ if you want to, and most people do)

integer types can also be `signed` (the default) or `unsigned`

number range

whether or not `char` is signed is *platform dependent* (CPU, compiler, OS)

`signed char`	$n = 8$	10000000_2	-2^{n-1}	-128_{10}	minimum
		01111111_2	$2^{n-1} - 1$	127_{10}	maximum
`unsigned char`	$n = 8$	00000000_2	0	0_{10}	minimum
		11111111_2	$2^n - 1$	255_{10}	maximum
`signed short`	$n = 16$	1000000000000000_2	-2^{n-1}	-32768_{10}	minimum
		0111111111111111_2	$2^{n-1} - 1$	32767_{10}	maximum
`unsigned short`	$n = 16$	0000000000000000_2	0	0_{10}	minimum
		1111111111111111_2	$2^n - 1$	65536_{10}	maximum
`signed int`	$n = 32$	$1000000\ldots0000000_2$	-2^{n-1}	-2147483648_{10}	minimum
		$0111111\ldots1111111_2$	$2^{n-1} - 1$	2147483647_{10}	maximum
`unsigned int`	$n = 32$	$0000000\ldots0000000_2$	0	0_{10}	minimum
		$1111111\ldots1111111_2$	$2^n - 1$	4294967295_{10}	maximum
`signed long`	$n = 64$	$1000000\ldots0000000_2$	-2^{n-1}	$-9223372036854775808_{10}$	minimum
		$0111111\ldots1111111_2$	$2^{n-1} - 1$	9223372036854775807_{10}	maximum
`unsigned long`	$n = 64$	$0000000\ldots0000000_2$	0	0_{10}	minimum
		$1111111\ldots1111111_2$	$2^n - 1$	$18446744073709551615_{10}$	maximum

constants

integer constants such as `1234` have type `int`

writing `L` at the end makes it a `long` integer constant: `1234L`

writing `U` at the end makes it an `unsigned` integer constant: `1234U`

you can combine both for `unsigned long`: `1234UL`

floating point constants such as `1.23` have type `double`

writing `F` at the end makes it a `float` constant: `1.23F`

octal, hexadecimal, and binary constants

the examples above use decimal notation

integers that begin with `0` are treated as octal, therefore: `65 == 0101`

integers that begin with `0x` are treated as hexadecimal, therefore: `65 == 0x41`

integers that begin with `0b` are treated as binary, therefore: `65 == 0b01000001`

the following all represent the integer 42: `'*' 0x2A 0b00101010 052`

character constants

character constants such as `'X'` have type `int`

characters can also be written using their octal or hexadecimal ASCII code

character:	`'A'`	`'\081'`	`'\x41'`	← this notation is especially useful within strings
value:	65_{10}	81_8	41_{16}	

some characters can be written as *escape sequences*, such as '`\n`' (newline)

`\a`	bell	`\b`	backspace	`\f`	form feed
`\n`	newline	`\r`	carriage return	`\t`	tab
`\v`	vertical tab	`\'`	single quote	`\"`	double quote
`\\`	backslash	`\`*ooo*	octal code	`\x`*hh*	hexadecimal code

e.g: asterisk has character code 42 and is written `'*'` = `'\052'` = `'\x2A'`

string constants

zero or more characters surrounded by double quotes: `"I am a string"`

to place a double quote character inside a string, *escape* it

```
"the double quote \" delimits a string"
```

string constants are *concatenated* at compile time, therefore

```
"hello, world"
```

is the same as

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

(which is useful for splitting strings over multiple lines)

within strings, \u*hhhh* and \U*hhhhhhhh* represent 16- and 32-bit Unicode characters

```
printf("welcome to \u4eac\u90fd!\n");
```

named integers: enumerated constants

list of constant integer values: `enum boolean { FALSE, TRUE };`

- the first element is 0
- successive elements increase by 1
- unless explicit values are specified

```
enum escapes {
  BELL = '\a',  BSP  = '\b',  TAB  = '\t',
  NL   = '\n',  CR   = '\r',  ESC  = '\033',
};

enum hex { A = 10, B, C, D, E, F };
```

`enum`s associate constant values with names, similar to `#define`

compared to `#define`, `enum`s have the advantages that

- successive values can be generated automatically
- in some situations, the compiler will warn when `enum` constants are misused

constant expressions

expressions involving only constants are *constant expressions*

they can be evaluated at compile time

constant expressions can be used anywhere a constant is required

```
char s[256], t[128*2];

int i = 2;
char u[128*i];    // illegal: 128*i is not constant

enum { i = 2 };   // or: const int i = 2;
char v[128*i];    // OK: 128*i is constant
```

arithmetic conversions

most operators require operands of the same type

- “narrower” types are converted to “wider” types
- the type of the result is the same as the “wider” type

the rules are as follows:

- if either operand is `double`, convert the other to `double`, otherwise
- if either operand is `float`, convert the other to `float`, otherwise
- if either operand is `long`, convert the other to `long`, otherwise
- convert `char` and `short` to `int`, then
- if either operand is `unsigned`, convert the other to `unsigned`

e.g: `3 * 4.0` has type `double`

example arithmetic conversions

```
3   + 4
3   + 4U
3.0 + 4
3.0 + 4.0f
3   + 4.0f
3L  + 4U

'A' +  1
'A' + '1'

short s, t;

s + t
```

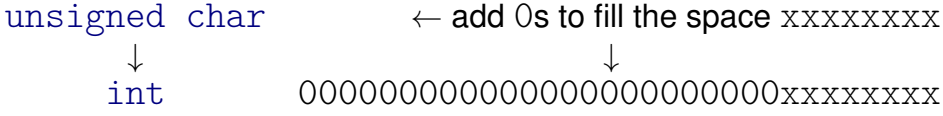
- if either operand is `double`, convert the other to `double`, otherwise
- if either operand is `float`, convert the other to `float`, otherwise
- if either operand is `long`, convert the other to `long`, otherwise
- convert `char` and `short` to `int`, then
- if either operand is `unsigned`, convert the other to `unsigned`

note: conversions are also performed to make function arguments match the their corresponding parameter's declared type

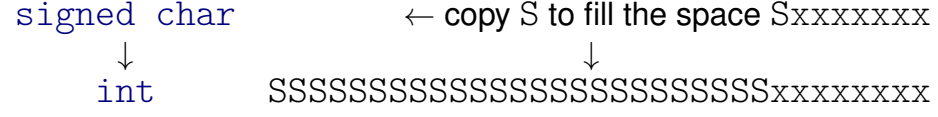
```
char myFunction(int i, double d) { return i + d; }
myFunction('*', 42); // result type?
```

widening integers

unsigned integers are widened by adding 0s to the left: *zero extension*



signed integers are widened by copying the sign bit to the left: *sign extension*



assignment type conversions

in assignments, the right hand side is converted to the type of the left hand side

- wider integers are converted to narrower ones by discarding most significant bits
- floating-point values are converted to integers by discarding the fractional part

```
int   i = 0x1234;
char  c = i;        // c is now 0x34 (most significant bits discarded)

float f = 2.99;
int   i = f;        // i is now 2 (fractional part discarded)


int   i = 3;
float f = 3.0;

i = (i * 3 / 2) * 3;  // i == ?
i = (f * 3 / 2) * 3;  // i == ?
```

explicit type conversions: casts

a value can be *coerced* from one type to another using a *cast*

$(type)\,expression$

which converts the result of *expression* to the given *type*

```
int i = 255;          // 00000000000000000000000011111111
printf("%d\n", i);    // 255

i = (signed char)i;   // int -> signed char -> sign extend 11111111 -> -1
printf("%d\n", i);    // -1

float f = (int)2.99; // 2.99 truncated -> 2 -> float 2.0
printf("%f\n", f);    // 2.000000

printf("%d\n", (unsigned char)0x1001);   // ??
```

exercise preparation: dividing by the radix (base)

dividing a number by its radix is the same as moving the separator to the left

consider a decimal number, base $r = 10$

$$\begin{array}{rcl} & & 2\ 3\ 5\ 9\ . \\ \div 10 & = & 2\ 3\ 5\ .\ 9 \\ & & \uparrow \end{array}$$

$abcd \div 10 = abc$ remainder d

if you have a decimal integer $abcd$ how would you

- remove the least significant ('units') digit from the number, leaving only abc?
- remove all but the least significant ('units') digit, leaving only d?

(in C, the "divide" operator is '/' and the "remainder" operator is '%')

exercise preparation: printing decimal digits

in a small embedded system you might not have library functions like `printf`

knowing how to convert numbers to strings could be important

character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
integer code:	48	49	50	51	52	53	54	55	56	57

what does `putchar('0' + 3)` print?

if you have an integer between `0` and `9`

- how do you print the corresponding digit character?

```
void printDigit(int n) // 0 <= n <= 9
{
  putchar(?);  // print the digit character for the integer n
}
```

exercise preparation: printing integers in decimal

how would you write a function to print an entire integer `abcd` in decimal?

```
void printNumber(int abcd) { ... }
```

hints:

- you already know how to print the units digit
- you already know how to remove the units digit from a number
- combine the two and you can print all the digits of a number

“divide and conquer” strategy...

exercise preparation: printing integers in decimal

"divide and conquer" strategy...

to **print an integer** `abcd` in decimal:

1. solve one very easy part of the problem
   e.g., **print one digit** `d`, the least significant ('units') digit of `abcd`

2. convert the remaining part of the problem into a similar, smaller problem
   e.g., **remove the units digit** from `abcd` leaving `abc`

3. solve the smaller problem, *if necessary*, using the solution to the large problem
   e.g., *if it is not empty* then **print the** smaller **integer** `abc` in decimal

one trivial change makes the function work perfectly (◡‿◡)

next 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

Chapter 2. Types, Operators, and Expressions
2.4 Declarations
2.5 Arithmetic Operators
2.6 Relational and Logical Operators
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

exercises

please download assignment from

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

exercises

printing binary numbers

printing hexadecimal numbers

printing numbers in any base

printing signed numbers

converting digit characters to values

reading digits in any base

reading numbers in any base

reading signed numbers in any base

13:00 5 last week: characters, functions
13:05 5 variable names
13:10 5 positional number systems
13:15 5 decimal
13:20 5 octal
13:25 5 hexadecimal
13:30 5 binary
13:35 5 radix conversion
13:40 5 terminology
13:45 5 signed numbers
13:50 5 data types and sizes
13:55 5 number range
14:00 5 integer constants: decimal, octal, hexadecimal, binary
14:05 5 character constants, string constants
14:10 5 enumerated constants
14:15 5 constant expressions
14:20 5 type conversions, extension
14:25 5 assignment and casts

14:30 10 *break*

14:40 15 exercise preparation: printing decimal integers
14:55 75 exercises
16:10 00 end