

Session 2

Data Memory Representation



This work is licensed under a Creative Commons Attribution – NonCommercial – NoDerivatives 4.0 International License.

Objectives

- Analysing how **data are represented** in the computer
 - Number representation, positional system and binary system
 - int (two's complement), float (IEEE 754) and char (ASCII)
 - Arithmetic operations with integer and floating-point numbers
- **Data conversion** between different data types

Rules of implicit and explicit conversions (cast)
- Specific operators to **manipulate bits** of data

Logical operators, bit shift operators and ones' complement

0101
zero one

Number Representation

Information

- Information can take several **different forms**

Number, instruction, image, sound, video, etc.
- The bit is the **elementary information**
 - 0 or 1 (binary digit)
 - Encoding to map external/internal representation ($A \leftrightarrow 65\dots$)
- **Two kinds** of information are dealt with by the computer
 - **Instructions** are executed by the computer
 - **Data** are manipulated by these latter to perform computation

Bit Grouping

- **Byte** (*B*) sometimes referred to as *char*
 - Set of adjacent bits, often 8, but 6, 7, 9 also exist
 - Smallest data unit accessible through a data bus
- *Eight-bit byte* (*o*) sometimes referred to as **octet**
A byte with eight bits encoding a given information
- **Word** groups bytes or octets together
We can obtain 32-bit words with four octets/word

char Size

- Size of a **char** can be found in the `limits.h` file

Number of bits for a char stored in the CHAR_BIT constant

- Defines the **length of a byte** for a specific machine

The value will typically be 8 on most modern system

```
1 #include <limits.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     printf("A char has %d bits.\n", CHAR_BIT);
7
8     return 0;
9 }
```

Amount of Information

- Two kinds of **amounts** of information used in computer science
 - Memory capacity, disk space, file size, etc. (GB)
 - Speed and transfer speed (Gbps, Gb/s)
- Two kinds of **prefix** can be used with these units
 - **International System of Units** (SI): powers of 10
 - **International Electrotechnical Commission** (IEC): binary

Binary Prefix

- First definition IEC 60027-2 or more recent **IEEE 1541-2002**

Used by Ubuntu >10.10 and macOS >10.6

- Using **binary prefixes** instead of powers of 10

Greater difference with large factors

SI Prefix			Binary Prefix			Error
kilo	kB	10^3 B	kibi	KiB	2^{10} B	2%
mega	MB	10^6 B	mébi	MiB	2^{20} B	5%
giga	GB	10^9 B	gibi	GiB	2^{30} B	7%
tera	TB	10^{12} B	tébi	TiB	2^{40} B	10%
peta	PB	10^{15} B	pébi	PiB	2^{50} B	13%
exa	EB	10^{18} B	exbi	EiB	2^{60} B	15%
zetta	ZB	10^{21} B	zébi	ZiB	2^{70} B	18%
yotta	YB	10^{24} B	yobi	YiB	2^{80} B	21%

Legal Disputes

- Hard drives are sometimes sold following decimal meaning...

A “160 GB” hard drives have 160 billions bytes

- But some operating systems just show “149.01 GB”

They are counting using binary sense but not showing it so...

- A 100 GB hard drive “only” has 93.13 GiB

- Class action lawsuits against digital storage manufacturers
- e.g. Safier v. Western Digital Corporation, July 7, 2005...
- Added disclaimer on packaging, compensation for customers

1024 or 1000?

THERE'S BEEN A LOT OF CONFUSION OVER 1024 VS 1000,
KBYTE VS KBIT, AND THE CAPITALIZATION FOR EACH.

HERE, AT LAST, IS A SINGLE, DEFINITIVE STANDARD:

SYMBOL	NAME	SIZE	NOTES
kB	KILOBYTE	1024 BYTES OR 1000 BYTES	1000 BYTES DURING LEAP YEARS, 1024 OTHERWISE
KB	KELLY-BOOTLE STANDARD UNIT	1012 BYTES	COMPROMISE BETWEEN 1000 AND 1024 BYTES
KiB	IMAGINARY KILOBYTE	1024 JFI BYTES	USED IN QUANTUM COMPUTING
kb	INTEL KILOBYTE	1023.937528 BYTES	CALCULATED ON PENTIUM F.P.U.
Kb	DRIVEMAKER'S KILOBYTE	CURRENTLY 908 BYTES	SHRINKS BY 4 BYTES EACH YEAR FOR MARKETING REASONS
KBa	BAKER'S KILOBYTE	1152 BYTES	9 BITS TO THE BYTE SINCE YOU'RE SUCH A GOOD CUSTOMER

Bit Sequence

- Information represented by **bit sequences**
Typically words or sequences of bytes/octets
- **Interpretation** according to type of encoded information
Distinction to make between binary number and bit sequence
- **Hexadecimal notation** to shorten long sequences
A nibble (4 bits) represents a hexadecimal digit

0010 1101 0100
 \u2014\u2014\u2014\u2014
 2 D 4

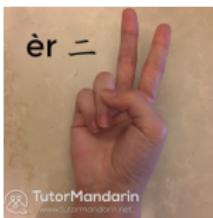
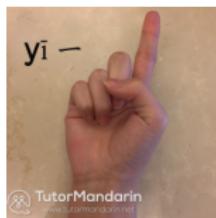
Number

- Concept to evaluate and compare quantities
Can also be used to order items by numbering
- Number relation studied by arithmetic and number theory
Interaction through operations summarised by calculation rules
- Numbering system to represent numbers
Set of signs, words, gestures to write/state numbers

Chinese Number Gesture

- Chinese people can count from one to nine with only **one hand**

Ordering food at noisy market, tell the waiter number of people...



Unary System

- Juxtaposition of the corresponding amount of **a symbol**

One unique symbol available that represents the unit

- Definition of the **four basic operations**

- Addition** by concatenation: $||| + || = |||||$

- Subtraction** by withdrawal: $||| - || = |$

- Multiplication** by substitution: $||| \times || = |||||$

- Division** by substitution: $||| \div || = |$, with remainder $|$

Positional Notation

- Other **numbering systems**: binary, quinary, vigesimal...

Grouping units by packets following a numbering base

- **Positional system** to write down a number with digits
 - Position of the digit in the writing indicates weight assigned
 - System in base b requires b digits (from 0 to $b - 1$)
- **Value in base 10** of the number n written $d_{k-1} \dots d_1 d_0$ in base b

$$n = \left(\sum_{i=0}^{k-1} d_i b^i \right)_{10} = \left(d_0 + d_1 \cdot b + \dots + d_{k-1} \cdot b^{k-1} \right)_{10}$$

with $d_i \in \{0, \dots, b - 1\}$

Common Base

- Several **common bases** often used in computer science
 - Binary (base 2): bit vectors, flags, etc.
 - Octal (base 8): not much used anymore
 - Hexadecimal (base 16): colour codes, MD5 checksum, etc.
- For **example**, let's take the $n = (42)_{10}$ number
 - $n = (101010)_2 = 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 = (42)_{10}$
 - $n = (52)_8 = 5 \times 8^1 + 2 \times 8^0 = (42)_{10}$
 - $n = (2A)_{16} = 2 \times 16^1 + 10 \times 16^0 = (42)_{10}$ ($A \leftrightarrow 10\dots$)

Literal Form of int

- Writing a constant integer number with its **literal form**

Just write its digits one after the other, with a possible sign

- Using a **prefix** to determine the base
 - By default, integer literal written in base 10
 - Adding 0b for base 2, 0 for base 8 and 0x for base 16

```
1 int a = 0b101010;
2 int b = 0123;
3 int c = 0xABc;
4
5 printf("%d %d %d\n", a, b, c);      // Prints "42 83 2748"
```

Binary...



Binary System

- Number n written $b_{k-1} \dots b_1 b_0$ as a **sum of powers of two**

$$n = \left(\sum_{i=0}^{k-1} b_i 2^i \right)_{10} = \left(b_0 + b_1 \cdot 2 + \dots + b_{k-1} \cdot 2^{k-1} \right)_{10}$$

with $b_i \in \{0, 1\}$

- In computer science, **limited length** binary numbers

Due to memory limitation and computation performances

$\begin{matrix} MSB & & LSB \\ \downarrow & & \downarrow \\ 1011010100 & & \end{matrix}$

$$= (1 \cdot 2^9 + 0 \cdot 2^8 + \dots + 0 \cdot 2^0)_{10}$$

Integer Number

1
int

1
int

Integer Number Representation

- Representing integers on a **finite number of bits** k
 - Limits on the number of representable numbers
 - With k bits, there are only 2^k possibilities
- Several **constraints** influence the representation
 - Should it be possible to represent negative numbers?
 - Should all the available space be used or not?
 - Should some kinds of operations be easier to compute?

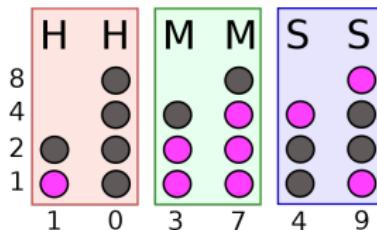
Binary-Coded Decimal (BCD)

- An integer is represented by a **sequence of nibbles**

Any decimal digit can be represented with 4 bits

- Encoding in **compacted or expanded** form in octets

This is using or not the two nibbles of an octet



Sign Bit

- Most significant bit used to indicate the sign

Positive integer with 0 and negative integer with 1

- Other bits represent the integer as a binary number

Bit usage: 1 sign bit and $k - 1$ information bits

- Space of representable integers of size $2 \cdot 2^{k-1} - 1$

- Integers between $-(2^{k-1} - 1)$ and $2^{k-1} - 1$

- Be careful, two possible representations for zero (-0 and $+0$)

00101010 : $(42)_{10}$

10101010 : $(-42)_{10}$

Ones' Complement

- Ones' complement of a binary number by inverting all its bits
 - Negative of original number for some operations
 - Ones' complement of n can be computed as $(2^k - 1) - n$
- Space of representable integers of size $2^k - 1$
 - Integers between $-(2^{k-1} - 1)$ and $2^{k-1} - 1$
 - Be careful, two possible representations for zero (-0 and $+0$)
 - First bit used to identify the sign of the integer

00000010 : $(2)_{10}$

11111101 : $(-2)_{10}$

Two's Complement (1)

- Two's complement of k -bit number with respect to 2^k
 - Ones' complement of n can be computed as $2^k - n$
 - Two's complement by adding 1 to the ones' complement
- Space of representable integers of size 2^k
 - Integers between -2^{k-1} and $2^{k-1} - 1$
 - First bit used to identify the sign of the integer

00000010 : $(2)_{10}$

11111110 : $(-2)_{10}$

Two's Complement (2)

- Two's complement eases fundamental **arithmetic operations**
 - Used to represent fixed point binary values
 - Simple operations on the underlying bits
- Automatic detection of **overflow**
 - Two operands with the same sign and result with opposite sign
 - No overflow with two opposite signs operands
- **Sign extension** to increase the number of bits

Just by repeating the sign bit (MSB)

Sum of Integers

- Sum of two integers with two's complement
 - Both encoded with the same number k of bits*
- Non representable sum results in an overflow
 - Overflow indicator as a bit in the processor*
- With $k = 6$ bits, possible to represents any $n \in [-32; 31]$

$$\begin{array}{r} 010100 \\ + 000101 \\ \hline 011001 \end{array} \quad \begin{array}{r} (20)_{10} \\ (5)_{10} \\ \hline (25)_{10} \end{array}$$
$$\begin{array}{r} 010100 \\ + 001111 \\ \hline 100011 \end{array} \quad \begin{array}{r} (20)_{10} \\ (15)_{10} \\ \hline (-29)_{10} \end{array}$$

Subtraction of Integers

- Subtraction of two integers with two's complement

Both encoded with the same number k of bits

- Just using addition with the negative representation

Overflow is also possible if the result is too small

- With $k = 6$ bits, possible to represent any $n \in [-32; 31]$

$$\begin{array}{r} 000111 \\ + 101101 \\ \hline 110100 \end{array} \quad \begin{array}{r} (7)_{10} \\ (-19)_{10} \\ (-12)_{10} \end{array} \quad \begin{array}{r} 101111 \\ + 101101 \\ \hline 011100 \end{array} \quad \begin{array}{r} (-17)_{10} \\ (-19)_{10} \\ (28)_{10} \end{array}$$

Multiplication of Integers

- Multiplication of two integers with two's complement
 - Just a succession of staggered additions
 - Be careful that the result can quickly overflow
 - Negate both operands if multiplier is negative
- With $k = 6$ bits, possible to represent any $n \in [-32; 31]$

$$\begin{array}{r} 000\textcolor{blue}{100} \\ \times 000101 \\ \hline 000\textcolor{blue}{100} \\ 000000 \\ + 0\textcolor{blue}{10000} \\ \hline 010100 \end{array} \quad (4)_{10} \quad \quad \begin{array}{r} 111100 \\ \times 000101 \\ \hline 111100 \\ 000000 \\ + 110000 \\ \hline 101100 \end{array} \quad (-4)_{10}$$

$(5)_{10} \qquad \qquad \qquad (5)_{10} \qquad \qquad \qquad (-20)_{10}$

Division of Integers

- Division of two integers with two's complement
 - Integer division produces a quotient and a remainder
 - Dividing absolute values then computes sign
- With $k = 6$ bits, possible to represent any $n \in [-32; 31]$

$$\begin{array}{r} (27)_{10} & 0\ 1\ 1\ 0\ 1\ 1 & | & 1\ 0\ 1 & (5)_{10} \\ & \underline{1\ 0\ 1} & 0\ 0 & \underline{0\ 1\ 0\ 1} & (4)_{10} \\ & & & 1\ 1\ 1 & \\ & & & \underline{1\ 0\ 1} & \\ (2)_{10} & & & 1\ 0 & \end{array}$$

Integer Overflow

- Multiplication overflows can result in wrong results

Integer literal are of type int by default

- One solution is to use long literals constants

As long as type of variable is big enough for the value to compute

```
1 #include <stdio.h>
2
3 int main()
4 {
5     long MICROS_DAY = 24 * 60 * 60 * 1000 * 1000;
6     long MILLIS_DAY = 24 * 60 * 60 * 1000;
7
8     printf("%d\n", MICROS_DAY / MILLIS_DAY);    // Prints "5"
9
10    return 0;
11 }
```



Floating-Point Number

Binary Fraction

- Integer and fractional parts separated by a **radix point**

Typically the point (.) in the scientific world

- **Negative powers** of two for the fractional part

Fundamental arithmetic operations performed as previously

- Value in base 10 of **binary fraction** n written $l_{k-1} \dots l_1 l_0.r_1 r_2 \dots r_\ell$

$$\begin{aligned} n &= \left(\sum_{i=0}^{k-1} l_i 2^i + \sum_{i=1}^{\ell} r_i 2^{-i} \right)_{10} \\ &= \left(l_0 + l_1 \cdot 2 + \dots + l_{k-1} \cdot 2^{k-1} + r_1 \cdot \frac{1}{2} + \dots + r_\ell \cdot \frac{1}{2^\ell} \right)_{10} \end{aligned}$$

with $l_i, r_i \in \{0, 1\}$

Real Number Representation

- All **real numbers** cannot be represented

At least if the number of bits to represent the number is limited

- **Fixed point** representation with a binary fraction

Position of the radix point chosen by the programmer

- **Floating-point** representation similar to scientific notation

- Based on a significand scaled using an exponent in a given base
- Trade-off between range and precision

$$n = \text{significand} \times \text{base}^{\text{exponent}} \quad 1.2345 = 12345 \times 10^{-4}$$

IEEE 754

- Standardised norm to represent binary **floating-point number**

Most common norm in computer science, used by CPU and FPU
- **Representation format** and special values (infinities and NaN)

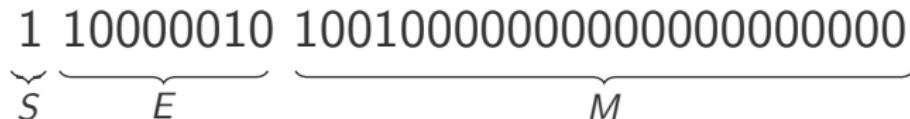
Sign, mantissa, exponent and denormalised number
- IEEE 754 defines **five basic formats** with extended formats
 - Single precision (32 bits with 24 for significand) float
 - Double precision (64 bits with 53 for significand) double
 - Double extended (≥ 79 bits, but often 80 bits) long double

Single Precision (1)

- Real number n encoded with **32 bits** in single precision
 - Sign bit S (1 bit): $s = \pm 1$
 - Exponent E (8 bits): $-126 \leq e \leq 127$
 - Fraction F (23 bits): $1 \leq 1.f < 2$
- **Mantissa** composed of an implicit leading bit and fraction bits

$$n = (-1)^S \times 2^{(E)_2 - 127} \times (1.F)$$

- $-12.5 = (-1)^1 \times 2^{130-127} \times 1.1001 = -1 \times 2^3 \times \left(1 + \frac{1}{2^1} + \frac{1}{2^4}\right)$

1  10010000000000000000000000000000

$\underbrace{}_S \quad \underbrace{}_E \quad \underbrace{}_M$

Single Precision (2)

- Denormalised number when zero E and non zero F

Smaller fractions than with normalised number, closer to 0

- Can represents infinity and special Not A Number (NaN)

Indeterminate operation (QNaN) or invalid operation (SNaN)

E	F	Type	Value
0	= 0	Zero	± 0
0	$\neq 0$	Denormalised number	$(-1)^S \times 2^{-126} \times 0.F$
Between 1 and 254		Normalised number	$(-1)^S \times 2^{(E)_2 - 127} \times 1.F$
255	= 0	Infinite	$\pm\infty$
255	$\neq 0$	NaN	quiet (QNaN) or signalling (SNaN)

IEEE 754 for -17.5

- Transform to $1.F$ by successively dividing 17.5 by 2

$$17.5 = 8.75 \times 2 = 4.375 \times 2^2 = 1.1875 \times 2^3 = 1.09375 \times 2^4$$

- Find the binary fraction F by successively multiplying by 2

$$1.09375 = \mathbf{1} + 0.09375$$

$$0.1875 = \mathbf{0} + 0.1875$$

$$0.375 = \mathbf{0} + 0.375$$

$$0.75 = \mathbf{0} + 0.75$$

$$1.5 = \mathbf{1} + 0.5$$

$$1 = \mathbf{1} + 0$$

$$\Rightarrow 1.09375 = 1 + 0.0625 + 0.03125 = 1 + \frac{1}{2^4} + \frac{1}{2^5} = (1.00011)_2$$

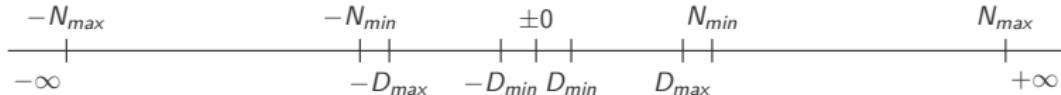
1 10000011 00011000000000000000000000000000
 $\underbrace{_S$ $\underbrace{_E = 131$ $\underbrace{_M$

Floating-Point Number

- Floating-point numbers are **equidistant** between powers of two
Same number of representable numbers between 2^k and 2^{k+1}



- Denormalised numbers used to get more precision close to 0
Remembering that not all real numbers can be represented



Double Precision

- Same representation principle as with single precision
With more bits for the exponent and fraction parts
- Real number n encoded with 64 bits in double precision
 - Sign bit S (1 bit): $s = \pm 1$
 - Exponent E (11 bits): $-1022 \leq e \leq 1023$
 - Fraction F (52 bits): $1 \leq 1.f < 2$
- More bits for the fraction part than for the exponent
The precision is more important than the amplitude

Multiplication of Floating-Point

- Addition of exponents and multiplication of mantissas

With possible (re)normalisation steps during calculation

- For example: $((1.10)_2 \times 2^{-3}) \times ((1.11)_2 \times 2^7)$
 - Exponent: $-3 + 7 = 4$
 - Mantissa: $(1.10)_2 \times (1.11)_2 = (10.1010)_2$
 - Result (before normalisation): $(10.1010)_2 \times 2^4$
 - Result (after normalisation): $(1.01010)_2 \times 2^5$

$$\Rightarrow (1.5 \times 2^{-3}) \times (1.75 \times 2^7) = 1.3125 \times 2^5 = 42$$

Division of Floating-Point

- Subtraction of exponents and division of mantissas

With possible (re)normalisation steps during calculation

- For example: $((1.01010)_2 \times 2^5) \div ((1.11)_2 \times 2^7)$

- Exponent: $5 - 7 = -2$
- Mantissa: $(1.01010)_2 \div (1.11)_2 = (0.11)_2$
- Result (before normalisation): $(0.11)_2 \times 2^{-2}$
- Result (after normalisation): $(1.1)_2 \times 2^{-3}$

$$\Rightarrow (1.3125 \times 2^5) \div (1.75 \times 2^7) = 1.5 \times 2^{-3} = 0.1875$$

Addition of Floating-Point

- **Addition** of mantissas

With denormalisation to bring back operands to same exponent

- For **example**: $((1.10)_2 \times 2^4) + ((1.11)_2 \times 2^6)$
 - Denormalisation: $(1.10)_2 \times 2^4 \rightarrow (0.011)_2 \times 2^6$
 - Mantissa: $(0.011)_2 + (1.11)_2 = (10.001)_2$
 - Result (before normalisation): $(10.001)_2 \times 2^6$
 - Result (after normalisation): $(1.0001)_2 \times 2^7$

$$\Rightarrow (1.5 \times 2^4) + (1.75 \times 2^6) = 1.0625 \times 2^7 = 136$$

Subtraction of Floating-Point

■ Subtraction of mantissas

- With denormalisation to bring back operands to same exponent
- May need to invert the operands and then fixing the sign
- For example: $((1.11)_2 \times 2^3) - ((1.101)_2 \times 2^7)$
 - Denormalisation: $(1.11)_2 \times 2^3 \rightarrow (0.000111)_2 \times 2^7$
 - Mantissa: $(1.101)_2 - (0.000111)_2 = (1.100001)_2$
 - Result (before normalisation): $(1.100001)_2 \times 2^7$
 - Result (after normalisation): $(1.100001)_2 \times 2^7$

$$\Rightarrow (1.75 \times 2^3) - (1.625 \times 2^7) = 1.515625 \times 2^7 = -194$$

Floating-Point Precision

- Not **all reals** can be represented with floating-point numbers

Approximations are made with literals and on computations

- One solution is to make computations **with cents**

Stored in integer number variables such as `int` or `long`

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float price = 3.26;
6     float paid = 100.00;
7     float change = paid - price;
8
9     printf("Change: %f\n", change);    // Prints "Change: 96.739998"
10
11 }
12 }
```

Floating-Point Comparison (1)

- Never compare floating-point numbers for (non-)equality

Due to rounding to closest representable floating-point number

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float result = 1.0 / 10.0;
6     printf("Value: %f\n", result);    // Prints "Value: 0.100000"
7     if (result == 0.1) {
8         printf("Okay!\n");
9     }
10
11     return 0;
12 }
```

Floating-Point Comparison (2)

- Test floating-point numbers **closeness** given an ε

Use the `fabs` function and the `FLT_EPSILON` constant

```
1 #include <float.h>
2 #include <math.h>
3 #include <stdio.h>
4
5 int almostEqual(float a, float b)
6 {
7     return fabs(a - b) <= FLT_EPSILON;
8 }
9
10 int main()
11 {
12     float result = 1.0 / 10.0;
13     printf("Value: %f\n", result);    // Prints "Value: 0.100000"
14     if (almostEqual(result, 0.1)) {
15         printf("Okay!\n");           // Prints "Value: Okay!"
16     }
17
18     return 0;
19 }
```

Character

Text

- Alphanumeric **characters** and special characters

Association of a numeric identity to each character

- Several **mapping tables** do exist and can be used

- ASCII (*American Standard Code for Information Interchange*) on 7 bits
- EBCDIC (*Extended Binary Coded Decimal Internal Code*) on 8 bits
- Unicode on 8, 16 then 32 bits

- Mapping tables are implemented with an **encoding**

- Coding/decoding a (sequence of) character(s) into bits
- Way to represent numeric identities with a sequence of bits

ASCII

- The ASCII **character table** (iso-646) contains 128 characters

Enough characters in this table for texts in English

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STH	ETH	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	CD2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	spc	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	—
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

For example, A \leftrightarrow (41)₁₆ = (65)₁₀

- Extended ASCII on 8 bits or more to have more characters

Basic ASCII extensions such as EBCDIC, for example

char Type

- Smallest unit of storage is the `char` occupying **one byte**
Typically following an extended ASCII encoding with 8 bits
- **Three categories** of characters can be defined
 - 95 human readable: letters, digits, punctuation marks
 - 33 not printable but still visible: space, line feed, tab, etc.
 - 128 special codes used by computer: start/end of text, etc.
- Can use some integer number **operations** with characters
Typically arithmetic and comparison operators

Print Alphabet

- Letters and digits follow each other in ASCII table

Alphabet can be printed with a loop, starting with 'A' to 'Z'

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char c;
6     for (c = 'A'; c <= 'Z'; c++)
7         printf("%c", c);
8     printf("\n");
9
10    return 0;
11 }
```

Data Conversion



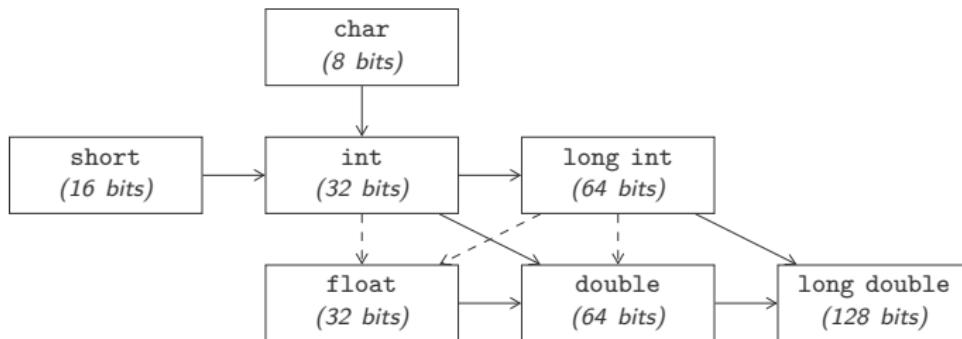
Data Conversion (1)

- Data can be **converted** from one type to another type

Same value but represented with another data type

- Only some conversions are possible and **make sense**

Depending on involved data types and size occupied in memory



Data Conversion (2)

- Two types of conversion between types are possible
 - Conversion without loss of information to a broader type
 - Conversion with loss of information to a narrower type
- Possible loss of precision converting to floating-point number

From int to float and from long int to double
- Two ways to convert data from one type to another one

Implicit conversion in several situations and explicit conversion

Implicit Conversion (1)

- Data can be implicitly converted in **three different situations**
 - By assigning a value to a variable
 - By arithmetic promotion during an operation
 - By compiler conversion
- **Implicit conversion** occurs without programmer intervention

Programmer is not always aware that such conversion occurred

```
1 short a = 10L;           // 10 converted from long to short
2 char b = 65;             // 65 converted from int to char
3 double c = 10 * b;       // b converted from char to int and then
                           // result of 10 * b converted from int to
                           // double
```

Implicit Conversion (2)

- Implicit conversion if compiler has information about type
 - Parameters on procedure/fonction calls
 - Returned values in functions
- Two operands converted to the same type before operation
 - The result of the operation is of the larger type
 - Smallest type used is int by integer promotion

Explicit Conversion

- Data can be **explicitly converted** with the cast operator

Possible to convert from any type to any other type

- Value after conversion can be completely **changed**

- The sequence of bits is simply truncated for integer numbers
- The closest value is chosen for floating-point numbers

- **Cast operator** is the destination type between parentheses

To be placed in front of the value to convert

```
1 int i = 1000;  
2 char c = (char) i;      // i converted from int to char: (result -24)
```

Information Loss

- Information can be lost when converting to a narrower type

Not always possible to represent same information with less bits

- The sequence of bits is truncated to fit the new types

Depending on the conversion, even the sign can change

```
int i = 1000;  00000000 00000000 00000011 11101000  
char c = (char) i;                      11101000
```

Precision Loss

- Precision can be lost when converting to floating-point

Because not all real numbers can be represented

- The order of magnitude after conversion is the same

But the converted number can lose precision

```
1 int j = 123456789;  
2 float f = j;           // 123456792.000000
```

Hidden Cast

- Hidden cast with compound operators on some situations

If type of left operand narrower than type of right one

- A cast is implicit and some information loss can occur

Avoid narrower type to the left or use full assignment

```
1 short x = 0;
2 int i = 123456;
3
4 x += i;
5 printf("%d\n", x);      // Prints "-7616"
```



Bit Manipulation

Bit Manipulation Operator

- Three **logical operators** at the binary level
 - `&`: binary “*AND*”
 - `|`: binary “*OR*” (inclusive or)
 - `^`: binary “*XOR*” (exclusive or)
- Two **bit shift** operators
 - `<<`: shift bits to the left with 0 padding
 - `>>`: shift bits to the right with 0 padding for `unsigned`
For signed, 0 padding if logical and 1 padding if arithmetical
- The `~` operator computes the **ones' complement**

Multiplication and Division

- Shifting bits of an integer **multiplies/divides** it by a power of 2

Left shift is a multiplication and right shift a division

- Only true with **arithmetic shifts** but not with logical shifts
 - Some issues may occur with negative integer numbers
 - Shifters are often way faster than dividers

```
1 int i = 13;
2
3 printf("%d\n", i << 2);      // Prints 52 (that is, 13 * 2^2)
4 printf("%d\n", i >> 1);      // Prints 6 (that is, 13 / 2)
5 printf("%d\n", -i << 2);     // Prints -52
6 printf("%d\n", -i >> 1);    // Prints -7
```

Bit Masking

- Binary “AND” operator can be used to **mask bits**

We have $(0 \& x)$ always gives 0 and $(1 \& x)$ always gives x

- **Retrieve n bits from x starting at position p**
 - First removing undesired bits to the right ($x >> (p+1-n)$)
 - Mask with n 1's to the right, then one zero and again 1's

```
1 unsigned char get_bits(unsigned char x, int p, int n)
2 {
3     return (x >> (p+1-n)) & ~(~0 << n);
4 }
```

Display Binary Representation

- Display the **sequence of bits** of the data in a variable

Analysing the value bit by bit, with bit manipulation operator

- Combination of **masking and bit shifting** operators

- Shift a bit to 1 under the desired position ($1 \ll i$)
- Apply the mask to extract it ($1 \ll i \& n$)
- Shift the obtained value to the right to recover the bit

```
1 void print_bits(unsigned char n)
2 {
3     char i;
4     for (i = 8 * sizeof(unsigned char) - 1; i >= 0 ; i--)
5         printf("%d", (1 << i & n) >> i);
6 }
```

Bit Vector (1)

- A **bit vector** stores boolean values on single bits
 - Saves memory space when it is limited
 - Stores multiple flags in a single variable
- **Reading the value of a bit** as in the previous example
Shifting a 1 to the right position of the bit and mask with &

```
1 unsigned char read_bit(unsigned char x, int n)
2 {
3     return (1 << n & x) >> n;
4 }
```

Bit Vector (2)

- Modification of a bit setting it to 0 or 1
 - Set to 1 with binary “OR”: $(1 \mid x)$ is always 1
 - Set to 0 with binary “AND”: $(0 \& x)$ is always 0
- Combination of the operators & or | with shifting of bits

So that the modification only changes one bit

```
1 unsigned char set_bit(unsigned char x, int n)
2 {
3     return x | (1 << n);
4 }
5
6 unsigned char reset_bit(unsigned char x, int n)
7 {
8     return x & ~(1 << n);
9 }
```

Flag Variable

- Define constants whose values are **distinct powers of two**

Each constant only has one bit set to one

- Select **multiple flags** by “OR”-ing the desired flags

Reading, modifying and clearing flags by manipulating bits

```
1 #include <stdio.h>
2
3 #define ALARM      1
4 #define HEATING    2
5 #define COOLING   4
6 #define LIGHTS    8
7 #define DOORLOCK 16
8
9 int main()
10 {
11     char config = HEATING | LIGHTS | DOORLOCK;
12     printf("%d\n", config);           // Prints "26"
13 }
```

Short-Circuit Property

- Logical operator has the **short-circuit property**
 - Right operand is only evaluated if it necessary
 - $(0 \&& x)$ is always 0 and $(1 \mid\mid x)$ is always 1
- Bit manipulation operator can be used as **logical operator**
Faster, but does not have the short-circuit property

```
1 int x = 1, y = 2;  
2  
3 printf("%d\n", x & y);           // Prints "0"  
4 printf("%d\n", x && y);         // Prints "1"
```

References

- Bradley Mitchell, *The Use of Octets in Computers and Networking: Octets are bytes ... sometimes*, June 21, 2019.
<https://www.lifewire.com/what-is-octet-818391>
- IEEE, *IEEE Standard for Prefixes for Binary Multiples*, September 18, 2009. (ISBN: 978-0-7381-3386-7)
- Safier v. Western Digital Corporation, Case No. 05-03353 BZ (N.D. Cal. Mar. 17, 2006).
- Vedraw, *Why Unary is the Best Number System*, November 29, 2015.
<https://medium.com/@veedrac/why-unary-is-the-best-number-system-cc1e0edfb928>
- Steve Hollasch, *IEEE Standard 754 Floating Point Numbers*, August 24, 2018.
<https://steve.hollasch.net/cgindex/coding/ieeefloat.html>
- David C. Zentgraf, *What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text*, April 27, 2015. <http://kunststube.net/encoding>
- Sean Eron Anderson, *Bit Twiddling Hacks*, May 5, 2005. <https://graphics.stanford.edu/~seander/bithacks.html>

Credits

- duncan c, November 1, 2015, <https://www.flickr.com/photos/duncan/35981681216>.
- Kilobyte, <http://xkcd.com/394>.
- 1 to 10, <http://www.xkcd.com/953>.
- f0rtytw0, November 29, 2014, <https://www.flickr.com/photos/entoropi/25488292585>.
- Alexander Jones & Eric Pierce, October 14, 2006, https://en.wikipedia.org/wiki/File:Binary_clock.svg.
- Kay Prior, October 2, 2009, https://www.flickr.com/photos/kay_prior/3977139291.
- Blondinrikard Fröberg, April 25, 2015, <https://www.flickr.com/photos/blondinrikard/17265715421>.
- Tom Saunders, May 30, 2017, <https://www.flickr.com/photos/tombass59/34155286024>.
- rjp, November, 5, 2007, <https://www.flickr.com/photos/zimpenfish/1880717616>.