Computer Programming

"First think, then program"

Prof. Henry F.Ledgard

Outline

- Programming style
  - Definition
  - Advice
- Digital Representations
  - Signed integers – signed magnitude; two's complement
  - Reals – floating point
- Variables
  - Declaration
  - Initialization
- Expressions
  - Definition
  - Evaluation

Programming style

- **Programming style**: set of general rules concerning the form of a program, and its implementation details. Characterized by:
  - Marking out the structure of a program
  - Program modularity
  - Data abstraction
  - Program clarity
  - Ease of further changes
  - Error handling
  - Generality of the solution

Programming style. Recommendations

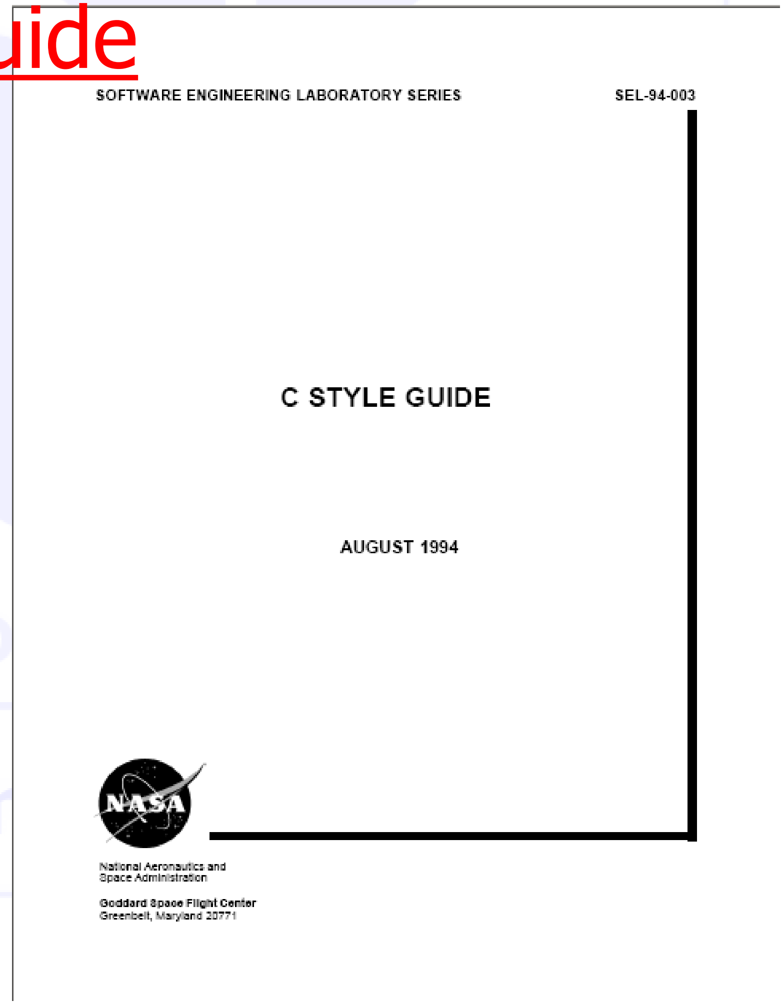
- Choose **meaningful names** for constants, types, variables, functions, etc.
- Use **indenting** to relieve flow control
- Use **comments** to document
  - Purpose of functions
  - Input and output data
  - Algorithm used
  - Comments are essential for maintenance
- **Avoid globals** – they have side effects
- **Avoid changing** a **for** loop control variable inside the loop body
- **Avoid forced exit** from loops
- Use as **few auxiliary variables** as possible

Programming style. Recommendations

- Do **NOT use un-initialized** variables
- **Check ranges** for variables
- **Avoid unclear** tricks
- Use **symbolic constants** if they occur many times, or the constants need to be changed later
- **Follow the algorithm** when coding
- **Postpone formatting** of output till you get the output correct, but do not forget it
- Proverbs:
  - Martin Fowler on style: “Any fool can write code that a computer can understand. Good programmers write code humans can understand”
  - Sue D. Nom on clarity: “Stupid programmer errors take hours to find. Smart programmer errors take days to find”

Programming style. Recommendations

- NASA C Style Guide


SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003

C STYLE GUIDE

AUGUST 1994

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Milestones in C's development as a language

- UNIX developed cca. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language ”B” a second attempt (cca. 1970).
- A totally new language ”C” a successor to ”B”. (cca. 1971)
- By 1973 UNIX OS almost totally written in ”C”.

Digital Representations Used in Computers

- **Fixed point representations**
- **Notation for signed integers:**

a_{n-1}	a_{n-2}	a_0
S	MSb	LSb

  - n = number of bits used in th representation (aka *precision*)
  - S =sign bit; MSb = most significant bit; LSb = least significant bit

Data Representation. Signed integers

- *Signed magnitude*

$$X = (-1)^{a_{n-1}} (2^{n-2} a_{n-2} + 2^{n-3} a_{n-3} + \ldots + a_0)$$

- Range: $[1-2^{n-1}, 2^{n-1}-1]$
- 2 representations for zero: 100..0, 000..0
- Examples

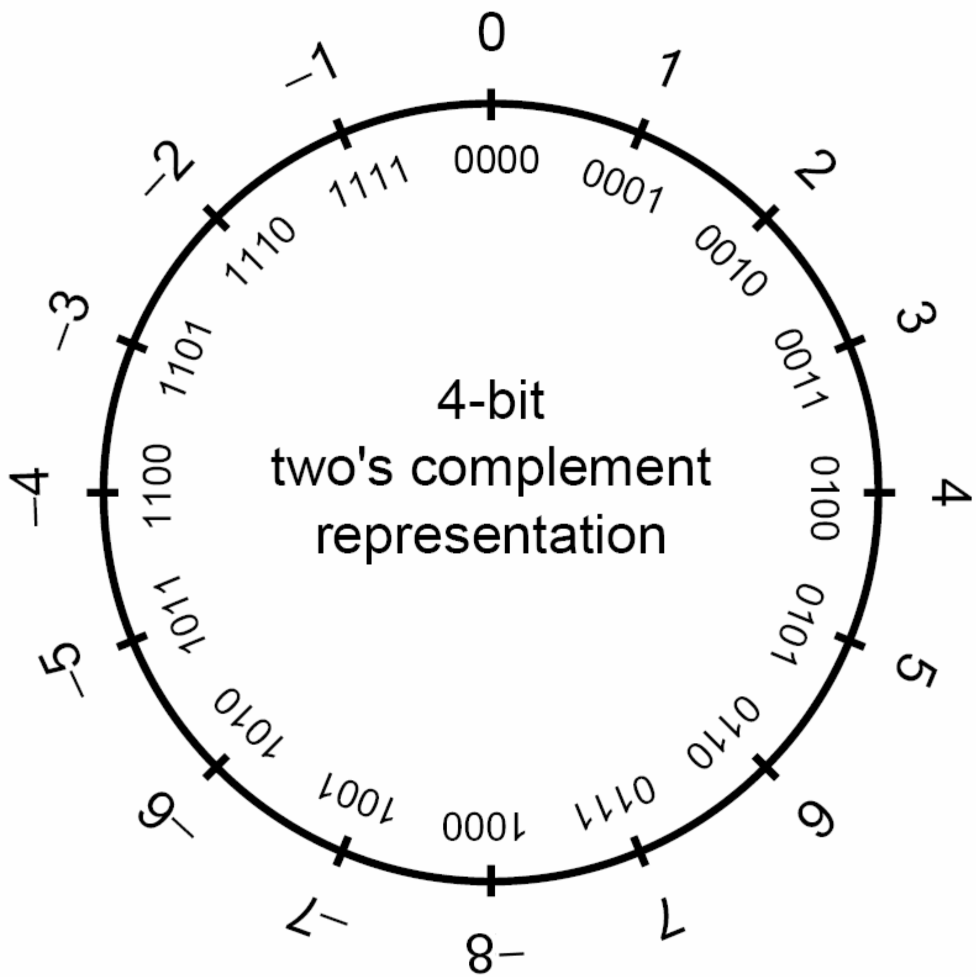
Data Representation. Binary Codes for Signed Integers

- ***Two's complement code***

$$X = (1 - 2^{n-1})a_{n-1} + 2^{n-2}a_{n-2} + 2^{n-3}a_{n-3} + \ldots + a_0)$$

- Range: $[-2^{n-1}, 2^{n-1}-1]$
- A single representation for zero: 000..0
- Examples
- Demo:DataReps.jar, datarep.jar (see Moodle site)

Two's Complement Code Example



Data Representation. Reals

- There are several possible exponential form representations of the same number
  - They vary in where the decimal point is placed, which determines the exponent value.
  - E.g. Given the number 4567000, it can also be written as:
    - 456.7×10^4
    - 45.67×10^5
    - 4.567×10^6 - *scientific notation*
    - $.4567 \times 10^7$ - *normalized exponential form*

Data Representation. Reals

- Computers represent real values in a form similar to that of scientific notation.
  - E.g. 1.23×10^4
    - The number has a *sign* (+ in this case)
    - The *significand* (1.23) is written with one non-zero digit to the left of the decimal point.
    - The *base* (*radix*) is 10.
    - The *exponent* (an integer value) is 4. It too must have a sign.

Scientific Notation for Numbers

- Examples

1 billion
= 1.000.000.000
= 1×10^9
  - Significand or mantissa: 1
  - base or radix : 10
  - exponent: 9

1999
= 1.999×10^3
  - Significand or mantissa: 1.999
  - base or radix : 10
  - exponent: 3

= 19.99×10^2
= 199.9×10^1

Scientific Notation for Numbers

- Three things change in scientific notation:
  - Sign
  - Absolute value of the mantissa
  - Exponent
- Radix 2:

  2.25_{10}

  $= 10.01_2$

  $= 10.01_2 \times 2^0$

  $= 1.001_2 \times 2^1$ ← normalized

Data Representation. Reals

- Real numbers are represented in floating point formats
- Most floating point formats employ scientific notation and use
  - some number of bits to represent a *mantissa* and
  - a smaller number of bits to represent an *exponent*.
  - floating point numbers can only represent numbers with a specific number of *significant* digits.
- Floating point numbers are *approximations* of real numbers
  - there are an infinite number of reals, and only 2^n possible floating point numbers in an n-bit field.
  - To store a real number, we must find a floating point number that is "close to" or "approximates" the real number.
  - Demos (C programs): *float-vs-double-approx, float-double-add-approx*

Data Representation. Reals. Floating Point

- Principle: let the point "float" and attach to each number an indication about the current position of the radix point (as in scientific notation)
- Example: an 8-bit format (not used in practice)

Sign 1bit	Exponent 3 bits	Mantissa 4 bits	Radix 2 number	Radix 10 number
0	001	1001	1.001×2^1	2.25
0	011	1110	1.11×2^3	7
	Cannot represent		1.11×2^7	224
0	101	1110	1.11×2^{-1}	0.875

Data Representation. Reals. Floating Point

- Exponent (3 bits) is biased by 3
- The 1 at the front of the mantissa is implicit
- Zero is represented as all zeros
- Example: an 8-bit format (modified)

Sign 1bit	**Exponent 3 bits**	**Mantissa 4 bits**	**Radix 2 number**	**Radix 10 number**
0	**100**	**0010**	$\mathbf{1.001 \times 2^{1}}$	**2.25**
0	**111**	**1100**	$\mathbf{1.11 \times 2^{4}}$	**28**
	Cannot represent		$\mathbf{1.11 \times 2^{7}}$	**224**
0	**010**	**1100**	$\mathbf{1.11 \times 2^{-1}}$	**0.875**

Data Representation. Reals. Floating Point

- During the 60s and 70s. IBM used an hexadecimal base for the exponent. Thus a value was:

$$X=(-1)^S \times m \times 16^e$$

where *m=mantissa. S=sign. e=exponent*

- 1985 (2008): IEEE 754 standard. **Three important requirements:**
  - **Consistent** representation of floating point numbers across machines
  - **Correct** arithmetic rounding
  - **Consistent** and **sensible** exception handling (e.g. divide by zero)

Data Representation. Reals. Floating Point IEEE 754

- Formats
  - 32 bits (simple precision)
    - sign (S): 1 bit
    - exponent: 8 bits
    - mantissa: 23 bits
    - Exponent bias: 127
  - 64 bits (double precision)
    - sign (S): 1 bit
    - exponent: 11 bits
    - mantissa: 52 bits
    - Exponent biased: 511
  - Extended

S	**Exponent (characteristic)**	**Mantissa (significand)**

Data Representation. Reals. Floating Point IEEE 754

- **Normalized numbers.** Nonzero numbers are normalized as:

  $$\boldsymbol{X = m \times 2^E}\text{. where } \mathbf{1} \leq \boldsymbol{m<2}\text{. i.e.}$$

  $$\boldsymbol{m = (b_0.b_1b_2b_3\ldots)_2}\text{. with } \boldsymbol{b_0 = 1}.$$

  Because we **know** that $b_0 = 1$. we do not need to store this bit
- Use the 23 bits of the significand to store $b_1b_{2,}b_3 \ldots b_{23}$ instead of $b_0b_1b_2b_3 \ldots b_{22}$ thus changing the **precision** from $\varepsilon = 2^{-22}$ to 2^{-23}
- The stored **bit string** represents the **fraction part** of the mantissa called a **fraction field**.

Data Representation. Reals. Floating Point IEEE 754

- Thus. given a bit string in the fraction field we must imagine that there is a leading **1** in front of the string. This is called a **hidden bit normalization**
- **Note: all zero digits** in the fraction field for a normalized number represent the value **1.0**. not **0.0.** Then:
- **Zero** must be a **special number**

IEEE 754

- **Special numbers:**
  - **Zero**
  - **Infinity ∞.** This enables **1.0/0.0** → ∞. instead of terminating the operation by an **overflow**
  - There is a distinct representation for −∞
- NaN (**Not a Number)**. is a bit pattern defined for **errors**.
- All special numbers. including **subnormal numbers**. are represented by specific bit patterns in the exponent field
- If the **exponent is zero**. but the **fraction is nonzero**. the represented number is **subnormal**.

IEEE 754 Single Precision. Example Values Representation

Type	sign (31)	exponent(8) (30–23)	fraction (23-bit) (22–0)	Hex	Watch in Windows®	Valule
Zero	0	00000000	00000000000000000000000	00000000	0.00000000	0
One	0	01111111	00000000000000000000000	3F800000	1.00000000	1
Minus One	1	01111111	00000000000000000000000	BF800000	-1.0000000	−1.0
Smallest denormalized number	*	00000000	00000000000000000000001	00000001	1.401e-045#DEN	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	00000000	10000000000000000000000	00400000	5.877e-039#DEN	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-12}7 \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	00000000	11111111111111111111111	007FFFFF	1.175e-038#DEN	$\pm (1 - 2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	00000001	00000000000000000000000	00800000	1.1754944e-038	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	11111110	11111111111111111111111	7F7FFFFF	3.4028235e+038	$\pm (2 - 2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	11111111	00000000000000000000000	7F800000	1.#INF000	$+\infty$
Negative infinity	1	11111111	00000000000000000000000	FF800000	-1.#INF000	$-\infty$
Not a number	*	11111111	non-zero	7F800001	1.#QNAN00	NaN

* Sign bit can be either 0 or 1 .

Figure: Floating-point Binary

Data Representation. Reals. Floating Point IEEE 754

- Examples (simple precision)

$$1 = (1.000\ldots0)_2 \times 2^0$$

0	01111111	00000000000000000000000

$$32. = (1.0)_2 \times 2^5$$

0	10000100	0000000000000000000000

$$(1.111\ldots1)_2 \times 2^{127} \text{ — } 3.4 \times 10^{38}$$

0	11111110	11111111111111111111111

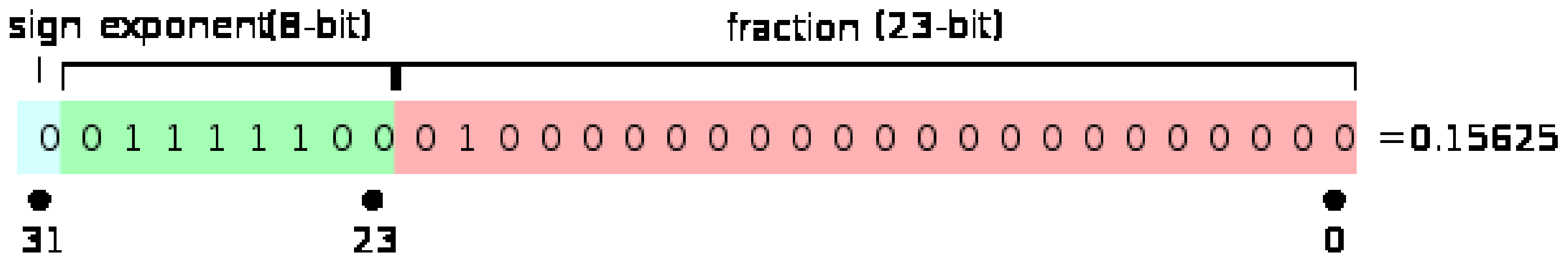
Data Representation. Reals. Floating Point IEEE 754

- Exception answers

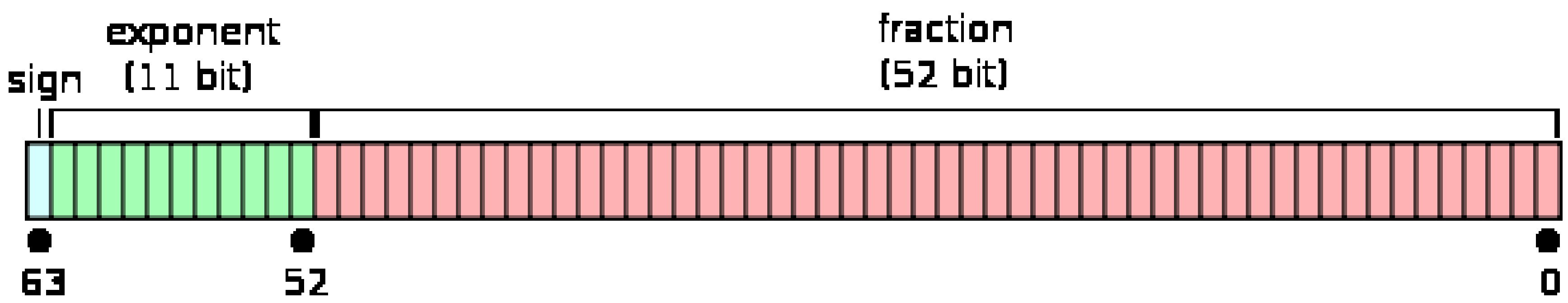
Invalid operation	NaN
Divide by 0	$\pm\infty$
Overflow	$\pm\infty$ or N_{max}
Underflow	0 or an adjacent subnormal number
Precision or inexact	The correct rounded value

- Exceptions are signaled
- Demos: IEEE-754.html

Data Representation. Reals. IEEE-754



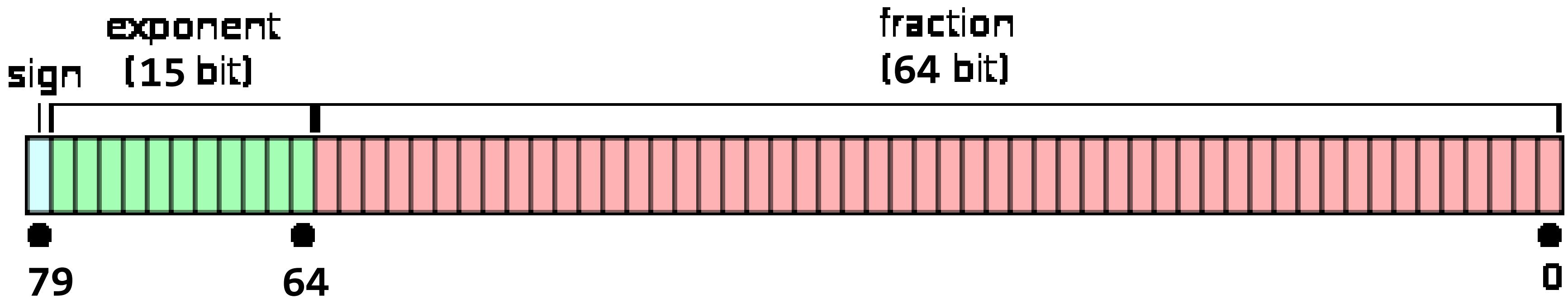
Single precision: 32 bits

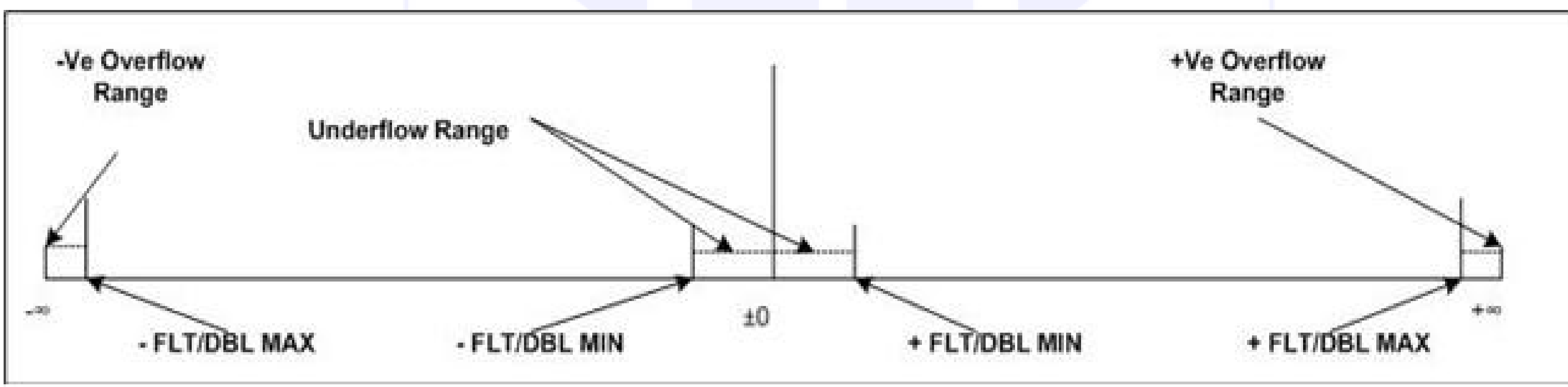


Double precision: 64 bits

http://en.wikipedia.org/wiki/IEEE_754-1985

Data Representation. Reals. IEEE-754





Data Representation. Reals. Floating Point

- **Shortcomings:**
  - **Representation error.** The *precision of a representation* is given by the *number of bits used* for the exponent and the significand.
    - number of bits is insufficient ⇒ represented number can be approximated by rounding or by truncation.
  - **Error propagation.** Arithmetic operations often multiply the effect of errors which exist in input data.
    - For instance. $2.51 \times 2.32 = 5.8232$.
    - BUT, using a single digit in the fraction part $2.5 \times 2.3 = 5.75$.
  - **Gaps/extremes** in the representation. The representation scheme is such a way that there is an option to represent zero and a *gap between zero and the smallest representable number*

C Primitive types

Type Name	Bytes	Other Names	Range of Values
char	1	signed char	–128 to 127
unsigned char	1		0 to 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
int	4	signed, signed int	–2,147,483,648 to 2,147,483,647
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	long long, signed long long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	unsigned long long	0 to 18,446,744,073,709,551,615
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	8	none	same as double
wchar_t	2	__wchar_t	0 to 65,535

Variable declaration

- For simple variables:

```
type identifier {, identifier };
{ type identifier {, identifier };}
```

  - Examples:

```
int i, j, k;
char c;
double x, y;
```

- For array variables:

```
base_type identifier[lim] {[lim] } {, identifier[lim]
  {[lim] }};
```

  - **Indices run from 0 to lim-1.**
  - **Limits are constant expressions, evaluated at compile time**
  - Examples:

```
int alpha[100];
double matrix[10][15];
```

Variable initialization

- Pattern: `type identifier = expression;`
  - **expression** is evaluated at compile time
  - **expression** value is converted to the type of the variable
- Example:

```
int n=10;
double x=20.5;
char alpha='a';

#define beta 10

int j=20+beta; /*evaluated at compile time */
int f(int m)
{
    int i=10; /* automatic variable */
    int j=i+20 /* automatic variable*/
    ...
}
```

Variable initialization. Arrays

- Arrays can be initialized, but it is not necessary to initialize all elements
- Initialization of a one-dimensional array

```
base_type identifier[lim]={v0, v1,..., vn};
```

where ***vi*** are expressions

- As a result:
  - `identifier[0]=v0, identifier[1]=v1, ...., identifier[n]=vn`
  - The other elements are initialized to zero if static or global, or have undefined values for automatic arrays

Variable initialization. Arrays

- Initialization of a two-dimensional array

```
base_type identifier[lim1][lim2]=
    {
        {v00, v01,..., v0n},
        {v10, v11 ,..., v1m},
        ...
        {vi0, vi1 ,..., vik}
    };
```

  where ***vij*** are expressions evaluated at compile time

- Examples:

```
int alpha[10]={0, 1, 2, 3, 4, 5};
int matrix[5][5]=
    {
        {1, 2, 3},
        {5, 6, 7, 8},
        {9, 10, 11},
        {7, 7, 7, 7},
    };
```

Expressions

- **Expression**=one or more *operands* joined by *operators*
- An **operand** can be:
  - A numeric or a character constant
  - A symbolic constant
  - A simple variable identifier
  - An array identifier
  - A structure identifier
  - A type identifier
  - A function identifier
  - An indexed variable
  - A component of a structure
  - A function call
  - An expression enclosed between parenthesis

Expressions

- An **operand**
  - Has a **type** and a **value**
- An **operator**
  - Can be **unary** or **binary**
    - Unary: applied to a single operand
    - Binary: applied to both preceding and following operand
  - Does not have any effect on character string constants

Expressions. Operators

- Operators are grouped in classes
  - Arithmetic operators: unary + – and binary + – * / %
  - Relation operators < <= > >=
  - Equality operators == !=
  - Logic operators: negation !, and &&, or ||
  - Bitwise operators: one's complement ~, left shift <<, right shift >>, and &, or |, exclusive or ^
  - Assignment operators: assignment =; compound assignment: **op=** where **op** is an arithmetic or bitwise operator. The effect is:

    $v \text{ op} = \textit{operand} \equiv v = v \text{ op } \textit{operand}$

Expressions. Operators

- Operators are grouped in classes (cont'd)
  - Increment/decrement operators: ++, – –
  - Type cast operators: (*type*) *operand*
  - Size operator: sizeof(*variable*), sizeof(*type*)
  - Referencing operator: &*identifier*
  - Parenthesis operator: () []
  - Conditional operator: ? : (*expression*? *exp*1: *exp*2)
  - Comma operator: ,
  - Dereferencing operators: *, ., ->
  - C++ specific:
    - Memory allocation/free: **new**/**delete**
    - Resolution: **::**

Expression evaluation

- What is taken into account:
  - Operator priority
  - Operator associativity for equal priority
  - Default conversion rule
  - Descending order of type priority:
    - long double
    - double
    - float
    - unsigned long
    - long
    - unsigned
    - int

Expression evaluation

- The result of evaluating relational expressions is true or false. **False** is represented by **zero**, **true** by **non-zero**
- Increment and decrement operators can be prefix (++*operand*) or postfix (operand--)
  - Prefix: in evaluation the value after increment/decrement is used, e.g.
    ```
    x=10;
    y=++x; /* y=11 and x=11 */
    ```
  - Postfix: the value before increment/decrement is used in evaluation
    ```
    x=10;
    z=x++; /* z=10 and x=11 */
    ```
- **Comma operator**. The expressions are evaluated strictly left to right and their values discarded, except for the last one, whose type and value determine the result of the overall expression. Usage examples (for decomposition of complex evaluations):
  ```
  y=((c=(a<0)? -a: a), (d=(b<0)? -b: b), (c>d)? c: d);
  alpha = ((a1=x+y+z), (a2=x+y+w), (a1*a2+10));
  ```

Logical operators. Truth tables

Truth table for the && (logical AND) operator.

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Truth table for operator ! (logical negation).

expression	! expression
0	1
nonzero	0

Truth table for the logical OR (||) operator.

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Operator precedence and associativity

Level	Operator(s)	Description	Associativity
17	::	global scope (unary)	right-to-left
17	::	class scope (binary)	left-to-right
16	-> .	member selectors	left-to-right
16	[ ]	array index	left-to-right
16	( )	function call	left-to-right
16	( )	type construction	left-to-right
16	sizeof	size in bytes	left-to-right
15	++ --	increment, decrement	right-to-left
15	~	bitwise NOT	right-to-left
15	!	logical NOT	right-to-left
15	+ -	unary plus, minus	right-to-left
15	* &	dereference, address-of	right-to-left
15	( )	cast	right-to-left
15	new delete	free store management	right-to-left
14	->* .*	member pointer selectors	left-to-right

Operator precedence and associativity

Level	Operator(s)	Description	Associativity
13	`* / %`	multiplicative operators	left-to-right
12	`+ -`	arithmetic operators	left-to-right
11	`<< >>`	bitwise shift	left-to-right
10	`< <= > >=`	relational operators	left-to-right
9	`== !=`	equality, inequality	left-to-right
8	`&`	bitwise AND	left-to-right
7	`^`	bitwise exclusive OR	left-to-right
6	`|`	bitwise inclusive OR	left-to-right
5	`&&`	logical AND	left-to-right
4	`||`	logical OR	left-to-right
3	`? :`	arithmetic if	left-to-right
2	`= *= /* %= += -=` `<<= >>= &= |= ^=`	assignment operators	right-to-left
1	`,`	comma operator	left-to-right

Reading

- King, Chapters 4 & 7
- Prata, Chapter 5
- Deitel, 2.5, 2.6, 3.11, 3.12, 4.10, 4.11
  (see Lecture 1, slide 14 for full book names)

Summary

- Programming style
  - Definition
  - Advice
- Digital Representations
  - Signed integers – signed magnitude; two's complement
  - Reals – floating point
- Variables
  - Declaration
  - Initialization
- Expressions
  - Definition
  - Evaluation