Programming and Data Structures in C

Grzegorz Jabłoński

Department of Microelectronics and Computer Science

tel. (631) 26-48

gwj@dmcs.p.lodz.pl

http://neo.dmcs.p.lodz.pl/pdsc

C Timeline

- 1969 - Ken Thompson creates Unix, B from BCPL
- 1970 - Thompson & Ritchie evolve B to C
- 1978 - K&R's "The C Programming Language"
- 1982 - ANSI forms a committee on standardizing C
- 1988 - K&R updated for ANSI Draft
- 1989 - ANSI approves C (called "C89")
- 1990 - ISO approves C (called "C90")
- 1995 - New committee formed for "C9X"
- 1999 - ISO approval (called "C99")
- 2000 - ANSI approves "C99"

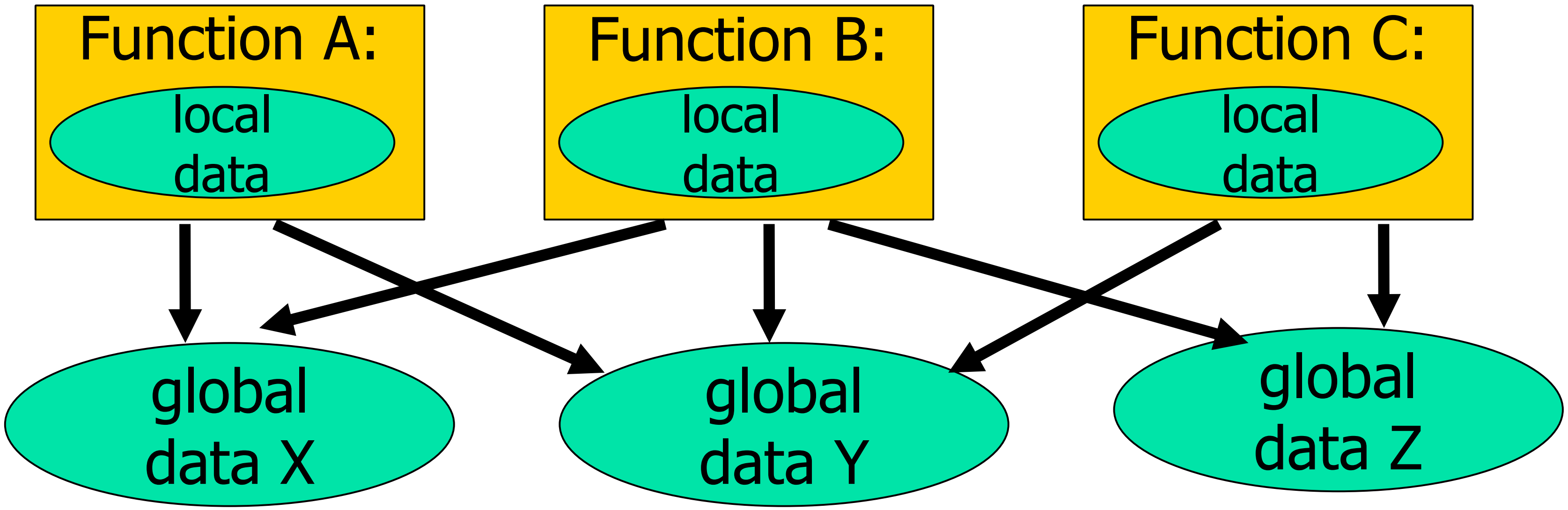
C Timeline

Structural Programming

- C, Pascal, Fortran are procedural programming languages.
- A program in a procedural language is a list of instructions, augmented with loops and branches.
- For small programs no other organizational principle (paradigm) is needed.
- Larger programs are broken down into smaller units.
- A procedural program is divided into functions, such that ideally each has clearly defined purpose and interface to other functions.
- The idea of breaking a program into functions can be further extended by grouping functions that perform similar tasks into modules.
- Dividing a program into functions and modules is the key idea of structured programming.

Problems with Structured Programming

- Functions have unrestricted access to global data



- Large number of potential connections between functions and data (everything is related to everything, no clear boundaries)
  - makes it difficult to conceptualize program structure
  - makes it difficult to modify and maintain the program
  - e.g.: it is difficult to tell which functions access the data

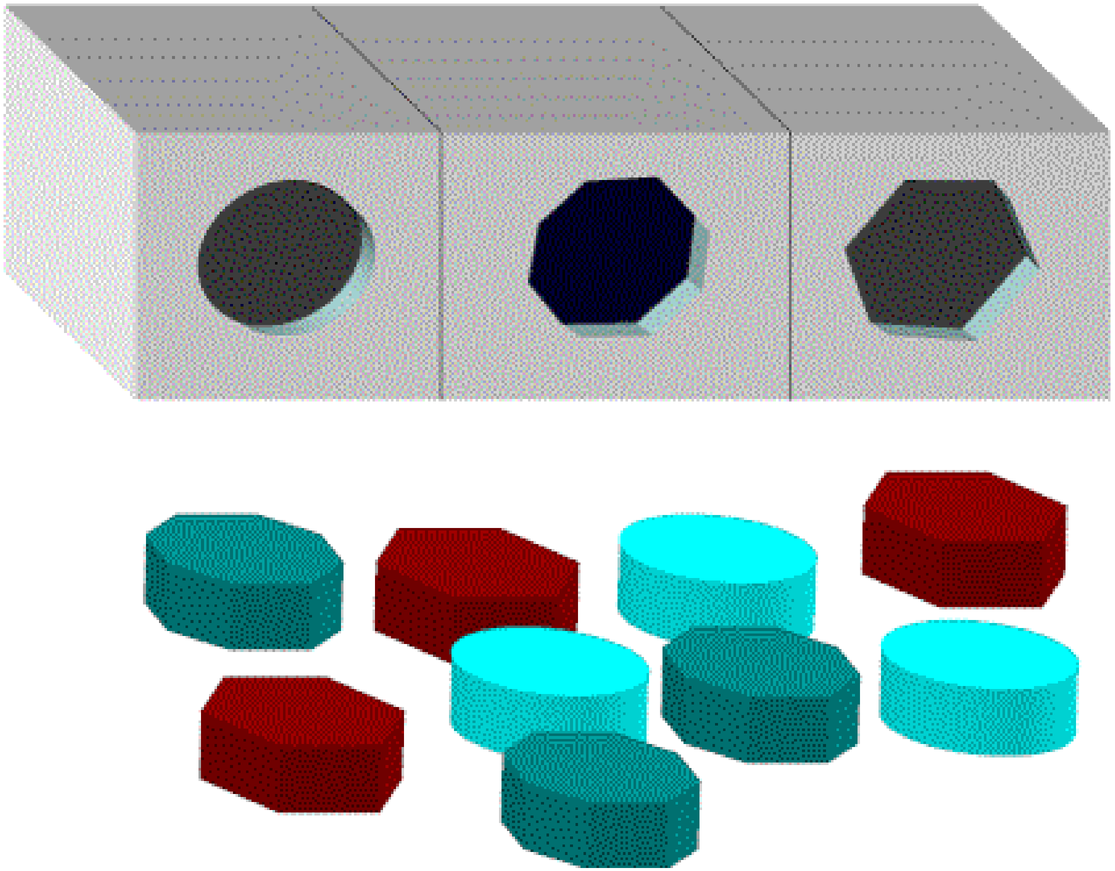
From Data to Data Structures

machine level data storage	`0100111001001011010001`
primitive data types	`28` `3.1415` 'A'
data aggregates	array structure
high-level data structures	stack queue tree

On each level...

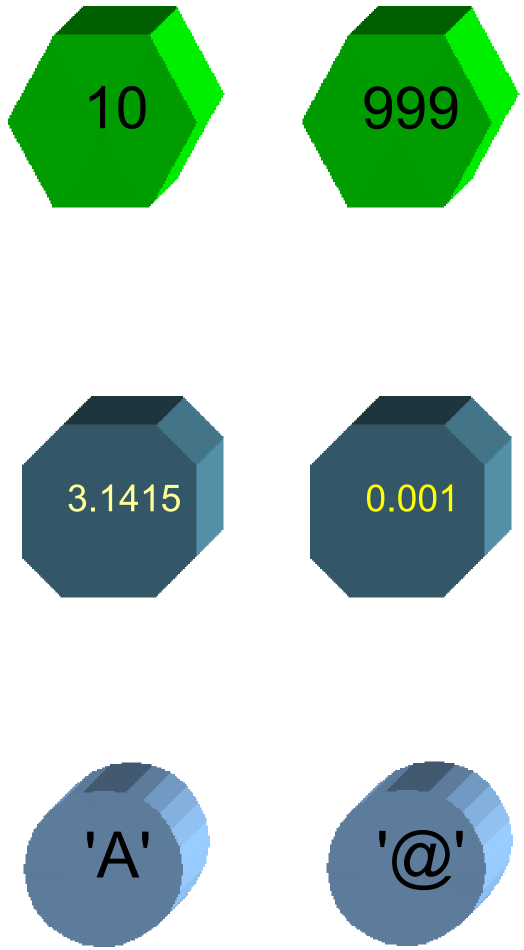
- We do not want to be concerned with the way to represent objects of this level via objects of lower level
- We want to be concerned with the semantics of data on this level.
- What is it ?
- What we can do with it ?

Primitive data types



Primitive Data Types

- Integer data
  - 1, 10, 999, 1000
- Floating point data
  - 2.7128, 0.003, 7.0
- Characters
  - 'A', 'B', '_', '@'



Representation of Integers – Positional Notation

- Number base B $\Rightarrow$ B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 (Binary): 0, 1
- Number representation:
  - $d_{31}d_{30} \dots d_1 d_0$ is a 32 digit number
  - value = $d_{31} \times B^{31} + d_{30} \times B^{30} + \dots + d_1 \times B^1 + d_0 \times B^0$
- Binary: 0,1 (In binary digits called "bits")
  - 0b11010 $= 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$
    $= 16 + 8 + 2$
    $= 26$

#s often written 0b…
non standard extension

  - Here 5 digit binary # turns into a 2 digit decimal #
  - Can we find a base that converts to binary easily?

Hexadecimal Numbers – Base 16

- Hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Normal digits + 6 more from the alphabet
  - In C, written as 0x… (e.g., 0xFAB5)
- Conversion: Binary⇔Hex
  - 1 hex digit represents 16 decimal values
  - 4 binary digits represent 16 decimal values
  - 1 hex digit replaces 4 binary digits
- One hex digit is a “nibble”. Two is a “byte”
- Example:
  - 1010 1100 0011 (binary) = 0x______ ?

Decimal vs. Hexadecimal vs. Binary

- Examples:
  - 1010 1100 0011 (binary) = 0xAC3
  - 10111 (binary) = 0001 0111 (binary) = 0x17
  - 0x3F9 = 11 1111 1001 (binary)
- How do we convert between hex and decimal?

MEMORIZE!

00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

How to Represent Negative Numbers?

- Obvious solution: define leftmost bit to be sign!
  - $0 \Rightarrow +$, $1 \Rightarrow -$
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude
- MIPS uses 32-bit integers. $+1_{ten}$ would be:

  0000 0000 0000 0000 0000 0000 0000 0001
- And -1_{ten} in sign and magnitude would be:

  1000 0000 0000 0000 0000 0000 0000 0001

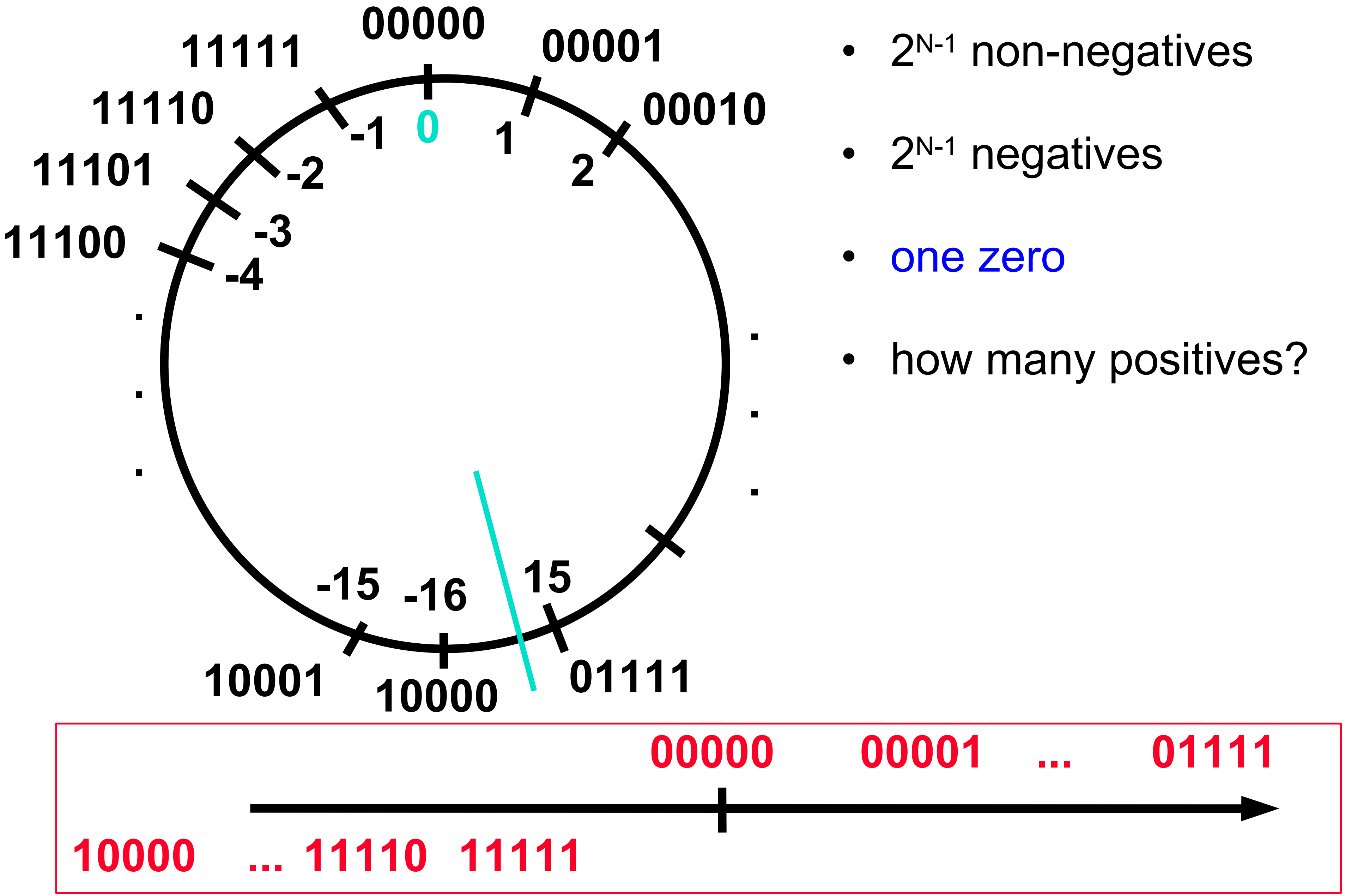
Shortcomings of Sign and Magnitude?

- Arithmetic circuit complicated
  - Special steps depending whether signs are the same or not
- Also, two zeros
  - 0x00000000 = $+0_{ten}$
  - 0x80000000 = -0_{ten}
  - What would two 0s mean for programming?
- Therefore sign and magnitude abandoned

Standard Negative Number Representation

- What is the result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
    - 3 - 4 $\Rightarrow$ 00…0011 - 00…0100 = 11…1111
  - With no obvious better alternative, pick representation that made the hardware simple
  - As with sign and magnitude,
    - leading 0s $\Rightarrow$ positive,
    - leading 1s $\Rightarrow$ negative
    - 000000...xxx is ≥ 0, 111111...xxx is < 0
    - except 1…1111 is -1, not -0 (as in sign & mag.)
- This representation is Two's Complement

2's Complement Number "line": N = 5



- 2^{N-1} non-negatives
- 2^{N-1} negatives
- one zero
- how many positives?

Two's Complement for N=32

$0000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0000_{two} = 0_{ten}$
$0000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0001_{two} = 1_{ten}$
$0000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0010_{two} = 2_{ten}$
. . .
$0111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ ...\ 0000\ \ 0000\ \ 0000\ \ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . .
$1111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1101_{two} = -3_{ten}$
$1111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1110_{two} = -2_{ten}$
$1111\ ...\ 1111\ \ 1111\ \ 1111\ \ 1111_{two} = -1_{ten}$

- One zero; 1st bit called sign bit
- 1 "extra" negative: no positive $2{,}147{,}483{,}648_{ten}$

Two's Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:
  - $d_{31} \times -(2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$
- Example: 1101_{two}

  $= 1\text{x}-(2^3) + 1\text{x}2^2 + 0\text{x}2^1 + 1\text{x}2^0$

  $= -2^3 + 2^2 + 0 + 2^0$

  $= -8 + 4 + 0 + 1$

  $= -8 + 5$

  $= -3_{ten}$

Two's Complement Shortcut: Negation

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result
- Proof: Sum of number and its (one's) complement must be $111...111_{two}$
  - However, $111...111_{two} = -1_{ten}$
  - Let x' $\Rightarrow$ one's complement representation of x
  - Then $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$
- Example: $-3 \Rightarrow +3 \Rightarrow -3$

  x : $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}$

  x': $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

  +1: $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_{two}$

  ()': $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$

  +1: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two}$

Two's Comp. Shortcut: Sign extension

- Convert 2's complement number rep. using n bits to more than n bits
- Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2's comp. positive number has infinite 0s
  - 2's comp. negative number has infinite 1s
  - Binary representation hides leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit:

$$1111\ 1111\ 1111\ 1100_{two}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$$

Two's Comp. Shortcut: Multiplication and Division by 2

- Multiplication by 2 is just a left shift (unless an overflow occurs)

  $(-5_{ten}) * 2_{ten} = -10_{ten}$

  $1111\ 1111\ 1111\ 1011_{two} * 2_{ten} = 1111\ 1111\ 1111\ 0110_{two}$

  $5_{ten} * 2_{ten} = 10_{ten}$

  $0000\ 0000\ 0000\ 0101_{two} * 2_{ten} = 0000\ 0000\ 0000\ 1010_{two}$

- Division by 2 requires shift-in of a copy of the most significant bit

  $(-4_{ten}) / 2_{ten} = -2_{ten}$

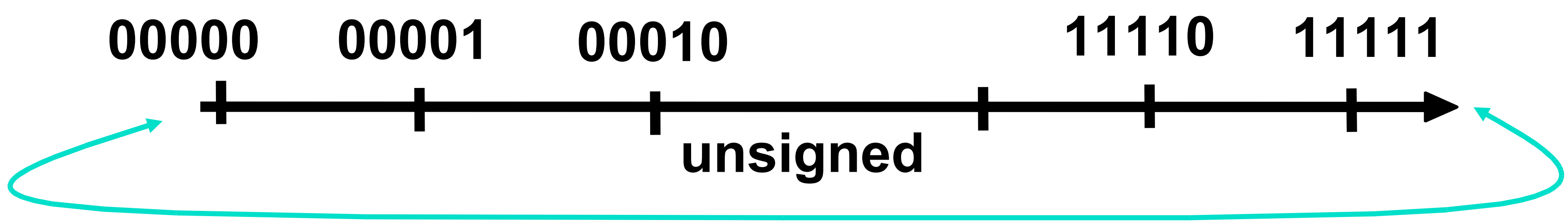
  $1111\ 1111\ 1111\ 1100_{two} / 2_{ten} = 1111\ 1111\ 1111\ 1110_{two}$

  $(4_{ten}) / 2_{ten} = 2_{ten}$

  $0000\ 0000\ 0000\ 0100_{two} / 2_{ten} = 0000\ 0000\ 0000\ 0010_{two}$

What If Too Big?

- Binary bit patterns above are simply representatives of numbers. Strictly speaking they are called "numerals".
- Numbers really have an ∞ number of digits
  - with almost all being same (00…0 or 11…1) except for a few of the rightmost digits
  - Just don't normally show leading digits
- If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.



C Integer Types

- **`signed`** and **`unsigned`**
  - treated as values with or without sign
  - signed usually stored in 2's complement format
  - same amount of bits, different range
- exact size and range of integer types is not defined in the standard, it is implementation-defined
  - number of bytes occupied by a variable of a given type can be determined using the **`sizeof()`** operator
  - range of values of a given type can be determined using macros in **`limits.h`**

`char` Type

- not defined, whether it is signed or unsigned
- must store every character from the character set
- can be qualified with the keyword signed or unsigned
- by definition, **`sizeof(char) == 1`**
- at least 8 bits wide

```
char c1; /* signed or unsigned */
unsigned char c2;
signed char c3;

printf("%d\n", sizeof(c1)); /* prints 1 */
printf("%d\n", sizeof(char)); /* also prints 1 */
```

char Type – Macros in <limits.h>

- **CHAR_BIT**
  - The macro yields the maximum value for the number of bits used to represent an object of type char
- **CHAR_MAX**
  - The macro yields the maximum value for type char. Its value is:
    - **SCHAR_MAX** if char represents negative values
    - **UCHAR_MAX** otherwise
- **CHAR_MIN**
  - The macro yields the minimum value for type char. Its value is:
    - **SCHAR_MIN** if char represents negative values
    - zero otherwise
- **SCHAR_MAX**
  - The macro yields the maximum value for type signed char
- **SCHAR_MIN**
  - The macro yields the minimum value for type signed char
- **UCHAR_MAX**
  - The macro yields the maximum value for type unsigned char

Character sets

- ASCII
  - Formula for representing English characters as numbers, with each letter assigned a number from 0 to 127; not all of those are really printable characters. An acronym for American Standard Code for Information Interchange.
  - ASCII control characters are presented in the table at the right
- EBCDIC
  - Extended Binary Coded Decimal Interchange Code
  - IBM's 8-bit extension of the 4-bit Binary Coded Decimal encoding of digits 0-9 (0000-1001).

Char	Dec	Control-Key	Control Action
NUL	0	^@	NULl character
SOH	1	^A	Start Of Heading
STX	2	^B	Start of TeXt
ETX		^!	End of TeXt
EOT	"	^#	End Of Tran$%i$$ion
EN&	'	^E	EN&(ir)
A!*	+	^,	A!*no-ledge
BEL	.	^/	BEL0 ring$ ter%inal 1ell
BS	2	^H	Bac3S4ace 5non6de$tr(cti7e8
HT	9	^:	Hori;ontal Ta1 5%o7e to ne<t ta1 4o$ition8
L,	10	^=	Line ,eed
>T	11	^*	>ertical Ta1
,,	12	^L	,or% ,eed
!?	1	^@	!arriage ?et(rn
SO	1"	^N	Shift O(t
S:	1'	^O	Shift :n
#LE	1+	^A	#ata Lin3 E$ca4e
#!1	1.	^&	#e7ice !ontrol 10 nor%all) XON
#!2	12	^?	#e7ice !ontrol 2
#!	19	^S	#e7ice !ontrol 0 nor%all) XO,,
#!"	20	^T	#e7ice !ontrol "
NA*	21	^U	Negati7e Ac*no-ledge
SBN	22	^>	SBNchrono($ idle
ETB	2	^C	End Tran$%i$$ion Bloc3
!AN	2"	^X	!ANcel line
E@	2'	^B	End of @edi(%
SUB	2+	^D	SUB$tit(te
ES!	2.	^E	ES!a4e
,S	22	^F	,ile Se4arator
/S	29	^G	/ro(4 Se4arator
?S	0	^^	?ecord Se4arator
US	1	^H	Unit Se4arator

ASCII Printing characters

Dec	Description
2	S4ace
	E<cla%ation %ar3
"	&(otation %ar3
'	! ro$$ hatch 5n(%1er $ign8
+	#ollar $ign
.	Aercent $ign
2	A%4er$and
9	! lo$ing $ingle J(ote 5a4o$tro4he8
"0	O4ening 4arenthe$e$
"1	! lo$ing 4arenthe$e$
"2	A$teri$3 5$tar0 %(lti4l)8
"	Al($
""	! o%%a
"'	H)4hen0 da$h0 %in($
"+	Aeriod
".	Sla$h 5for-ard or di7ide8
"2	Dero
"9	One
'0	T-o
'1	Three
'2	,o(r
'	,i7e
'"	Si<
''	Se7en
'+	Eight
'.	Nine
'2	! olon
'9	Se%icolon
+0	Le$$ than $ign
+1	EJ(al$ $ign
+2	/ reater than $ign
+	&(e$tion %ar3

Char	Dec	Description
@	+"	At6$ign
A	+'	U44er ca$e A
B	++	U44er ca$e B
!	+.	U44er ca$e !
#	+2	U44er ca$e #
E	+9	U44er ca$e E
,	.0	U44er ca$e ,
/	.1	U44er ca$e /
H	.2	U44er ca$e H
:	.	U44er ca$e :
=	."	U44er ca$e =
*	.'	U44er ca$e *
L	.+	U44er ca$e L
@	..	U44er ca$e @
N	.2	U44er ca$e N
O	.9	U44er ca$e O
A	20	U44er ca$e A
&	21	U44er ca$e &
?	22	U44er ca$e ?
S	2	U44er ca$e S
T	2"	U44er ca$e T
U	2'	U44er ca$e U
>	2+	U44er ca$e >
C	2.	U44er ca$e C
X	22	U44er ca$e X
B	29	U44er ca$e B
D	90	U44er ca$e D
E	91	O4ening $J(are 1rac3et
F	92	Bac3lah 5?e7er$e $lant8
G	9	! lo$ing $J(are 1rac3et
^	9"	! aret 5! irc(%fle<8
H	9'	Under$core

Char	Dec	Description
I	9+	O4ening $ingle J(ote
a	9.	Lo-er ca$e a
1	92	Lo-er ca$e 1
c	99	Lo-er ca$e c
d	100	Lo-er ca$e d
e	101	Lo-er ca$e e
f	102	Lo-er ca$e f
g	10	Lo-er ca$e g
h	10"	Lo-er ca$e h
i	10'	Lo-er ca$e i
K	10+	Lo-er ca$e K
3	10.	Lo-er ca$e 3
l	102	Lo-er ca$e l
%	109	Lo-er ca$e %
n	110	Lo-er ca$e n
o	111	Lo-er ca$e o
4	112	Lo-er ca$e 4
J	11	Lo-er ca$e J
r	11"	Lo-er ca$e r
$	11'	Lo-er ca$e $
t	11+	Lo-er ca$e t
(	11.	Lo-er ca$e (
7	112	Lo-er ca$e 7
-	119	Lo-er ca$e -
<	120	Lo-er ca$e <
)	121	Lo-er ca$e)
;	122	Lo-er ca$e ;
L	12	O4ening c(rl) 1race
M	12"	>ertical line
N	12'	! lo$ing c(rl) 1race
O	12+	Tilde 5a44ro<i%ate8
#EL	12.	#elete 5r(1o(t80 cro$$6hatch 1o<

EBCDIC Character Set

Dec	EBCDIC	
0	NUL	N(ll
1	SOH	Start of Heading
2	STX	Start of Te<t
	ETX	End of Te<t
"	A,	A(nch Off
'	HT	Hori;ontal Ta1
+	L!	Lo-er !a$e
.	#EL	#elete
10	S@@	Start of @an(al @e$$age
11	>T	>ertical Ta1
12	,,	,or% ,eed
1	!?	!arriage ?et(rn
1"	SO	Shift O(t
1'	S:	Shift :n
1+	#LE	#ata Lin3 E$ca4e
1.	#!1	#e7ice !ontrol 1
12	#!2	#e7ice !ontrol 2
19	T@	Ta4e @ar3
20	?ES	?e$tore
21	NL	Ne- Line
22	BS	Bac3$4ace
2	:L	:dle
2"	!AN	!ancel
2'	E@	End of @edi(%
2+	!!	!(r$or !ontrol
2.	!U1	!($to%er U$e 1
22	:,S	:nterchange ,ile Se4arator
29	:/S	:nterchange /ro(4 Se4arator
0	:?S	:nterchange ?ecord Se4arator
1	:US	:nterchange Unit Se4arator
2	#S	#igit Select
	SOS	Start of Significance
"	,S	,ield Se4arator
+	BBA	B)4a$$
.	L,	Line ,eed

Dec	EBCDIC	
2	ETB	End of Tran$%i$$ion Bloc3
9	ES!	E$ca4e
"2	S@	Set @ode
"	!U2	!($to%er U$e 2
"'	EN&	EnJ(ir)
"+	A!*	Ac3no-ledge
".	BEL	Bell
'0	SBN	S)nchrono($:dle
'2	AN	A(nch On
'	?S	?eader Sto4
'"	U!	U44er !a$e
''	EOT	End of Tran$%i$$ion
'9	!U	!($to%er U$e
+0	#!"	#e7ice !ontrol "
+1	NA*	Negati7e Ac3no-ledge
+	SUB	S(1$tit(te
+"	SA	S4ace
."	W	!ent Sign
.'	X	Aeriod0 #eci%al Aoint0 TdotT
.+	Y	Le$$6than Sign
..	5	Left Aarenthe$i$
.2	Z	Al($ Sign
.9	M	Logical O?
20	[	A%4er$and
90	\	E<cla%ation Aoint
91	]	#ollar Sign
92	^	A$teri$30 T$tarT
9	8	?ight Aarenthe$i$
9"	_	Se%icolon
9'	`	Logical NOT
9+	6	H)4hen0 @in($ Sign
9.	a	Sla$h0 >irg(le
10.	0	!o%%a
102	b	Aercent
109	H	Underline0 Under$core

Dec	EBCDIC	
110	P	/reater6than Sign
111	Q	&(e$tion @ar3
122	R	!olon
12	S	N(%1er Sign0 Octothor40 T4o(ndT
12"	@	At Sign
12'	U	A4o$tro4he0 Ari%e
12+	V	EJ(al Sign
12.	T	&(otation @ar3
129	a	a
1 0	1	1
1 1	c	c
1 2	d	d
1	e	e
1 "	f	f
1 '	g	g
1 +	h	h
1 .	i	i
1"'	K	K
1"+	3	3
1".	l	l
1"2	%	%
1"9	n	n
1'0	o	o
1'1	4	4
1'2	J	J
1'	r	r
1+2	$	$
1+	t	t
1+"	(	(
1+'	7	7
1++	-	-
1+.	<	<
1+2	)	)
1+9	;	;
12'	I	/ra7e Accent

Dec	EBCI
19	A
19"	B
19'	!
19+	#
19.	E
192	,
199	/
200	H
201	:
209	=
210	*
211	L
212	@
21	N
21"	O
21'	A
21+	&
21.	?
22+	S
22.	T
222	U
229	>
2 0	C
2 1	X
2 2	B
2	D
2"0	0
2"1	1
2"2	2
2"	
2""	"
2"'	'
2"+	+
2".	.
2"2	2
2"9	9

`int` Type

- signed type
- basic integer type, represents natural integer type for the machine
- at least 16 bits wide
- can be qualified with the keyword signed or unsigned

```
int i1; /* signed */
unsigned int i2;
signed int i3;

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

`long int` Type

- signed type
- at least 32 bits, no shorter than **`int`**
- can be qualified with the keyword signed or unsigned
- int keyword can be omitted in declarations

```
long int i1; /* signed */
unsigned long int i2;
signed long int i3;
long i4; /* same type as i1 */
unsigned long i5; /* same type as i2 */
signed long i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

short int Type

- signed type
- at least 16 bits, no longer than int
- can be qualified with the keyword signed or unsigned
- int keyword can be omitted in declarations

```
short int i1; /* signed */
unsigned short int i2;
signed short int i3;
short i4; /* same type as i1 */
unsigned short i5; /* same type as i2 */
signed short i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

`long long int` Type

- C99 addition
- signed type
- at least 64 bits, no shorter than long
- can be qualified with the keyword signed or unsigned
- int keyword can be omitted in declarations

```
long long int i1; /* signed */
unsigned long long int i2;
signed long long int i3;
long long i4; /* same type as i1 */
unsigned long long i5; /* same type as i2 */
signed long long i6; /* same type as i3 */

printf("%d\n", sizeof(i1));
/* result is implementation defined */
```

Integer Types – Macros in <limits.h>

- **`INT_MAX`**
  - The macro yields the maximum value for type **`int`**
- **`INT_MIN`**
  - The macro yields the minimum value for type **`int`**
- **`UINT_MAX`**
  - The macro yields the maximum value for type **`unsigned int`**
- **`LONG_MAX, LONG_MIN, ULONG_MAX`**
  - The same for type **`long`**
- **`SHRT_MAX, SHRT_MIN, USHRT_MAX`**
  - The same for type **`short`**
- **`LLONG_MAX, LLONG_MIN, ULLONG_MAX`**
  - The same for type **`long long`**

Integer Constants (Literals)

- Decimal notation:
  - `int`: `1234`
  - `long int`: `1234L, 1234l`
  - `unsigned int`: `1234U, 1234u`
  - `unsigned long int`: `1234UL, 1234ul, 1234Ul, 1234uL`
  - `long long int`: `1234LL, 1234ll`
  - `unsigned long long`: `1234ULL, 1234ull, 1234uLL, 1234Ull`
- Octal notation:
  - starts with `0` (zero)
  - `031 == 25`
    - (31 Oct == 25 Dec, easy to confuse Christmas with Halloween)
  - the same suffixes as above applicable
- Hexadecimal notation:
  - starts with `0x` (zero x)
  - `0x31 == 49`
  - the same suffixes as above applicable

Character Constants (Literals)

- Direct notation:
  - `'a', 'b', ..., 'z', '0', ..., '9'`
- Special characters:
  - `'\n'` - newline
  - `'\r'` - carriage return
  - `'\a'` - visible alert
  - `'\b'` - backspace
  - `'\f'` - form feed
  - `'\t'` - horizontal tabulation
  - `'\v'` - vertical tabulation
  - `'\''` - single quote
  - `'\"'` - double quote
  - `'\?'` - question mark
  - `'\\'` - backslash
- Octal notation:
  - `'\077'`
  - `'\0'` (called **`NUL`** – note single 'l')
- Hexadecimal notation:
  - `'\x32'`

Floating Point

- Floating point is used to represent “real” numbers
  - 1.23233, 0.0003002, 3323443898.3325358903
  - Real means “not imaginary”
- Computer floating-point numbers are a subset of real numbers
  - Limit on the largest/smallest number represented
    - Depends on number of bits used
  - Limit on the precision
    - 12345678901234567890 --> 12345678900000000000
    - Floating point numbers are approximate, while integers are exact representation

Scientific Notation

$+ \; 34.383 \times 10^{2} = 3438.3$

Sign

Mantissa

Exponent

$+ \; 3.4383 \times 10^{3} = 3438.3$

Normalized form: Only one digit before the decimal point

+3.4383000E+03 = 3438.3

Floating point notation

8 digit mantissa can only represent 8 significant digits

Binary Floating Point Numbers

$+\ 101.1101$

$= 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$
$= 4 + 0 + 1 + 1/2 + 1/4 + 0 + 1/16$
$= 5.8125$

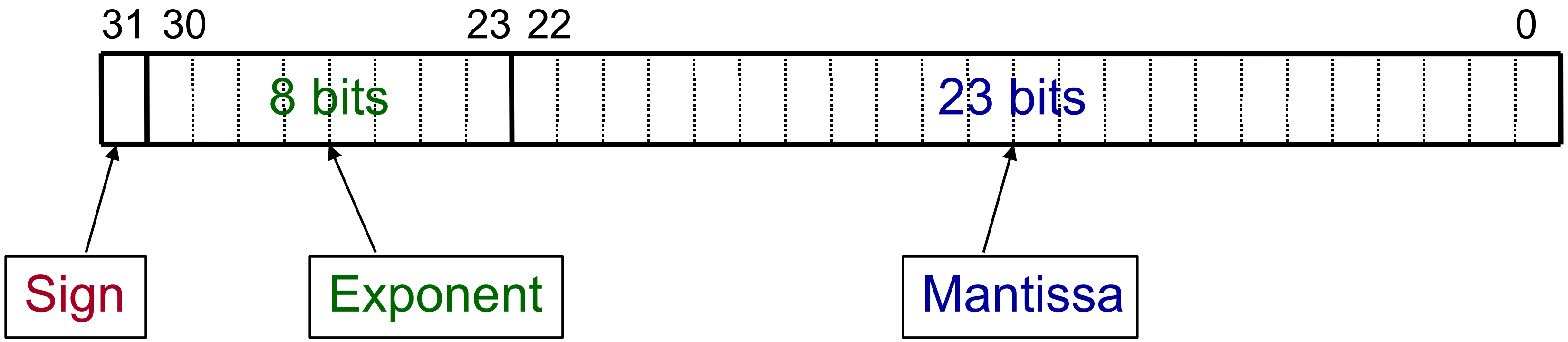
+1.011101 E+2

Normalized so that the binary point immediately follows the leading digit

Note: First digit is always non-zero
--> First digit is always one.

IEEE Floating Point Format

- The Institute of Electrical and Electronics Engineers
- Pronounce I-triple-E
- Is best known for developing standards for the computer and electronics industry



0: Positive

1: Negative

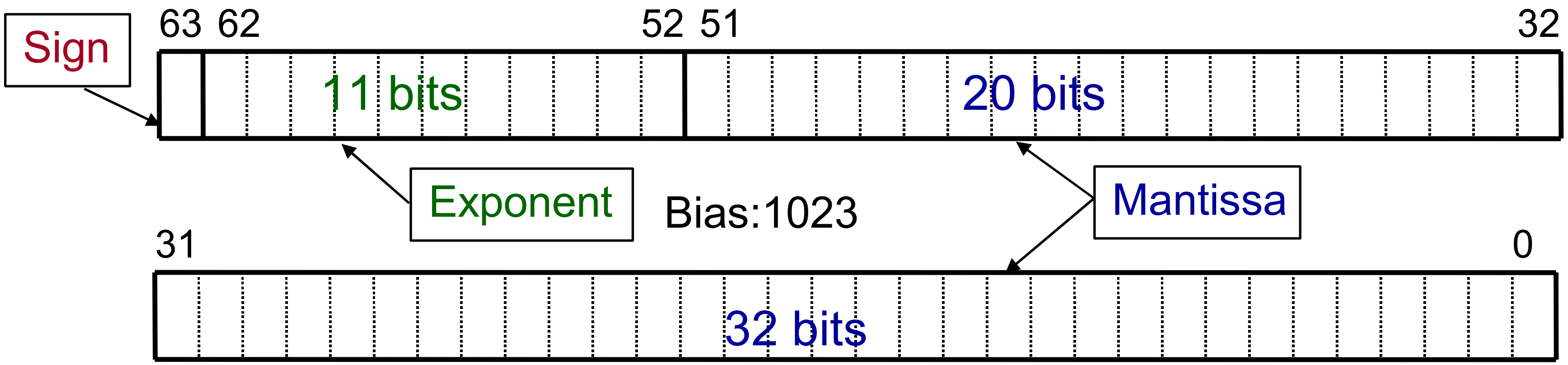
Biased by 127.

Leading '1' is implied, but not represented

$$\text{Number} = -1^{S} * (1 + M) \times 2^{E-127}$$

- Allows representation of numbers in range 2^{-127} to 2^{+128} ($10^{\pm 38}$)
- Since the mantissa always starts with '1', we don't have to represent it explicitly
  - Mantissa is effectively 24 bits

IEEE Double Precision Format



$$\text{Number} = -1^{S} * (1 + M) \times 2^{E-1023}$$

- Allows representation of numbers in range 2^{-1023} to 2^{+1024} ($10^{\pm 308}$)
- Larger mantissa means more precision

IEEE Extended Precision

- Optional recommendations for precision greater than float/double
    - Implemented in hardware (Intel 80-bit)
  - Single precision
    - Must support at least p = 32
    - At least 11 bits for exponent n–□
  - Double precision
    - p >= 64 &fiStts
    - Exponent range >= 15 bits
  - We won't say m t

Floating Point Data Types in C

- Three floating point types in C
  - **`float`**
  - **`double`**
  - **`long double`**
- Most frequently stored using IEEE standard
  - not necessarily, can even use base different than 2
  - floating point characteristics defined in **`<float.h>`**
- Three additional complex types in C99
- Constants:
  - **`1234.3`** – constant of type **`double`**
  - **`12345.5e7`** – constant of type **`double`**
  - **`123.4f`** – constant of type **`float`**
  - **`123.4F`** – constant of type **`float`**
  - **`123.4l`** – constant of type **`long double`**
  - **`123.4L`** – constant of type **`long double`**

Problems with Floating Point Numbers

- Many numbers cannot be represented exactly
  - The representation of 1/3 is 0.3333
    - 3 * “1/3” $\neq$ 1
  - The same problem with 1/10 in binary
- Results from floating-point calculations are almost never exactly equal to the corresponding mathematical value
- Results from a particular calculation may vary slightly from one computer system to another, and all may be valid. However, when the computer systems conform to the same standard, the amount of variation is drastically reduced.
- Results may vary with the optimization level
  - Values can be stored with greater precision in processor registers, than in memory

Problems with Floating Point Numbers

```
int main()
{
  float a = 2.501f;
  a *= 1.5134f;
  if (a == 3.7850134)
          printf("Expected value\n");
  else
          printf("Unexpected value\n");
  return 0;
}
```

- Never compare floating point numbers for equality
  - do not use `if (a == b) ...`
  - use `if( fabs(a - b) < error) ...` instead

Identifiers

- Names of things (variables, functions, etc.)
  - `int nMyPresentIncome = 0;`
  - `int DownloadOrBuyCD();`
- Up to 31 chars (letters, numbers, including _)
- Must begin with a letter
- Case sensitive! (“Url” is different from “URL”)

Naming Styles

- Styles:
  - lower_case
  - CAPITAL_CASE
  - camelCase
  - PascalCase (aka TitleCase)
  - szHungarianNotation
- Hungarian Notation:
  - Invented by Charles Simonyi, a Hungarian, born in Budapest in 1948

Implicit Type Conversion

- Implicit
  - `char b = ’9’;   /* Converts '9' to 57 */`
  - `int   a = 1;`
  - `int   s = a + b;`
- Integer promotion before operation: **`char/short`** ⇒ **`int`**
- When calling undeclared function, also floating point promotion: **`float`** ⇒ **`double`**
- If one operand is **`double`**, the other is made **`double`**
- else if either is **`float`**, the other is made **`float`**

```
int a = 3;
float x = 97.6F;
double y = 145.987;
y = x * y;
x = x + a;
```

Explicit Type Conversion

- Explicit (type casting)
- Sometimes you need to change the default conversion behavior

```
float x = 97.6;
x = (int)x + 1;
```

- Sometimes you need to help the compiler

```
float x = 97.6f;
printf("%d\n", x);        ⇒ 1610612736
printf("%d\n", (int) x);⇒ 97
```

- Almost any conversion does something – but not necessarily what you intended!!

Bad Type Conversion

- Example:

```
int x = 35000;
short s = x;
printf("%d %d\n", x, s);
```

- Output is:

```
35000 -30536
```

Constants

- Every variable can be qualified with the const modifier
  - `const int base = 345;`
- This variable now becomes a constant
- Constant must be assigned a value at a point where it is declared
- Trying to modify a constant will trigger a compile time error

Boolean Values in C

- C89 doesn't have booleans
- C99 defines a `_Bool` type
- Emulate as int or char, with values 0 (false) and 1 or non-zero (true)
- Allowed by control flow statements:

```
if ( success == 0 ) {
    printf( "something wrong" );
}
```

- You can define your own boolean:

```
#define FALSE 0
#define TRUE  1
```

Boolean Values in C

- This works in general, but beware:

```
if ( success == TRUE ) {
    printf( "everything is a-okay" );
}
```

- If success is greater than zero, it will be non-zero, but may not be 1; so the above is NOT the same as:

```
if ( success ) {
  printf( "Something is rotten in the state of "
          "Denmark" );
}
```

Enumeration

- Enums allow you to group logically related constants
  - `enum color {BLACK, RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW, WHITE, COLOR_MAX};`
- Here's another way to mock-up a Boolean

```
enum boolean { FALSE, TRUE };
enum boolean eAnswer = TRUE;
```

- Enum constants are treated as integer type

Enumeration

- Starts with 0 unless you specify value to start from
  - **`enum boolean { FALSE, TRUE };`**
  - **`enum genre { TECHNO, TRANCE=4, HOUSE };`**
- You can also specify values
  - **`enum channel { TVP1=1, HBO=32, RTL=44 };`**
- Constant names must be different but values can be the same
  - **`enum boolean { FALSE=0, TRUE=1, NO=0, YES=1 };`**

Enumeration - `typedef`

- Use typedef to save some typing

```
enum boolean { FALSE, TRUE };
typedef enum boolean Eboolean;
EBoolean eAnswer = TRUE;
```

- Better yet, combine the typedef and an anonymous enum definition

```
typedef enum { FALSE, TRUE } Eboolean;
EBoolean eAnswer = TRUE;
```

- Typedefs will come in handy later on when we talk about structures and function pointers

Arithmetic Operators

- Basic: **`x+y, x-y, x*y, x/y`**
- Remember:
  - Mismatched operands are promoted to "wider" type:
    `char/short` ⇒ **`int`** ⇒ **`float`** ⇒ **`double`**
  - Integer division truncates the fractional part:
    - **`5/2`** ⇒ **`2`**
    - **`5.0/2`** ⇒ **`2.5`**
    - **`(float)5/2`** ⇒ **`2.5`**

Modulo (%)

- Aka "mod", remainder
  - Should only be applied to positive integers
  - Examples:
    - `13 / 5 == 2`
    - `13 % 5 == 3`
    - is x odd?
    - is x evenly divisible by y?
    - map 765˚ to 0˚ - 360˚ range
    - convert 18:45 to 12-hour format
    - simulate a roll of a six-sided dice
  - Was year 2000 a leap year?
    - Must be divisible by 4 AND must not be divisible by 100, except years divisible by 400 are always leap years
  - How do we code this?

Assignment (= and <op>=)

- Assignment is an expression – its value is the value of the left-hand side after the assignment
  - Regular assignment: `x = x + y`
  - Equivalent way: `x += y`
  - More: `x += y, x -= y, x *= y, x /= y, x %= y`
- The left side of an assignment operator is evaluated only once

```
(c=getchar()) += 1;
```

is different than

```
(c=getchar()) = (c=getchar()) + 1;
```

Increment/Decrement

- Pre-increment/decrement (prefix): **`++x, --x`**
- Post-increment/decrement (postfix): **`x++, x--`**
- **`++x`** acts like **`x = x + 1`** or **`x += 1`**
- However, be careful when using in expressions!
  - **`++x`** increments first and then returns **`x`**
  - **`x++`** returns **`x`** first and then increments

```
int x = 0;
assert(x == 0);
assert(++x == 1);
assert(x == 1);
assert(x++ != 2);
assert(x == 2);
```

Bitwise Operators

- When you need to manipulate/access individual bits
  - Only for integral types (**`char, short, int, long, unsigned/signed`**)
  - Bitwise operators:
    - **`&`** Bitwise AND
    - **`|`** Bitwise inclusive OR
    - **`^`** Bitwise exclusive OR (XOR)
    - **`<<`** Left shift
    - **`>>`** Right shift
    - **`~`** One's complement (unary)
  - With assignment: **`x &= y, x |= y, x ^= y,`**
    `x <<= y, x >>= y`

Bitwise Operators

Examples:

- `&` Bitwise AND `0110 & 0011` ⇒ `0010`
- `|` Bitwise OR `0110 | 0011` ⇒ `0111`
- `^` Bitwise XOR `0110 ^ 0011` ⇒ `0101`
- `<<` Left shift `01101110 << 2` ⇒ `10111000`
- `>>` Right shift `01101110 >> 3` ⇒ `00001101`
- `~` One's complement `~0011` ⇒ `1100`

- Notice: << and >> multiply/divide by 2^n
- `>>` operator may not work as expected on signed types – can perform logical or arithmetical shift (with sign bit duplication)
- Don't confuse bitwise `&` `|` with logical `&&` `||`

Packing Colors into 32 bits

```
/*
 * Format of RGBA colors is
 *  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *  |    alpha      |      red      |     green     |     blue      |
 *  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 */
#define GET_ALPHA(val)     ((val) >> 24)
#define GET_RED(val)       (((val) >> 16) & 0xff)
#define GET_GREEN(val)     (((val) >> 8) & 0xff)
#define GET_BLUE(val)      ((val) & 0xff)
#define MAKE_ARGB(a,r,g,b) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b))
```

Bit Flags

- Can treat each bit as a flag (1=on, 0=off)
  - This allows you to pack up to 32 flags into a single unsigned integer
  - Ex:

```
#define READONLY        0x00000010
#define NOSYSLOCK       0x00000800
#define NOOVERWRITE     0x00001000
#define DISCARD         0x00002000
#define NO_DIRTY_UPDATE 0x00008000
```

  - Use `|` to turn a flag on
    - `int flags = READONLY | DISCARD;`
  - Use `&` to check a flag
    - `if (flags & READONLY) ...`

Logical and Relational Operators

- Logical:
  - `x == y` Equal
  - `x != y` Not equal
  - `x && y` logical AND
  - `x || y` logical OR
  - `!x` NOT
- Relational:
  - `x <  y` Less-than
  - `x <= y` Less-than-or-equal-to
  - `x >  y` Greater-than
  - `x >= y` Greater-than-or-equal-to

Miscellaneous Operators

- **`sizeof`** – Returns the size in bytes

```
int x = 0;
unsigned size = sizeof(int);     ⇒  4
size = sizeof(x);                ⇒  4
```

- ternary
  - **`x ? y : z`**
  - This is short for:
  - if (x) y else z
  - e.g: **`z=(a>b)?a:b; /* z = max(a,b) */`**
- comma
  - **`x, y`**

Associativity and Precedence

- Addition and subtraction associate left to right
  - `4 + 5 + 6 + 7` is equivalent to `(((4 + 5) + 6) + 7)`
- Multiplication, division, and modulo associate left to right
  - `4 * 5 * 6 * 7` is equivalent to `(((4 * 5) * 6) * 7)`
- Assignment operators associate right to left
  - `a = b = c = d` is equivalent to `(a=(b=(c=d)))`
- For complicated expressions with multiple operators, precedence rules determine the order of operation:

  Ex: `c = getchar() != EOF`

  Because `!=` has higher precedence then `=`, the above is equivalent to

  `c = (getchar() != EOF)`

  Definitely not what we wanted!
- When in doubt, or in cases where the expression is non-trivial, use parenthesis
  - `(c = getchar()) != EOF`

Associativity and Precedence

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * (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
`,`	left to right

Side Effects and Evaluation Order

- Function calls, nested assignment statements, and increment and decrement operators cause side effects - some variable is changed as a by-product of the evaluation of an expression.
- In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated.
- C does not specify the order in which the operands of an operator are evaluated, except for **`&&`**, **`||`**, **`?:`**, and `','` operators. In a statement like

```
x = f() + g();
```

  `f` may be evaluated before **`g`** or vice versa.
- Intermediate results can be stored in temporary variables to ensure a particular sequence.
- The order in which function arguments are evaluated is not specified, so the statement

```
printf("%d %d\n", ++n, power(2, n));/* WRONG */
```

  can produce different results with different compilers.
- Another typical situation of this kind is represented by the expression

```
a[i] = i++;
```

Control Flow Overview

- Expressions, statements, and blocks
- `if`, `else`
- `switch`
- Looping
  - `while`
  - `do-while`
  - `for`
  - `break` and `continue`
- `goto` and labels

Expressions, Statements, and Blocks

- We've already seen many examples of these
  - Expressions yield a value: **`x + 1`**, **`x == y`**, etc.
  - Statements are expressions ending with ;
  - Curly braces **`{ }`** are used to group statements into a block
  - Blocks are also used for function bodies and **`if`**, **`else`**, **`while`**,**`for`**, etc.

`if` Statements

- Simple if statement

```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
```

- if-else

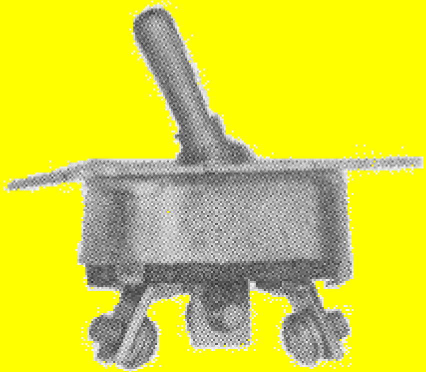
```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
else
 printf("How soon 'till the weekend?\n");
```

- if-else-if-else

```
if (eDay == eMONDAY)
 printf("I hate Mondays!\n");
else if (eDay == eWEDNESDAY)
 printf("The weekend is in sight!\n");
else
 printf("How soon 'till the weekend?\n");
```

`switch` Statements

```
int c = getchar();
switch (c)
{
 case '?':
  printf("Please answer Y or N\n");
  break;
 case 'y': case 'Y':
  printf("Answer is yes\n");
  break;
 case 'n': case 'N':
  printf("Answer is no\n");
  break;
 default:
  printf("By default, the answer is maybe\n");
  break;
}
```



- Multi-way decision test
- Notice: Cases with multiple statements don't require curly braces
- **`default`** is optional but you usually want to include it
- Don't forget break!

`while` and `do-while`

- We've already seen an example

```
while((c = getchar()) != EOF)
  ...
```

- **`while`** checks the condition and then executes the body
- **`do-while`** executes the body and then checks the condition

```
int nDone = 0;
do {
  ...
} while (!nDone);
```

`for` Statement

- Compact looping statement

```
for(expr1; expr2; expr3)
{
  statements
}
```

  - This is equivalent to

```
expr1;
while (expr2)
{
  statements
  expr3;
}
```

  - **`expr1, expr2, expr3`** are optional

`for` Statement – Examples

- Print 4 spaces

```
for(i = 0; i < 4; ++i)
  putchar(' ');
```

- Print the alphabet

```
for(c = 'a'; c <= 'z'; ++c)
  printf("%c ", c);
```

- Print even digits between 0 and 100

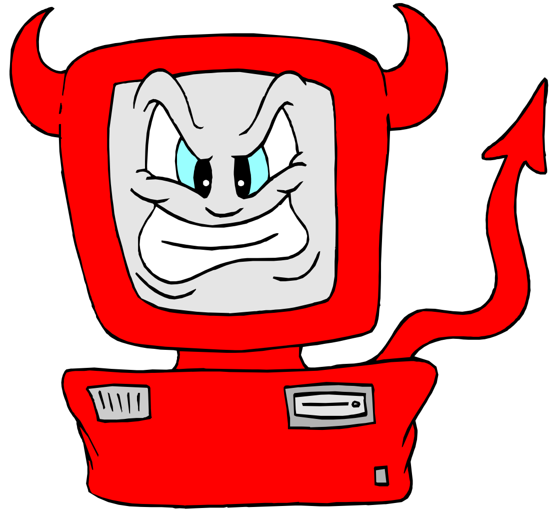
```
for(n = 0; n <= 100; n += 2)
  printf("%d ", n);
```

- When to use **`while`**, **`do-while`**, **`for`**?

`break` and `continue`

- **`break`**
  - Use **`break`** to break out of a loop (**`while`**, **`do-while`**, **`for`**)
  - First statement after the loop will be executed
- **`continue`**
  - Skips the remaining statements in the loop body
  - Proceeds to loop condition (**`while`** and **`do-while`**) or **`expr3`** (**`for`**)

`goto` Statement and Labels



```
goto label;
...
label:
```

- Causes program execution to jump to the label
- Used indiscriminately, goto is evil and leads to spaghetti code
- Two cases where its permissible:
  - Breaking out of a nested loop
  - Executing cleanup code

Arrays

- Simplest aggregates
- Fixed length (we'll cover dynamic arrays later)
  - All elements are the same type
  - Kinds of arrays
    - Character arrays (strings)
    - Other arrays
    - Multi-dimensional

Character arrays ("strings")

```
const char szMsg[] = "compiler";
```

- This is stored as an array of characters terminated with a `'\0'` (**`NUL`**) to mark the end

`c`	`o`	`m`	`p`	`i`	`l`	`e`	`r`	`\0`

- First element of the array starts at index 0
  - **`szMsg[3]`** refers to the 4th char (not 3rd) ⇒ **`'p'`**
  - **`sizeof(szMsg)`** = size of the array in bytes = **9** (don't forget the `'\0'`!)
- Number of elements
  - = array size / element size
  - = **`sizeof(szMsg)/sizeof(char)`**

Character arrays

- Let's create another string

```
char szMyMsg[4];
szMyMsg[0] = 'f';
szMyMsg[1] = 'o';
szMyMsg[2] = 'o';
szMyMsg[3] = '\0'; /* Did you forget this? */
```

  - Here's another way to initialize a string

```
char szMyMsg[] = { 'f', 'o', 'o', '\0' };
```

Other Arrays and Initialization

- Arrays can be any data type, including other arrays!

```
int aryDigitCount[10];  /* uninitialized array */
```

  - Can initialize an array with the `={}` notation

    ```
    int aryDays[]= { 31, 28, 31, 30, 31, 30, 31,
    31, 30, 31, 30, 31};
    ```

  - In this case, you can leave out the element count because the compiler can figure it out.
  - If element count is specified and the number of initializers is less, the compiler will fill the remaining elements with 0. This provides a handy way to initialize an array with all zeros:

    ```
    int aryDigitCount[10] = { 0 };
    ```

  - You should always initialize automatic arrays; don't assume they are initialized to 0

Array sizes

- Given a string, how do we determine its length?
  - Given an arbitrary array, how do we determine the number of elements?
  - Can't use **`sizeof`** if the array is passed into a function
  - Number of elements of an array is usually obtained from:
    - a terminating element (`'\0'` for strings, `0` for **`argv`**)
    - a separate count variable (e.g. **`argc`**)
    - count encoded in the data somehow (e.g. **`BSTR`**)
    - a constant (e.g. **`MAX_SIZE`**)
  - How can we write **`strlen()`**?
  - What is a disadvantage of using a terminating element?

2D Arrays

```
char arySmiley[4][8] = {
" -- -- ",
" @   @ ",
"   +   ",
" |---/ ", /* trailing comma is legal */
};
```

- This is an array of 4 strings each with 8 chars (don't forget `\0`!)
- A 2D array is really a 1D array, each of whose elements is an array
- What is the size in bytes?

2D Arrays

- Suppose we want to add colors to the smiley
  - Store an RGB value, packed in an **`int`** as **`0x0rgb`**, for each element

```
/* Initialize all colors to black */
unsigned long arySmileyColors[4][7] = { 0L };

/* Paint eyebrows, nose, and chin white */
arySmileyColors[0][1] = 0xFFFFFFL;
arySmileyColors[0][2] = 0xFFFFFFL;
arySmileyColors[0][4] = 0xFFFFFFL;
arySmileyColors[0][5] = 0xFFFFFFL;
arySmileyColors[2][3] = 0xFFFFFFL;
arySmileyColors[3][1] = 0xFFFFFFL;
arySmileyColors[3][5] = 0xFFFFFFL;
```

  - How do we paint the eyes and mouth?
  - Why only 7 **`int`**s when there are 8 **`char`**s?

Array Caveats

- You must make sure you access only valid array elements!
  - Accessing/modifying elements that are out-of-bounds in C has undefined consequences!
  - **`ary[-1]`** and **`ary[999]`** will not generate any compiler errors
  - If you are lucky(!), program crashes with

    `Segmentation fault (core dumped)`
  - What's wrong with this code?

```
int aryChrCount[26] = { 0 };   /* A-Z */
char c = '\0';
while ((c = getchar()) != EOF)
  ++aryChrCount[c];
```

Function Definition

format of a
function definition:

```
type func_name( parameter_list )
{
   declarations
   statements
}
```

header

body

declara-

tions

statements

```
int fact(int n)
{
     int i, product = 1;

     for (i = 1; i<=n; ++i)
         product *= i;
     return product;
}
```

```
int main(void)
{
        int m = 12;

     printf("%d\n",fact(m));
     return 0;
}
```

Function Header

```
type   func_name( parameter_list )
```

function name

list of arguments:

```
type parameter_name
```

type returned by the function

(`void` if no value returned)

multiple arguments
are separated by commas

`void` if no parameters

Examples:

```
int fact(int n)

void error_message(int errorcode)

double initial_value(void)

int main(void)
```

Usage:

```
a = fact(13);

error_message(2);

x=initial_value();
```

Why Use Functions?

- Write your code as collections of small functions to make your program modular
  - structured programming
  - code easier to debug
  - easier modification
  - reusable in other programs

Function Prototypes

- If a function is not defined before it is used, it must be declared by specifying the return type and the types of the parameters

```
double sqrt(double);
```

  - tells the compiler that the function **`sqrt()`** takes an argument of type double and returns a double.
  - This means, incidentally, that variables will be cast to the correct type; so **`sqrt(4)`** will return the correct value even though **`4`** is **`int`** not **`double`**.
- These function prototypes are placed at the top of the program, or in a separate header file, **`file.h`**, included as **`#include "file.h"`**
- Variable names in the argument list of a function declaration are optional:

```
void f (char, int);
void f (char c, int i); /*equivalent but makes code more readable */
```

- If all functions are defined before they are used, no prototypes are needed. In this case, **`main()`** is the last function of the program.

Function Calls

- When a function is called, this is what happpens:
  - expressions in the parameter list are evaluated (in no particular order!)
  - results are transformed to the required type
  - parameters are copied to local variables for the function
  - function body is executed
  - when return is encountered, the function is terminated and the result (specified in the return statement) is passed to the calling function (for example main)

```
int fact (int n)
{
    int i, product = 1;
    for (i = 2; i <= n; ++i)
                  product *= i;
    return product;
}
```

```
int main (void)
{
    int i = 12;
    printf("%d",fact(i));
    return 0;
}
```

Scope Rules for Blocks

- Identifiers (i.e. variables etc.) are accessible only within the block in which they are declared.

```
{
    int a = 2;                   /* outer block a */
    printf("%d\n", a);           /* 2 is printed  */
    {
        int a = 5;               /* inner block a */
        printf("%d\n", a);       /* 5 is printed  */
    }
    printf("%d\n", a);           /* 2 is printed  */
}
/* a no longer defined */
```

outer a masked

- A variable that is declared in an outer block is available in the inner block unless it is redeclared. In this case the outer block declaration is temporarily "masked".
- Avoid masking!
  Use different identifiers instead to keep your code debuggable!

Scope Rules for Functions

- Variables defined within a function (including main) are local to this function and no other function has direct access to them!
  - the only way to pass variables to a function is as parameters
  - the only way to pass (a single) variable back to the calling function is via the return statement

```
int func (int n)
{
    printf("%d\n",b);
    return n;
}
```

```
int main (void)
{
    int a = 2, b = 1, c;
    c = func(a);
    return 0;
}
```

b not defined locally!

- Exceptions:
  - Global Variables
  - Pointers

Global Variables

- Variables defined outside blocks and functions are global, i.e. available to all blocks and functions that follow
- Avoid using global variables to pass parameters to functions!
- Only when all variables in a function are local, it can be used in different programs
- Global variables are confusing in long code

```
#include <stdio.h>

int a = 1, b = 2; /* global variables */

int main (void)
{
   int b = 5;   /* local redefinition */
   printf(“%d”, a+b); /* 6 is printed */
   return 0;
}
```

Call by Value

- Arguments to functions are evaluated, and the copies of the values – not any variables in the argument – are passed down to the function
  - Good: protects variables in calling function
  - Bad: copying inefficient, for example for large arrays ⇒ pointers

```
#include <stdio.h>
int     compute_sum (int n);   /* function prototype        */
/* ------------------------------------------------------- */
int main(void)
{
        int n = 3, sum;
        printf("%d\n", n);     /* 3 is printed              */
        sum = compute_sum(n);  /* pass value 3 down to func */
        printf("%d\n", n);     /* 3 is printed -  unchanged */
        printf("%d\n",sum);    /* 6 is printed              */
        return 0;
}
/* ------------------------------------------------------- */
int compute_sum (int n)        /* sum integers 1 to n       */
{
        int sum = 0;
        for ( ; n > 0; --n)    /* local value of n changes  */
                sum += n;
        return sum;
}
```

n unchanged

local copy of n, independent of n in calling function

Storage Classes

- Every variable and function in C has two attributes:
  - type (`int, float`, ...)
  - storage class
- Storage class is related to the scope of the variable
- There are four storage classes:
  - `auto`
  - `extern`
  - `register`
  - `static`
- `auto` is the default and the most common
- Memory for automatic variables is allocated when a block or function is entered. They are defined and are "local" to the block. When the block is exited, the system releases the memory that was allocated to the `auto` variables, and their values are lost.
- Declaration:
  - `auto type variable_name;`
- There's no point in using auto, as it's implicitly there anyway

extern

- Global variables (defined outside functions) and all functions are of the storage class **`extern`** or **`static`** and storage is permanently assigned to them
- To access an external variable, which is defined elsewhere, the following declaration is used:

  `extern type variable_name;`

  - it tells the compiler, that the variable **`variable_name`** with the external storage class is defined somewhere in the program
- Within a file variables defined outside functions have external storage class
- Files can be compiled separately, even for one program. **`extern`** is used for global variables that are shared across code in several files

`extern` in Multi-File projects

```
/*file1.c*/
#include <stdio.h>
int a =1, b = 2, c = 3; /* external variables */
int f(void);
int main (void)
{
    printf("%3d\n", f( ));
    printf("%3d%3d%3d\n", a, b, c);        print 4, 2, 3
    return 0;
}                                 a is global and changed by f

/*file2.c*/
int f(void)
{
    extern int a;  /* look for it elsewhere */
    int b, c;                  b and c are local and don't survive
    a = b = c = 4;
    return (a + b + c);
}                                        return 12
```

- compile as: `gcc file1.c file2.c -o prog`

static

- Static variables are local variables that keep their previous value when the block is reentered. A declaration
  - `static int cnt = 0;`

  will set cnt to zero the first time the function is used; thereafter, it will retain its value from previous iterations.
- This can be useful, e.g., for debugging: you can insert code like this anywhere without interfering with the rest of the program

```
{ /* debugging starts here */
  static int cnt = 0;
  printf("*** debug: cnt = %d, v = %d\n",++cnt, v);
}
```

- The variable cnt is local to the block and won't interfere with another variable of the same name elsewhere in an outer block; it just increases by one every time this block is encountered.
- `static` can be also applied to global variables, it means, that they are local to a file and inaccessible from the other files
- If not initialized explicitly, static and global variables are initialized to 0

Recursion

- To understand recursion, you must first understand recursion.
- A function is called recursive if it calls itself, either directly or indirectly.
- In C, all functions can be used recursively.
  - Example:

```
int sum(int n)
{
  if (n <= 1)
   return n;
  else
   return (n + sum(n - 1));
}
```

- If you don‘t want to generate an infinite loop, you must provide a condition to end the recursion (here **`n<=1`**), which is eventually met.
- Recursion is often inefficient as it requires many function calls.

Example: Fibonacci Numbers

- A recursive function for Fibonacci numbers (0,1,1,2,3,5,8,13...)

```
int fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return (fibonacci(n-1) + fibonacci(n-2));
}
```

- 1.4 x 10^9 function calls needed to find the 43^{rd} Fibonacci number! (which has the value 433494437)
- If possible, it is better to write iterative functions

```
int factorial (int n)   /* iterative version */
{
    for ( ; n > 1; --n)
            product *= n;
    return product;
}
```

Assertions

If you include the directive

`#include <assert.h>`

you can use the "assert" macro: this aborts the program if an assertion is not true.

You can disable assertions if you **`#define NDEBUG`**

```
#include <assert.h>
#include <stdio.h>
int f(int a, int b);
int g(int c);

int main(void)
{
int a, b, c;
.....
scanf("%d%d", &a, &b);
.....
c = f(a,b);
assert(c > 0);/* an assertion */
.....
}
```