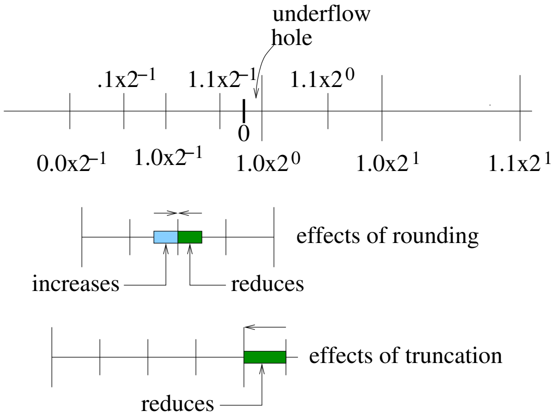
Floating Point Numbers

Problem in Representing Exact Value

- Suppose use 3 bit exponent and 4 bit mantissa, and we want to represent 80.
- With 4-bit normalized mantissa
  - maximum value: $.1111 = .5 + .25 + .125 + .0625 = .9375$
  - minimum value: $.1000 = .5$
- Since mantissa < 1 and $2^6 = 64 < 80$, the exponent should be $2^7 = 128$.
- $128 \times .6875$ (1011) $= 88 > 80$, so, mantissa which can take us near 80: $.1001 = .5625$ and $.1000 = .5$.
- However, both $128\times .5$ (=64), $128 \times .5625$ (=72) are less than 80.

Floating Point Numbers

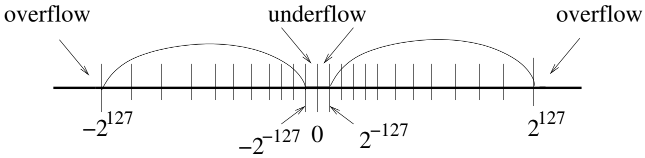
Problem in Representing Exact Value



Floating Point Numbers

Underflow, Overflow & Rounding Off

- Spacing between floating point numbers is not uniform.
    - Eg., $.1 \times 2^1 - .11 \times 2^1 = .5$ which much smaller compared to $.11 \times 2^{127} - .1 \times 2^{127} = 4.25353 \times 10^{37}$.
    - But if difference expressed with relative to corresponding numbers then they are the same (.5 times).
- The Least possible exponent is -126, so underflow occurs in interval -2^{-126} and 2^{-126}.



Floating Point Numbers

Underflow, Overflow & Rounding Off

- No value in the said interval can be represented except for 0.
- Similarly overflow occurs after -2^{127} and also beyond 2^{127}.
- Mantissa is restricted to 23 bits, so there is a gap between any two successive floating point number.
- Rounding-off occurs if exact value of a calculation can not be represented.

Examples

Number	Sign bit	Exponent in excess 16	Mantissa
$x = 0.0001101001101 \times 2^0$	**0**	**0000**	**000110100**
$x = 0.001101001101 \times 2^{-1}$	**0**	**1111**	**001101000**
$x = 0.01101001101 \times 2^{-2}$	**0**	**1110**	**011010000**
$x = 0.1101001101 \times 2^{-3}$	**0**	**1101**	**110100000**
$x = 1.101001101 \times 2^{-4}$	**0**	**1100**	**101000000**

An implied 1.0 exists, and by normalization highest precision is achieved.

- Biased exponent or excess representation achieves two important simplification.
  - No need to deal with sign of the exponent, i.e., 2's complement representation is avoided.
  - Integer sorting can be used to simplify the comparison of exponents.

Examples

With excess 16 representation 5-bit exponent field (range $-2^4 : 2^4 - 1$) will be:

Exponent	2's complement	Biased notation	Value in excess-16
15	01111	11111	31
14	01110	11110	30
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	00001	10001	17
0	00000	10000	16
-1	11111	01111	15
$\vdots$	$\vdots$	$\vdots$	$\vdots$
-15	10001	00001	1
-16	10000	00000	0

Examples

- Let us represent -0.75 in biased notation with $e = 5$ bits.
- -0.75$_{10}$ = -0.11$_2$ = $(-1.1 \times 2^{-1})_2$
- Biased exponent = -1 + 16 = 15.
- Without implied 1: the representation is **1 | 10000 | 1100...**
- With implied 1: the representation is **1 | 01111 | 1000...**

Arrays

Why Arrays?

- A collection of similar elements each of which may require same type of processing
- Basic advantage: lets us use one variable to access all elements systematically.
- Conceptually analogous to mathematical abstractions such as table, vectors, matrices.
- The individual elements can be accessed by associating indices to variable name.

Arrays

One Dimensional Array

- 1-D array declared as: `int a[8];`

a →

10	20	30	40	50	60	70	80
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

- Array size is important in declaration.
- Size can be specied either as shown above or as follows:

```
#define N 9
:
int a[N]
```

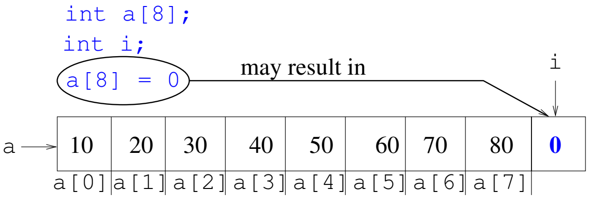
Arrays

One Dimensional Array

- `a[i]` is an **lvalue**, so it can be used in same way as a scalar variable.
- Each element `a[i]` is treated as `int` type.

Important Points

- Array bound is not checked.
- So, `a[9]` may have side-eects as indicated below.



Arrays

Expression for Array Indices

- Care must be taken in using expression for indices.
- Eg., in the following code, assignment to non-existing a[10] causes an overwrite on the next available memory location or i.

```
int a[10];
int i = 0;
while (i <= 0) {
    a[i] = 0; // Causes and overwrite at i for i=10
    i++;
}
```

- An innite loop results due to overwrite on location i.
- So, be careful when loop index has a side-effect.

Arrays

Reading and Printing

```
#include <stdio.h>
#define N 10
int main() {
    double a[N];
    int i;

    for (i = 0; i < N; i++)
        scanf("%lf", &a[i]); // read N elements one by one
    for (i = N-1; i > 0; i--)
        printf("%.3f   " a[i]); // print in reverse order
    pritf("\n");
}
```

Array Initialization

Some Examples

Like other scalar variables, array can also be initized.

```
/* Initial values are: a[0] = 1, a[1] = 2 *
 * a[2] = 3, a[3] = 4 and a[4] = 5         */

   int a[5] = {1, 2, 3, 4, 5};

/* Initial values are: a[5] to a[9] = 0 */

   int a[10] = {1, 2, 3, 4, 5};

/* Initial values are: a[0] to a[9] = 0 */

   int a[10] = {0};

/* Array length determined by initializer */

   int a[] = {1, 2, 3, 4, 5}
```

Array Initialization

Designated Initializer

- Suitable for arrays having few non zero elements
- Example below shows how an array of 15 elements having only 3 non zero elements can be initialized.

```
/* Designated intializers */
int a[15] = {[3] = 19, [12] = 14, [13] = 23}
/* Length determined by larged designated initializer */
int a[] = {[3] = 19, [12] = 14, [30] = 23}
```