We have used three standard data types: `int`, `double`, and `char`.

- Type `int` values are used in C to represent both the numeric concept of an integer and the logical concepts `true` and `false`.
- Standard types and user-defined enumerated types are **simple**, or **scalar**, data types because only a single value can be stored in a variable of each type.

### Differences Between Numeric Types

Uses of different data types:

- Data type `double` can be used for (many) “real” numbers.
- But:
  - Operations involving integers are faster than those involving `double`.
  - Less storage space is needed to store type `int` values (32-bits versus 64-bits).
  - Operations with integers are always precise, whereas some loss of accuracy can occur when dealing with type `double` numbers.
- These differences result from the way numbers are represented in the computer’s memory.

### Round-off Error I

Example

```c
1 b = 2.0;
2 printf("b = %.20f\n",b);
3 b = sqrt(2.0);
4 b = pow(b,2.0);
5 printf("b = %.20f\n",b);
```

Output:

```plaintext
1 b = 2.00000000000000000000
2 b = 2.00000000000000044409
```
Round-off Error II

Example

- Certain numbers, such as $\sqrt{2}$ are approximated
- Internally, some interpolation method(s) are used
- Thus, $\sqrt{2.0} = 1.41421356...$ is not accurate in the lower order digits

Base 10

- Humans have 10 fingers, so we naturally developed counting in base-10
- Single numbers are represented using arabic numerals 0 thru 9
- Single numbers are multiplied by various powers of 10 and added together to form any possible number in base-10

$$532.41 = 5 \times 10^2 + 3 \times 10^1 + 2 \times 10^0 + 4 \times 10^{-1} + 1 \times 10^{-2}$$

Base Conversion

- Algorithms for conversion are straightforward, but beyond this course
- Necessary to simply understand what's going on behind the scenes

$$110.011 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 4 + 2 + 0 + 0 + \frac{1}{4} + \frac{1}{8} = 6.375$$

Base 2

- Computers do not have fingers; they only have states: on/off, high-voltage/low-voltage, etc.
- Thus, computers work in base-2: binary
- All data are represented in memory as binary strings, strings of 0s and 1s.
- Single numbers are limited to 0, 1
- Single numbers are multiplied by various powers of 2 and added together to form any possible number in base-2

$$\cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots$$

Data Representation I

- The binary string stored for type in value 13 is not the same as the binary string stored for 13.0
- Integers are whole numbers and so have a definite end (nothing to the right of the decimal)
- Floating point numbers could have any number of bits to the left and to the right of the decimal
- In order to represent a larger range of values, computers store floating point numbers in a manner analogous to scientific notation

Scientific Notation I

- Scientific Notation (base-10): any number is “normalized” by shifting the decimal place so that it is between 0 and 10:

$$1.4326123 \rightarrow 1.4326123E4$$
$$0.00529 \rightarrow 5.29E-3$$

- Similarly, floating point numbers (double, float) are divided into two sections: the mantissa and the exponent
Scientific Notation II

- The mantissa is a binary fraction between $\frac{1}{2} = 2^{-1}$ and $1 = 2^0$ for positive numbers and between $-0.5$ and $-1.0$ for negative numbers.
- The exponent is an integer.
- The mantissa and exponent are chosen so that:

$$\text{real number} = \text{mantissa} \times 2^{\text{exponent}}$$

- Because of the finite size of memory cell, not all real numbers in the range allowed can be represented precisely as type double values (irrationals, even rationals such as $1/3$).

Mantissa-Exponent Example I

- Let 110.011 be our number as before
- Shift right to get the Mantissa: .110011
- We had to shift 3 places, so the exponent is 3 (in binary, that is 11)

Mantissa-Exponent Error I

- When converting to the mantissa-exponent format, we may shift lower order bits out of range (we can only hold 64 bits)
- Bits shifted out of range are dropped, leading to round off error

Range of Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Approximate Range</th>
<th>Significant Digits</th>
<th>Bits (CSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>$10^{-38}$ to $10^{38}$</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>$10^{-308}$ to $10^{308}$</td>
<td>15</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>$10^{-4931}$ to $10^{4931}$</td>
<td>19</td>
<td>128</td>
</tr>
</tbody>
</table>

Numerical Inaccuracies I

- Representation error: some fractions cannot be represented in the decimal number system (e.g., $1/3$ is 0.3333...), some fractions cannot be represented exactly as binary numbers in the type double format.
  - Sometimes called round-off error
  - This depends on the number of binary digits used in the mantissa. More bits $\rightarrow$ smaller error.
  - Because of this kind of error, an equality comparison of two type double values can lead to surprising results.
  - for(i=0.0; i != 10.0; i+=0.1) ...
  - Problems can occur when manipulating very large and very small real numbers.

Table: Range of values typical in most C implementations

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>$-32,767$ ... $32,767$</td>
</tr>
<tr>
<td>unsigned short</td>
<td>$0$ ... $65,535$</td>
</tr>
<tr>
<td>int</td>
<td>$-32,767$ ... $32,767$</td>
</tr>
<tr>
<td>unsigned int</td>
<td>$0$ ... $65,535$</td>
</tr>
<tr>
<td>long int</td>
<td>$-2,147,483,647$ ... $2,147,483,647$</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>$0$ ... $4,294,967,295$</td>
</tr>
</tbody>
</table>

Table: Range of values according to the ANSI C specification

<table>
<thead>
<tr>
<th>Type</th>
<th>Approximate Range</th>
<th>Significant Digits</th>
<th>Bits (CSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>$10^{-38}$ to $10^{38}$</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>$10^{-308}$ to $10^{308}$</td>
<td>15</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>$10^{-4931}$ to $10^{4931}$</td>
<td>19</td>
<td>128</td>
</tr>
</tbody>
</table>
Numerical Inaccuracies II

- **Cancelation error**: Adding a small number to a large number, the larger number may “cancel out” the smaller number.
- If \( x \) is much larger than \( y \), then \( x + y \) may have the same value as \( x \) (example: \( 1000.0 + 0.0000001234 \) is equal to 1000.0 on some computers).
- **Arithmetic underflow**: Multiplying small numbers may cause the result to be too small to be represented accurately, so it will be represented as zero.
- **Arithmetic overflow**: adding/multiplying large number (recall: 13!)

Round-Off Error Example

Recall the round cents function we wrote:

```c
double roundCents(double m) {
    double x;
    x = floor(x);
    x = x / 100;
    printf("m-x = %.50f
", m - x);
    if(m - x >= 0.005) {
        x = x + .01;
    }
    return x;
}
```

Automatic Conversion of Data Types

- **k + x**: \( k \) is converted to \( 3.15 \), but then is converted to \( 3 \) to store it in a type int
- These conversions are automatic and implicit

Explicit Conversion of Data Types

- In addition to automatic conversions, C also provides an explicit type conversion operation called a cast:
- \( z = (\text{double})k / (\text{double})m; \)
- The value to be converted causes the value to change to double data format before it is used in the computation
- Casting is a very high precedence operation, so it is performed before the division
- \( (\text{double})(k/m) \) will do \( k/m \) first: The highest precedence operator is always the parentheses

Enumerated Types

- Certain programming problems require new data types
- Ex: it makes sense in a calendar program to be able to distinguish between months
- C allows you to associate a numeric code with each category by creating an enumerated type that has its own list of meaningful values.

```c
typedef enum {
    january, february, march, april, may,
    june, july, august, september, october,
    november, december
} month_t;
```
Enumerated Types

- Defining type month as shown causes the enumeration constant `january` to be represented as the integer 0, `february` to be represented as integer 1, etc.
- Variable `month` and the twelve enumeration constants can be manipulated just as one would handle any other integers.
- Like variables and functions, user defined types must be defined before you use them.
- Enumerated types are integers, the keywords associated with them cannot be printed.
- Be careful when doing arithmetic operations on enumerated types.

```c
month_t myMonth;
myMonth = january;
myMonth++;
if (myMonth == february)
    printf("True");
else
    printf("False");
printf("myMonth = %d", myMonth);
myMonth = myMonth + 100000;
```

Common Programming Errors I

- Arithmetic underflow and overflow resulting from a poor choice of variable type are common causes of run-time errors.
- Programs that approximate solutions need to be careful of rounding errors.
- When defining enumerated types, only identifiers can appear in the list of values for the type.

Common Programming Errors II

- Do not reuse one of the identifiers in another type, as a variable name, etc. (compile error)
- Keep in mind that there is no built-in facility for input/output of the identifiers that are the valid values of an enumerated type. You must either display the underlying integer representation or write your own input/output functions.