Chapter 7

7.1 Representation and conversion of numeric types
7.2 Representation and conversion of type char
7.3 Enumerated Types
7.5 Common Programming Errors
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.
Representation and Conversion of Numeric Types

- Differences Between Numeric Types
- Numerical Inaccuracies
- Automatic Conversion of Data Types
- Explicit Conversion of Data Types
Uses of different data types:

- Data type `double` can be used for all numbers.
- But:
  - Operations involving integers are faster than `double`.
  - Less storage space is needed to store type `int` values.
  - 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.
All data are represented in memory as *binary strings*, strings of 0s and 1s.

- The binary string stored for type in value 13 is not the same as the binary string stored for 13.0.
- Positive integers are represented by standard binary numbers, $13 = 01101$.
- The format of type `double`, or *floating-point*, values is analogous to scientific notation $\rightarrow$ i.e. $3.141592 \times 10^0$ is $\pi$.
- Similarly, for `double` values, the storage area occupied by the number is divided into two sections: the *mantissa* and the *exponent*.
  - The mantissa is a binary fraction between .5 and 1.0 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`.
## Size of int/double

<table>
<thead>
<tr>
<th>Type</th>
<th>Range in ANSI standards</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>
<thead>
<tr>
<th>Type</th>
<th>Approximate Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>(10^{-37} \ldots 10^{38})</td>
</tr>
<tr>
<td>double</td>
<td>(10^{-307} \ldots 10^{308})</td>
</tr>
<tr>
<td>long double</td>
<td>(10^{-4931} \ldots 10^{4932})</td>
</tr>
</tbody>
</table>
Numerical Inaccuracies

One of the problems in processing data of type double is that sometimes an error occurs in representing real numbers.

- **Representation error:** Just as 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) ...`
Inaccuracies cont’d...

- Problems can occur when manipulating very large and very small real numbers.
  - **Cancellation error** Adding a small number to a large number, the larger number may “cancel out” the smaller number.
  - If $x$ is much larger than $y$, the $x + y$ may have the same value as $x$ (for 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**: Use your imagination for this one.
In Chapter 2, we saw several cases in which data of one numeric type were automatically converted to another numeric type.

```c
int    k = 5,    m = 4,    n;
double x = 1.5, y = 2.1, z;

k + x, conversion is done before + since x is of type double

z = k / m, conversion is done after / since k and m are both of type int, thus we get 1

n = x * y, we compute x * y to get 3.15 and then converted to type int and 3 is stored in n
```
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.
Representation and Conversion of Type char

- The data type char allows us to store and manipulate individual characters.
- Variables of type char have been used to store type char constants consisting of a single character enclosed in apostrophes.
- How does C compute ’A’ < ’Z’?
  - Each character has its own unique numeric code, the binary form of this code is stored in a memory cell that has a character value, see Appendix A for ASCII, EBCDIC, and CDC formats.
  - Thus ’A’ equals 65, ’Z’ equals 90, and ’|’ equals 108, thus ’A’ < ’Z’ is true and ’A’ < ’|’ is also true.
Enumerated Types

- Good solutions to many programming problems require new data types.
  - In a calendar program you might need to distinguish between the different months: january, february, march, april, may, june, july, august, september, october, november, december.
- 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;

month_t month;
```
Enumerated Types

• Defining type month as shown causes the enumeration constant january to be represented as the integer 0, constant february to be represented as integer 1, and so on.

• Variable month and the twelve enumeration constants can be manipulated just as one would handle any other integers.

```c
month = january;
month++;
if (month == february)
    printf("True");
else
    printf("False");

month = month + 100000;\n```
Common Programming Errors

• Predicting and hand-checking the results of every program is especially important because of the way C represents the various data types.
  • Arithmetic underflow and overflow resulting from a poor choice of variable type are common causes of erroneous results.
  • 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.

• Be careful not to reuse one of the identifiers in another type, or as a variable name in a function that needs your type definition.

• 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 scan and display the underlying integer representation or write your own input/output functions.