

Computer Science & Engineering 150A

Problem Solving Using Computers

Lecture 07 - Simple Data Types

Christopher M. Bourke

Spring 2009

- 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.

- 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 (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

Example

CSCE150A

Introduction

Automatic
ConversionEnumerated
Types

```
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:

```
1  b = 2.000000000000000000000000
2  b = 2.00000000000000000044409
```

- 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: $3.14192e0 = 3.141592 \times 10^0$ 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:

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

- 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 $1/3$).

Table: Range of values typical in most C implementations

Type	Range
<code>short</code>	-32,767 ... 32,767
<code>unsigned short</code>	0 ... 65,535
<code>int</code>	-32,767 ... 32,767
<code>unsigned int</code>	0 ... 65,535
<code>long int</code>	-2,147,483,647 ... 2,147,483,647
<code>unsigned long int</code>	0 ... 4,294,967,295

Table: Range of values according to the ANSI C specification

Type	Approximate Range	Significant Digits	Bits (CSE)
<code>float</code>	$10^{-37} \dots 10^{38}$	6	32
<code>double</code>	$10^{-307} \dots 10^{308}$	15	64
<code>long double</code>	$10^{-4931} \dots 10^{4932}$	19	128

- **Representation error:** some fractions cannot be represented in the decimal number system (e.g., $1/3$ is $0.3333\dots$), 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 \longrightarrow 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.

- **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

CSCE150A

Introduction

Automatic
ConversionEnumerated
Types

Recall the round cents function we wrote:

```
1  double roundCents(double m) {
2      double x;
3      x = m * 100;
4      x = floor(x);
5      x = x / 100;
6
7      printf("m-x = %.50f\n", m-x);
8      if(m - x >= 0.005)
9      {
10         x = x + .01;
11     }
12     return x;
13 }
```

First run:

```
1  enter a number: 100.075
2  m-x = 0.0050000000000000966338120633736252784729003906250000
3  The rounded value is: 100.080000
```

Second run:

```
1  enter a number: 171.065
2  m-x = 0.0049999999999999545252649113535881042480468750000000
3  The rounded value is: 171.060000
```

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

- $k + x$: k is converted before adding since x is of type `double`
- $z = k / m$: conversion is done after division, since both operands are of type `int`
- $n = x * y$: $x * y$ evaluates to get 3.15, but then is converted to 3 to store it in a type `int`
- These conversions are *automatic* and *implicit*

- In addition to automatic conversions, C also provides an explicit type conversion operation called a **cast**:

```
z = (double)k / (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
- `(double)(k/m)` will do `k/m` first: The highest precedence operator is always the parentheses

- 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.

```
1 typedef enum {  
2     january, february, march, april, may,  
3     june, july, august, september, october,  
4     november, december}  
5 month_t;
```

- 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

```
1     month_t myMonth;  
2     myMonth = january;  
3     myMonth++;  
4     if (myMonth == february)  
5         printf("True");  
6     else  
7         printf("False");  
8  
9     printf("myMonth = %d", myMonth);  
10    myMonth = myMonth + 100000;
```

- 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.

- 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.