CSIS/COMP 1117B
Computer Programming

Computing with Numbers

Computing with Numbers

- Integers
- Expression revisited
- Reals
- Computing with floating-point numbers

Integers

- Introduction
- Integer representation
- Integer overflow
- Unsigned integers

Introduction

- Mathematically, there is an infinite number of integers:

…, -3, -2, -1, 0, 1, 2, 3, …

- An integer m written as a sequence of digits $d_n d_{n-1} \ldots d_0$ has value:

$$m = d_n \times 10^n + d_{n-1} \times 10^{n-1} + \ldots + d_0$$

- 10 is known as the **base** of the representation of m (as a sequence of digits) and each digit is between 0 and 9 (base – 1)

Integer Representation

- Integers are represented in base 2 (**binary system**); the same *m* will be represented by a much longer sequence of 0s and 1s $b_k b_{k-1} \ldots b_0$:
  $m = b_k \times 2^k + b_{k-1} \times 2^{k-1} + \ldots + b_0$
- For example, the binary representation of 3322_{10} is 110011111010_2
- As binary numbers tend to be very long, we sometimes write them in base 8 or base 16; 3322_{10} as 6372_8 or CFA_{16}

Two's Complement

- Recall that the basic addressable unit of storage is 8 bits (1 byte). What is the range of integers that can be represented using 8 bits?
  – between -128 (-2^{8-1}) and 127 ($2^{8-1}-1$)
  – notice that we have 1 fewer positive number
  – all positive numbers have a leading 0
  – all negative numbers have a leading 1
- 8-bit, 16-bit, 32-bit, and 64-bit integers are all available in C/C++; they have types: char, int, long, long long respectively.

Representing Negative Numbers

- To determine the binary representation of a negative integer –*m* in 2’s complement:
  1. generate the binary representation of *m*
  2. invert the bits of the binary representation of *m*; that is, 1 becomes 0 and 0 becomes 1
  3. add 1 to the inverted representation and the result is the 2’complement representation of –*m*
- For example, the 16-bit representation of -1 is:
  1. 0000000000000001_2
  2. 1111111111111110_2
  3. 1111111111111111_2

Two's Complement Arithmetic

- The sum of 1 and -1 from the previous example is:
  10000000000000000_2
- Note that the result is correct if we only consider the lower 16 bits; but, the result actually required 17 bits for its representation, is there an *overflow*?
- The above is ***not*** an overflow situation because the result is 0 and can be represented using 16 bits.
- Note that it is actually wrong to include the leading 1 in the result! Operations on 2's complement numbers require that operands and results are represented by the ***same number of bits***.

Integer Overflow

- An **overflow** is said to have occurred if the result of an arithmetic operation ***exceeds the representable range***.
- Consider 16-bit integers, the largest integer is
  0111111111111111_2
- If we add 1 to it, we get
  1000000000000000_2
- Which is actually the smallest 16-bit integer and the result fits in 16 bits!

Test of Integer Overflow

- Note that the sum of two integers of opposite signs will ***never*** overflow.
- Overflow occurs when two integers of the same sign generates a result of the opposite sign.
- In our previous example, both 32767 and 1 are positive numbers but the result is -32768; therefore, an overflow has occurred in the operation.
- ***C/C++ does not report integer overflow!!***

The Standard Library <climits>

- The maximum and minimum of various integer types are defined in the standard library <climits>
- For example:
  - **CHAR_BIT** is the number of bits in a char object
  - **INT_MAX** is the maximum value of an object of type int
- The number of bytes making up an object of a certain type *t* can be determined using the *built-in function* sizeof(*t*):

```
m = sizeof(long long);    // equivalent to m = 8;
```

Expression Revisited

- Introduction
- Variable initialization
- Operator arity
- Precedence rules
- Association rules

Introduction

- An expression specifies a sequence of operations to be applied to various operands to generate some value.
- The process of working out the value of an expression is called **evaluation**.
- Operators may take 1, 2, 3 or a variable number of operands.
- There are rules governing the order in which operators are evaluated: precedence rules and association rules.

Variable Initialization

- Recalled that we can define symbolic (named) constants by prefixing an variable declaration with the keyword const:

  const int MAX_ENROLLMENT = 100;

- Variables are ***not*** initialized in its declaration and its value is **undefined**.
- Variables can be initialized in a manner similar to symbolic constants except that their values can be changed later in the program.

Variable Declaration Again

- A **variable declaration** is of the form:

  <var_decl> → [const] <type> <var_list> ‘;’

  <var_list> → <variable> { ‘,’ <variable> }

  <variable> → <identifier> [<init>]

  <init> → ‘=‘ <expression> | ‘(‘ <expression> ‘)’
- For example:

```
int num_enrollment = 10;
int n = 10, m(5), k;       // initializes n to 10, m to 5
                           // k is not initialized (undefined)
```

- The second form of variable initialization is C++ only.

Operator Arity

- Operator arity refers to the number of operands that an operator requires.
  - **unary** operators require only 1 operand

    +k   // identity

    -k   // negation

    ++k   // pre-increment: k = k + 1 and *yield* the updated k

    k++   // post-increment: k = k + 1 and yield the original k

    --k   // pre-decrement: k = k – 1 and yield the updated k

    k--   // post-decrement: k = k – 1 and yield the original k
  - **binary** operators require 2 operands
  - **ternary** operators require 3 operands

Binary Operators

- Arithmetic:

  m + n   // addition

  m – n   // subtraction

  m * n   // multiplication

  m / n   // division, n ≠ 0

  m % n   // remainder, n ≠ 0

- Note that remainder always has the same sign as the dividend (represented by m), for example:

  -17 % 3 evaluates to -2

  17 % -3 evaluates to 2

Relational Operators

- Compares a pair of integers and returns either true or false:

```
m ==  n   // true if m equals n
m !=  n   // true if m and n are not equal
m >= n    // true if m is greater than or equal to n
m <= n    // true if m is less than or equal to n
m > n     // true if m is greater than n
m < n     // true if m is less than n
```

Ternary Operators

- The **conditional operator** takes 3 operands:

  <expression> ? <expression> : <expression>
  - the first expression is evaluated
  - if the result is true, the second expression is evaluated and the result is returned as the result of the conditional
  - If the result is false, the third expression is evaluated and the result is returned as the result of the conditional
  - *note that the second and the third expression must yield (evaluate to) the same type of value*
- For example, the sign function can be re-written as:

```
int sign (int n) {
   return (n > 0) ? 1 : (n < 0) ? -1 : 0;
}
```

Precedence Rules

- Specify the priority among different operators:

high priority ↑ k++, k--

unary +, *unary* –, ++k, --k

*, /, %

+, –

>=, <=, >, <

==, !=

?:

low priority =

- For example:

a + b * - c is equivalent to a + (b * (-c))

Association Rules

- Specify the execution order among operators of the same priority:

right-to-left	*unary* +, *unary* –, ++k, k++, --k, k--
left-to-right	*, /, %
left-to-right	+, –
left-to-right	==, !=, >=, <=, >, <
right-to-left	?:
right-to-left	=

- For example:

  a = b = c is equivalent to a = (b = c)

  3 – 2 – 1 is equivalent to (3 – 2) – 1, ***not*** 3 – (2 – 1)

Using Parentheses

- In general, parentheses can be used to override the order determined by the precedence and the association rules and to make the order of evaluation explicit.
- For example, the following formula

$$\frac{u-v}{x \times y} \times \frac{-b}{p / q}$$

can be written as:

((u – v) / (x * y)) * (-b / (p / q))

Say No to Complicated Expressions!

- What is the result of the following expression?

  a = (b = 10) + (++b)

- How about the following one?

  a = (++b) + (b = 10)

- They are supposed to be the same since + is *commutative*, are they?
- How about the following pair?

      a = (b = 10) + (c = ++b)

  and

      a = (c = ++b) + (b = 0)

- ***When a variable may be updated in an expression, do not use that variable again in the same expression***.

Unsigned Integers

- There are *signed* and *unsigned* versions of int and the default is signed.
- Unsigned integers are useful for *counting* as they provide a larger range of integers, using *n* bits :

    between 0 and $2^n – 1$

  instead of

    between $-2^{n–1}$ and $2^{n–1} – 1$

- Unsigned integers are declared by prefixing the type name with the keyword unsigned, for example:

```
unsigned long m;        // m is an unsigned 32-bit integer
unsigned char k(255);   // k is an unsigned 8-bit integer,
                        // it is initialized to 255
```

Reals

- Introduction
- Floating-point numbers
- Scientific notation
- Real number representations
- IEEE floating-point standard

Introduction

- Mathematically, there is an infinite number of real numbers and integers form a proper subset of real numbers.
- Examples of real values in daily life:

  height, weight, speed, distance, interest rate, ...
- Examples of real numbers:

  123.456, 10, 2/3, $\sqrt{2}$, π, e
- Real numbers are approximated by floating-point numbers in digital computers.

Floating-Point Numbers

- The set of **floating-point numbers** is a subset of real numbers. A floating-point number consists of an **integer part** and a **fractional part**, both may consist of a ***variable*** number of digits.
- Examples of floating-point numbers:

number	*integer part*	*fractional part*
12.0	12	0
171.003	171	003
-7.43	-7	43
-0.625	-0	625

- *There is an infinite number of real numbers between any two floating-point numbers*.

Scientific Notation

- A floating-point number can be represented in the **scientific notation** where a value is represented as:

  fraction $\times$ 10^{exponent}, where $1 \leq |\text{ fraction }| \leq 10$

  – note that **fraction** is also known as **mantissa** or **coefficient**.

- For example:

number	*in scientific notation*
120.0	1.2×10^{2}
-0.0010004	-1.0004×10^{-3}
-1213141516	-1.213141516×10^{9}
0.0	0.0×10^{0}

Real Number Representations

- Only a subset of floating-point numbers is represented in computers. We need to fix the **precision** (number of digits in fraction) and the range of values for exponent.
- A real number is approximated by a nearby representable floating-point number; for example:

real number	*floating-point representation*
0.6666666666666666...	$6.6666666667 \times 10^{-1}$
π (3.1415926535897...)	$3.1415926536 \times 10^{0}$
$\sqrt{2}$ (1.41421356237...)	$1.4142135624 \times 10^{0}$

- Note that floating-point operations are very often vendor specific, but most also support the IEEE 754 Standard.

IEEE Floating-Point Standard

- The IEEE standard for floating-point arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronic Engineers (IEEE). The standard defines:
  - *arithmetic formats*: sets of binary and decimal floating-point data, which consist of:
    - finite numbers, including signed zeros and denormal numbers
    - infinities and special *not a number* values (NaN)
  - *interchange formats*: encodings that may be used to exchange floating-point data in an efficient and compact form
  - *rounding rules*: properties to be satisfied when rounding numbers during arithmetic operations and conversions
  - *operations*: arithmetic and other operations on arithmetic formats
  - *exception handling*: indications of exceptional conditions such as division by zero, overflow, etc.
  - additional operations such as trigonometric functions, among others

Computing with Floating-Point Numbers

- The data type double
- The cmath library
- Rounding errors
- More output formatting

The Data Type double

- Introduction
- Special constants
- Double constants
- Double operations
- Casting again

Introduction

- doubles are represented in scientific notation except that the base is 2 instead of 10.
- Each double occupies 8 bytes (64 bits):
  - 1 bit for the sign of the value
  - 52 bits for the fraction
  - 11 bits for the exponent
- Limits on double values are defined in the <cfloat> library, for example:
  - **DBL_MIN**: minimum double value
  - **DBL_MAX**: maximum double value

Special Constants

- An **overflow** occurs if the magnitude of the result of an operation is larger than the maximum after rounding. If the result is positive, it is set to Inf (+∞) and a negative one is set to –Inf (–∞). For example:

  DBL_MAX * 2.0 yields Inf

  -1.0 / 0.0 yields –Inf
- An **underflow** occurs if the result is less than 2^{-1074}, zero is typically returned as result.
- There is also a special NaN (not a number) that is used to represent *indefinite.* For example:

  sqrt(–1.0) yields NaN

  1.0 / 0.0 also yields NaN

Denormal Floating-Point Numbers

- Floating-point numbers are typically represented with a nonzero leading digit in the fraction. If the fraction is in base 2, ***all*** fractions will have a leading 1. Hence, a **normal** floating-point number will not actually have a leading 1 in its fraction!
- For numbers between $2^{-1022 \text{ and}}$ 2^{-1074}, the exponent is too small; we can fix the exponent at 2^{-1022} but now the fraction will not have the full 52 bits of precision and we can no longer assume a leading 1 bit. For example:

  1.11×2^{-1025} will be represented as 0.00111×2^{-1022}
- These numbers are called **denormal** numbers and the exponent is set to zero.
- Zeroes are actually denormal numbers as both the fraction and the exponent are zero (except for the sign).
- Inf, NaN are all represented by denormal numbers.

Double Constants

- Double constants are of the form:

  <fp_const> → [‘+’ | ‘-’] <fixed> [< exponent>]

  <fixed> → <digits> ‘.’ [<digits>] | ‘.’ <digits>

  <exponent> → (‘e’ | ‘E’) [‘+’ | ‘-’] <digits>

  <digits> → <digit> { <digit> }

- Examples of double constants:

  123.456, 5., 0.0, .3, +9.0, .123456666666779

  1.09e-3, -1E10, 0.003e5, 135.47E-12, 1.e-10

- Examples of bad double constants:

  123, 1d3, .e-2, 2.0e0.5

Double Operations

- Unary operations:

  +x // identity

  -x // negation

- Binary operations:

  x + y // addition

  x – y // subtraction

  x * y // multiplication

  x / y // division, $y \neq 0.0$

- The precedence and associations are the same as those of int.
- *How about relational operators?*

Casting Again

- If an int is supplied as an operand to a double operation, it will be coerced into a double.
- Conversion can be made explicit by using the casting (double).
- For example:

```
d = 3;            // has the same effect as the following
d = (double)3;
```

- Note that **truncation** (fractional part removed) occurs when converting a floating-point value to integer:

```
i = 2.7;          // i gets 2, and a warning from the compiler
i = (int)2.7;     // works just fine
```

The cmath Library

- This library provides functions for computing many common mathematical functions, for example: *arctan*(*x*), *exp*(*x*), *fabs*(*x*), *log*(*x*), *pow*(*x*), *sin*(*x*), *sqrt*(*x*), …
- To look up the description of a function I <cmath>:

1. start from http://www.cplusplus.com/ref
2. click cmath
3. click the function that you want to look up

Rounding Errors

- **Rounding errors** are induced by the *inexactness* of the floating-point number representation for real numbers.
- Rounding errors may be accumulated and magnified during a series of computations:

nature of error	*expression*	*error*
inexactness	$1/3 \cong 0.333$	0.000333…
accumulation	$1/3+1/3 \cong 0.666$	0.000666…
magnifying	$100*(1/3-0.33) \cong 0.3$	0.0333…

Effects of Limited Precision - Example

- Assuming only 4 digits of precision, consider the following system of simultaneous equations:

$$\begin{cases} 3x + 4.127y = 15.41 \\ x + 1.374y = 5.147 \end{cases}$$

- Divide the first equation by 3, then eliminate *x* by subtracting the results from the second equation. Solving the resulting equation for *y*, we get:

$$y = \frac{5.147 - 15.41/3}{1.374 - 4.127/3} = \frac{5.147 - 5.137}{1.374 - 1.376} = \frac{0.01}{-0.002} = -5$$

- The correct answer for *y* is actually -6.2

A Programming Example

- Consider the following identity:

$$\frac{x}{x-\sin x}-\frac{\sin x}{x-\sin x}=\frac{x-\sin x}{x-\sin x}=1$$

- The following program "checks" the above identity:

```
double x, y;
int      k = -30;
while (k++ < 0) {  // k starts from -29
   x = pow(2.0, (double)k);  // x = 2^k
   y = x – sin(x);
   cout << x << "  " << x/y–sin(x)/y << endl;
}
```

output:

```
1.86265e-9  47.5
3.72529e-9  -4.96875
. . .
(26 lines omitted)
. . .
0.5  1
1  1
```

More Output Formatting (1)

- In the first line of output from the previous example, the first double is output in **scientific notation** whereas the second is in **fixed-point notation**.
- The default format, fixed-point or scientific, is usually determined by the magnitude of an output value. (As the program does not specify any formatting requirements.)
- To specify an output notation for double values, write

```
cout.setf(ios::fixed);        // output set to fixed-point notation
```

or

```
cout.setf(ios::scientific);  // output set to scientific notation
```

- To clear specific formatting flag, for example,

```
cout.unsetf(ios::fixed);
```

More Output Formatting (2)

- To set the width for output:

```
cout.precision(<int value>);
```

  - for fixed-point notation, it specifies the number of digits after the decimal point
  - for scientific notation, it specifies the number of significant digits
- To insists showing the decimal point and the trailing zeros, write

```
cout.set(ios::showpoint);
```