CSE / ENGR 142 Programming I

Arithmetic Expressions

Assignment Statement Review

```
double area, radius;

area = 3.14 * radius * radius;
```

assignment statement — expression

Why Study Expressions?

1. We need precise rules that define exactly what an expression means:

   What is the value of 4 - 4 * 4 + 4?

2. Arithmetic on a computer isn't always precise:

   (1.0 / 9.0) * 9.0 could be 0.99999998213

3. Division of "int" type variables can give REALLY different results from what you probably expect:

   2 / 3 is zero in C

Expressions

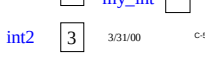
- Expressions are things that have values
  - A variable by itself is an expression: radius
  - A constant by itself is an expression: 3.14
- Often expressions are combinations of variables, constants, and operators.

  *area = 3.14 * radius * radius;*
- The overall value of the expression is based on the data and operators specified.
  - Data means the integer or floating-point constants and/or variables in the expression.
  - Operators are things like addition, multiplication, etc.

The Big Picture

- In an assignment statement,
  - the expression (right hand side) is first evaluated,
  - then its value is assigned to (stored in) the assignment variable (left hand side).
- How this happens depends on the data types in the expression, the operators, and the type of the assignment variable.

```
my_int = int1 + int2;
```



Unary and Binary

- Binary: operates on two operands
  - *3.0 * b*
  - *zebra + giraffe*
- Unary: operates on one operand
  - *-23.4*
- C operators are unary or binary
- Then what about expressions like *a+b+c*?
  - Answer: this is two binary ops, in sequence

Expressions with *doubles*

REVIEW:

Doubles are floating-point values that represent real numbers within the computer.

Constants of type double:

0.0, 3.14, -2.1, 5.0, 6.02e23, 1.0e-3

not 0 or 17

Operators on doubles:

unary: -

binary: +, -, *, /

Expressions with *doubles*: Examples

double height, base, radius, x, c1, c2 ;

Sample expressions (not statements):

*0.5 * height * base*

*(4.0 / 3.0) * 3.14 * radius * radius * radius*

*- 3.0 + c1 * x - c2 * x * x*

Expressions with *ints*

REVIEW:

An integer represents a whole number with no fractional part.

Constants of type *int*:

0, 1, -17, 42 not 0.0 or 1e3

Operators on *ints*:

unary: -

binary: +, -, *, /, %

int division and remainder

Integer operators include *integer division* and *integer remainder*.

Caution: looks like an old topic, but it's new!

```
  2
100 )299
  200
   99
```

/ is integer division: no remainder, no rounding

299 / 100 → 2 , 6 / 4 → 1 , 5 / 6 → 0

% is mod or remainder:

299 % 100 → 99 , 6 % 4 → 2 , 5 % 6 → 5

Expressions with *ints*: Time Example

Given: total_minutes 359

Find: hours 5
minutes 59

Solution:

hours = total_minutes / 60 ;

minutes = total_minutes % 60 ;

A Cautionary Example

```
int radius;
double area;
.
.
.
area = ( 22 / 7 ) * radius * radius;
```

Why Use *int*s? Why Not *double*s Always?

- Sometimes only *int*s make sense
  - "give me the 15th spreadsheet cell"
  - "give me the (14.9999998387)th cell" ??
- *Double*s may be inaccurate representing "ints"
  - In mathematics 3 • 15 • (1/3) = 15
  - In computer arithmetic
    3.0 * 15.0 * (1.0 / 3.0) might be 14.999999997
- Last, *and least*
  - arithmetic with *double*s is slower on some computers
  - *double*s often require more memory

Operator Precedence

Precedence determines the order of evaluation of operators.

Is `a + b * a - b` equal to `( a + b ) * ( a - b )` or `a + ( b * a ) - b` ??

And does it matter?

Try this:

`4 + 3 * 2 - 1`

`(4 + 3) * (2 - 1) =`

`4 + (3 * 2) - 1 =`

Operator Precedence

Precedence rules:

1. do ( )'s first, starting with innermost
2. then do unary minus (negation): -
3. then do "multiplicative" ops: *, /, %
4. lastly do "additive" ops: binary +, -

Associativity

Associativity determines the order among consecutive operators of equal precedence

Is `a / b * c` equal to `a / ( b * c )` or `( a / b ) * c` ??

Most C arithmetic operators are "left associative", within the same precedence level

`a / b * c` equals `(a / b) * c`

`a + b - c + d` equals `( ( a + b ) - c ) + d`

C also has a few operators that are right associative.

The Full Story...

- C has about 50 operators & 18 precedence levels…
- A "Precedence Table" shows all the operators, their precedence and associativity.
  - Look on inside front cover of our textbook
  - Look in any C reference manual
- When in doubt: check the table
- When faced with an unknown operator: check the table

Precedence and Associativity: Example

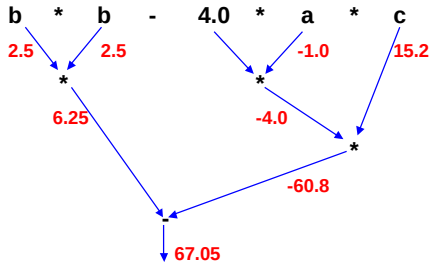
Mathematical formula:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

C formula:

`(- b + sqrt ( b * b - 4.0 * a * c) ) / ( 2.0 * a )`

Expressions & Values



Mixed Type Expressions

What is *2 * 3.14* ?

Compiler will implicitly (automatically) convert *int* to *double* when they occur together:

int + *double* → *double* + *double* (likewise -, *, /)

2*3 * 3.14 → (2*3) * 3.14 → 6 * 3.14 → 6.0 * 3.14 → 18.84

2/3 * 3.14 → (2/3) * 3.14 → 0 * 3.14 → 0.0 * 3.14 → 0.0

We strongly recommend you avoid mixed types:
e.g., use 2.0 / 3.0 * 3.14 instead.

Conversions in Assignments

```
int total, count, value;
double avg;
total = 97 ;    count = 10 ;
avg = total / count ;          /*avg is 9.0*/

value = avg*2.2;          /*BAD (why?)*/
```

implicit conversion to double

Explicit Conversions

(Section 7.1)

- To be explicit in the program, you can use a cast
  - convert the result of an expression to a different type.
- Format: *(type) expression*
- Examples:
  (double) myage
  (int) (balance + deposit)
- This does not change the rules for evaluating the expression (types, etc.)

Using Casts

```
int total, count ;
double avg;
total = 97 ;    count = 10 ;
avg = total / count ;                    /*avg is 9.0*/

avg = (double) total / (double) count ;  /*avg is 9.7*/
avg = (double) (total / count) ;         /*avg is 9.0*/
```

implicit conversion to double

explicit conversion to double

C is "Strongly Typed"

- Every variable, value, and expression has a type
- C cares a lot about what the type of each thing is
- Lots of cases where types have to match up
- Start now: be constantly aware of the type of everything in your programs!

Basic Lessons

- Write in the clearest way possible for the reader.
- Keep it simple; for very complex expressions, break them up into multiple statements.
- Use parentheses to indicate your desired precedence for operators where it may be ambiguous.
- Use explicit casts to avoid implicit conversions in mixed mode expressions and assignments.
- Be aware of types.