CSE 142
Computer Programming I

Arithmetic Expressions

Overview

Types
- Numbers aren't number
- Conversions and casts
- Mixed Type Expressions

Arithmetic expressions
- Precedence
- Associativity

Functions

Unary and binary operators

Symbolic constants

Reading: Text sec. 2.5.

Numbers Aren't Numbers

Remember:
C distinguishes integers (int's) from real numbers (double's)

```
double   total
double   count;   <-- This is backwards from what you'd probably actually do
int      average;
/* Initialization */
total = 97.0 ;
count = 10.0;
/* Some assignment statements */
average = total / count;
```

warning C4244: '=' : conversion from 'double ' to 'int ', possible loss of data

Implicit vs Explicit Conversion

C ***implicitly converts*** between int's and double's

You should ***explicitly convert***, to show you mean it.

```
double   total
double   count;
int      average;
/* Initialization */
total = 97.0 ;
count = 10.0;
/* Some assignment statements */
average = (int)(total / count);
```

~~warning C4244: '=' : conversion from 'double' to 'int ', possible loss of data~~

This is called an ***explicit type cast***

Why Was Explicit Conversion Required?

```
average = total / count;
double  = int / int;
```

Evaluate

Assign

C has two different division operators, both written '/'

- int / int → integer division
- double / double → real-number division
- Int / double → ?
- double / int → ?

Mixed Type Expressions

- int / double → ?
- double / int → ?

If either *operand* is double, then the operator is double. An implicit conversion of the other operand takes place.

```
double   total
int      count;
double   average;
/* Initialization */
total = 97.0 ;
count = 10;
/* Some assignment statements */
average = total / count;
```

average = total / count; —*implicitly*→ average = total / (double)count;

C implicitly converts an int to a double for all mixed type operations: +, -, *, /

Mixed Type Expressions

```
------    average;
------    total
int       count;

/* Initialization */
total = 97 ;
count = 10;
average = total / count ;
```

Type of *average*	Type of *total*	Equivalent Explicit Cast	Expression Value	Final Value of *average*
double	int	average = (double)(total / count);	9	9.0
int	int	average = total / count;	9	9
double	double	average = total/(double)count;	9.7	9.7
int	double	average = (int)(total/(double)count);	9.7	9

D-7

Explicit Casts

You should perform explicit casts, within reason

At a minimum, no

warning C4244: '=' : conversion from 'double ' to 'int ', possible loss of data

D-8

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

Sometimes only *int*'s make sense:

The 15th spreadsheet cell, not the 14.997th cell

Double's may be inaccurate:

In mathematics 3 • 15 • (1/3) = 15

But, 3.0 * 15.0 * (1.0 / 3.0) might be 14.9999997

(Of course, in C 3*15*(1/3) is 0!)

D-9

Why Study Expressions?

10 + 8 * 6 - 3 = ?

10 + 8*6 - 3 =	**55**
10+8 * 6-3 =	**54**
10 + 8 * 6 - 3 =	**105**

D-10

Removing Ambiguity: Parentheses

You can always make it completely clear what you mean using parentheses:

((10+(8*6))-3) =	55
((10+8)*(6-3)) =	54
(((10+8)*6)–3) =	105

- Easy for a computer to understand.
- Not so easy for a human to understand.
- Not so easy for a human to type -> lots of errors.

D-11

Removing Ambiguity: Operator Precedence

Precedence indicates "how urgent" an operator is.

Given a choice, higher precedence operators are evaluated first.

*** and / are higher precedence than + and –**

(…) is higher precedence than everything else

So 10 + 8 * 6 - 3 is equivalent to 10+(8*6)-3 = 55, but (10+8)*6-3 is 105

D-12

Operator Precedence Examples

Higher ↑ ()

*, /

Lower +, -

32 – 8 * 3 is 8
12 + 4 / 2 is 14
22 / 7 + 3 * 4 is 15
2 + 32 / 4 * 2 is 18
2 + 32 / (4 * 2) is 6

D-13

Precedence Isn't Enough

Precedence doesn't help if all the operators have the same precedence

*Is a / b * c* equal to

*a / (b * c)* or *(a / b) * c* ??

Associativity determines the order among consecutive operators of equal precedence

Does it matter? Try this: 15 / 4 * 2

D-14

Associativity Rules

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.)

D-15

Other Operators: Unary Minus

Binary operator: operates on two operands

*3.0 * b*
zebra + giraffe

Unary operator: operates on one operand

-23.4

Unary minus applies to int's and double's, to literals and to variables

-zebra

Unary minus has precedence *higher* than * and /

D-16

Other Operators: mod

/ is integer division: <u>no</u> remainder, <u>no</u> rounding

299 / 100 → 2
6 / 4 → 1
5 / 6 → 0

% is mod or remainder:

299 % 100 → 99
6 % 4 → 2
5 % 6 → 5

mod has precedence *equal* to * and /

D-17

Expressions with *mod*: Time Example

Given: totalMinutes 359
Find: hours (5)
minutes (59)

Solution in C:

```
hours   = totalMinutes / 60 ;
minutes = totalMinutes % 60 ;
```

D-18

The Full Story...

C has about 50 operators & 18 precedence levels…

A "Precedence Table" shows all the operators, their precedence and associativities.

- Look on inside front cover of our textbook
- Look in any C reference manual

When in doubt: check the table

When in doubt: use parentheses

D-19

Advice on Writing Expressions

Write in the clearest way possible to help the reader

Keep it simple; break very complex expressions into multiple assignment statements

Use parentheses to indicate your desired precedence for operators when it is not clear

Use explicit casts to avoid (hidden) implicit conversions in mixed mode expressions and assignments

Be aware of types

D-20

Other “Operators”: Functions

C includes functions for additional calculations that are not available using operators like +, -, *, /, etc.

```
rootOfTwo = sqrt(2.0);

x = 2.1 * sin(theta/1.5) + 17.0;

eightyOne = pow(3.0, 4.0);
```

Functions can be used in expressions just like constants or variables, with two *caveats…*

D-21

Functions and #include

To use these (particular) functions, you have to tell the compiler where to find out about them:

```
#include <math.h>
int main(void) {
    …
    result = sqrt(2.0) / 10;
    …
```

← Description (but not code) of sqrt(), pow(), etc.

The #include line tells the compiler what it needs to know at compile time: what is the type of the value provided by the sqrt() function?

D-22

Functions and Libraries

The #include line tells the compiler about the functions.

The linker needs to find the machine code (.obj file) for the functions.

(Standard) C functions are actually organized into *libraries.*

The development environment (e.g., MSVC or CodeWarrior) (usually) knows how to find these libraries (but if it doesn’t, you will get a linker error).

D-23

Symbolic Constants

Generally speaking, a value that “could change” should not be written as a literal in your program.

Example: Suppose you’re writing a program to compute the number of doughnuts and coffee to order for each meeting of CSE142. An average person eats 1.3 doughnuts and drinks 5 ounces of coffee. There are 252 students in the class.

You should not write:

```
totalDoughnuts = 252 * 1.3;
totalCoffeeInOunces = 252 * 5;
```

Why?

What should you do?

D-24

What Should You Do?

```
#include <stdio.h>
int main(void)
{
    int      NUMBER_OF_STUDENTS = 252;
    double   DOUGHNUTS_PER_STUDENT = 1.3;
    double   COFFEE_PER_STUDENT = 5.0;
    …
    totalDoughnuts = NUMBER_OF_STUDENTS * DOUGHNUTS_PER_STUDENT;
    totalCoffeeInOunces = NUMBER_OF_STUDENTS * COFFEE_PER_STUDENT;
}
```

Notes:
- Initialization takes place on the declaration line.
- Names in all caps indicates "This is a constant – I shouldn't be assigning to it after the declaration."
- These are just "conventions" – C doesn't make you follow these rules; they help the (human) reader.

D-25

Why?

Centralize changes:

If you later want to change the value, you have to edit exactly one line (not hundreds)

No "magic numbers":

A reader looking at your code sees the logical idea of what you're doing, not numbers that could be anything (and don't matter to understanding the correctness of the program)

Reduce the chance that you have a bug due to a mistyped constant value

D-26

Literal Constants in Your Code

Some constants will likely appear in your code, but only in special circumstances.

I've written a program that grades exams, and counts the number of exams it has finished so far:

totalGraded = totalGraded + 1;

I wouldn't define a symbolic constant for the literal 1:

- The value of 1 is never going to change
- The logical intent of what I'm doing (incrementing by 1) is clearer with the literal than by creating a symbolic constant named ONE

0 is another commonly appearing literal. Almost nothing else is. (Sometimes –1, sometimes 2, but it's rare.)

D-27

PENNIES_PER_DOLLAR?

Style Wars

- *How to implement* symbolic constants is a matter of style.
- *Whether or not to use them* is *not* a matter of contention.
- The "usual convention" in C is to use a mechanism called "#define". (The 9:30 class is doing that.)
- We're moving into the mid-1990's with this convention, which has some clear advantages.
- We'll talk about #define later, when we're able to understand better what it is and what it isn't, and deal with the mayhem it has a tendency to cause.

D-28

Next Time

We'll discuss input and output

See you then!

D-29