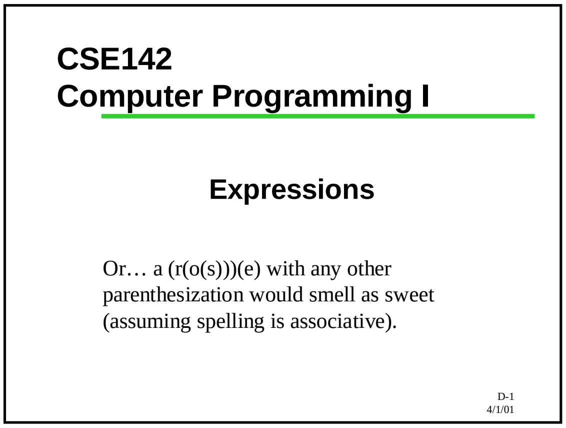
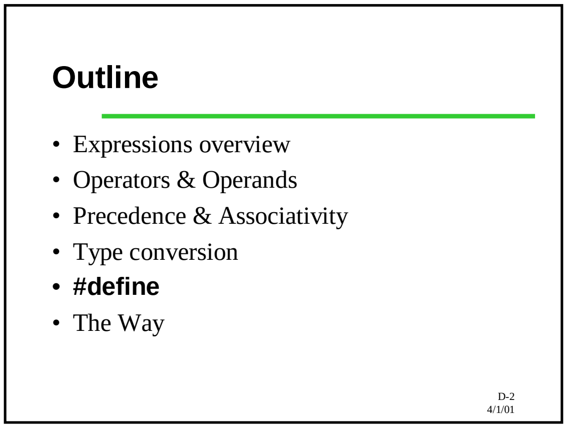


CSE142
Computer Programming I

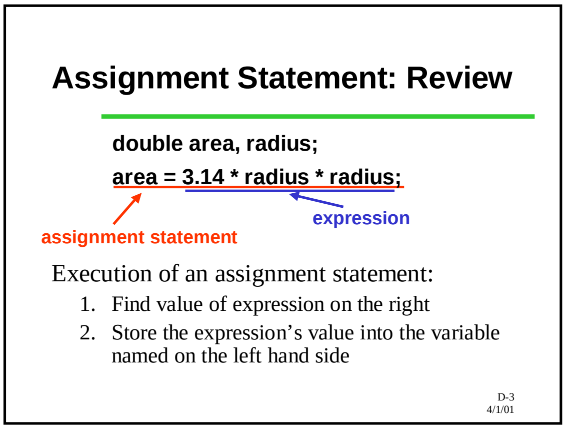
Expressions

Or… a (r(o(s)))(e) with any other parenthesization would smell as sweet (assuming spelling is associative).



Outline

- Expressions overview
- Operators & Operands
- Precedence & Associativity
- Type conversion
- **#define**
- The Way



Assignment Statement: Review

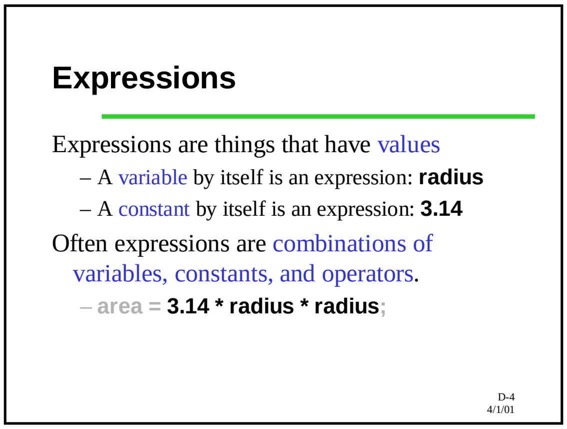
double area, radius;

area = 3.14 * radius * radius;

assignment statement
expression

Execution of an assignment statement:

1. Find value of expression on the right
2. Store the expression's value into the variable named on the left hand side



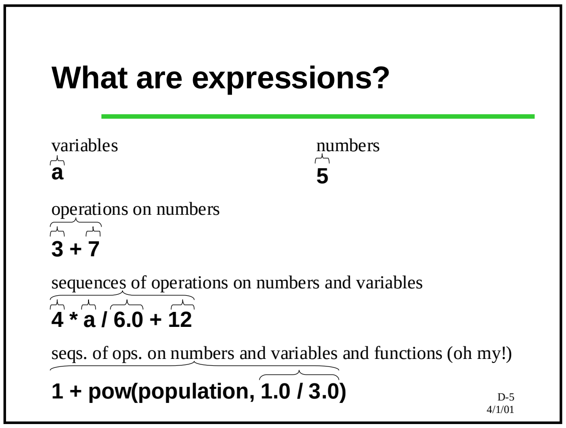
Expressions

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**;



What are expressions?

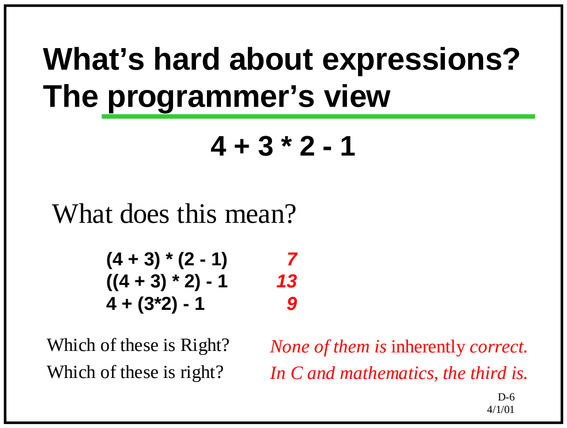
variables
a

numbers
5

operations on numbers
3 + 7

sequences of operations on numbers and variables
4 * a / 6.0 + 12

seqs. of ops. on numbers and variables and functions (oh my!)
1 + pow(population, 1.0 / 3.0)



What's hard about expressions? The programmer's view

4 + 3 * 2 - 1

What does this mean?

(4 + 3) * (2 - 1)	*7*
((4 + 3) * 2) - 1	*13*
4 + (3*2) - 1	*9*

Which of these is Right? *None of them is inherently correct.*

Which of these is right? *In C and mathematics, the third is.*

What’s hard about expressions? The computer’s view

result = 4 + 3 * 2 – 1;

How *must* we say this to the computer?

```
result = 3 * 2;
result = 4 + result;
result = result – 1;
```

The computer does all its calculations and operations on a pair of numbers (or just one).

Expression Evaluation

Some terminology:

- Operators are things like addition and multiplication.
- Operands (or data) are the things the operators work on:
  variables, real and integer constants, etc.
- The value of an expression will depend on the data types, the values, and the operators used.
- Additionally, the final result of an assignment statement will depend on the *type of the assignment variable*.

Arithmetic Types: Review

C provides two kinds of numeric values

- Integers (**0**, **12**, **-17**, **142**)
  - Type int
  - Values are exact
  - Constants have no decimal point or exponent
- Floating-point numbers (**3.14**, **-6.023e23**)
  - Type double
  - Values are approximate (~12-14 digits precision)
  - Constants must have decimal point and/or exponent

Operator Jargon

- Binary: operates on two operands

  3.0 * b   zebra + giraffe
- Unary: operates on one operand

  -23.4
- C operators are unary or binary
- Puzzle: what about expressions like a+b+c?

This expression has two binary operators, executed one after the other: **(a+b)+c**

Expressions with doubles

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: **+ - * /**

Note: there’s no exponentiation operator in C!

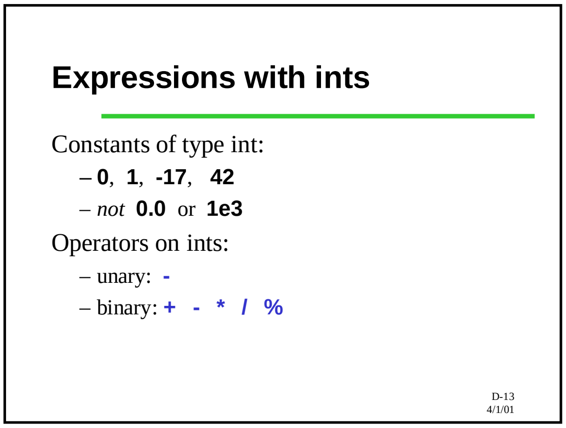
Some Expressions w/Doubles

Declarations:

```
double height = 10.0, base = 2.5;
double radius = 0.2;
double x = 2.0, coeff1 = 8.0, coeff2 = 0.0;
```

Sample expressions (not statements):

```
0.5 * height * base
( 4.0 / 3.0 ) * 3.14 * radius * radius * radius
- 3.0 + coeff1 * x - coeff2 * x * x
```



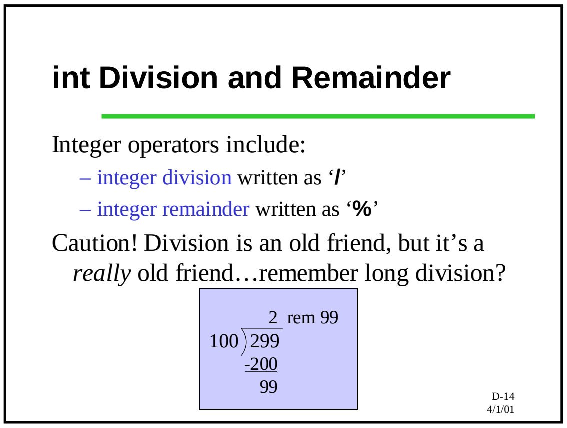
Expressions with ints

Constants of type int:

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

Operators on ints:

- unary: **-**
- binary: **+ - * / %**



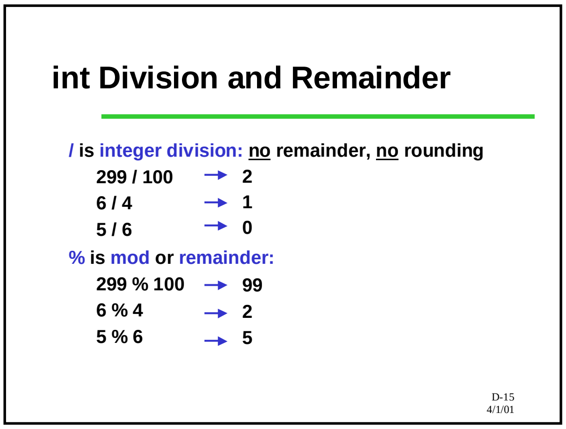
int Division and Remainder

Integer operators include:

- integer division written as '**/**'
- integer remainder written as '**%**'

Caution! Division is an old friend, but it's a *really* old friend…remember long division?

```
       2  rem 99
100 ) 299
     -200
       99
```



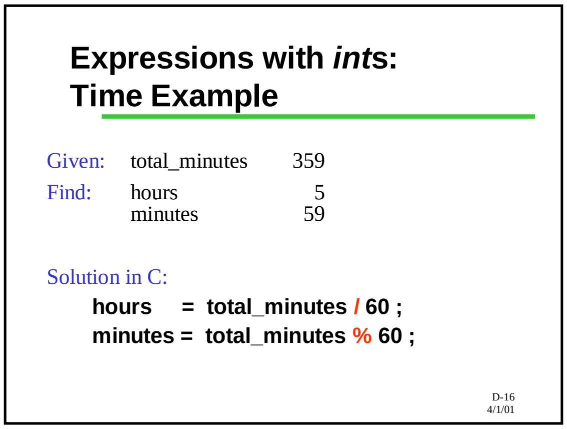
int Division and Remainder

/ 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

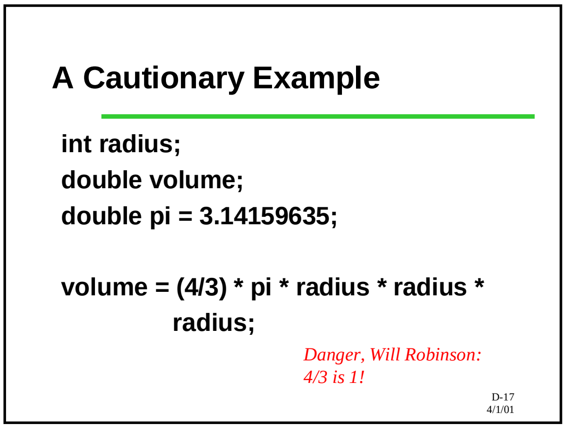


Expressions with *ints*: Time Example

Given:	total_minutes	359
Find:	hours	5
	minutes	59

Solution in C:

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

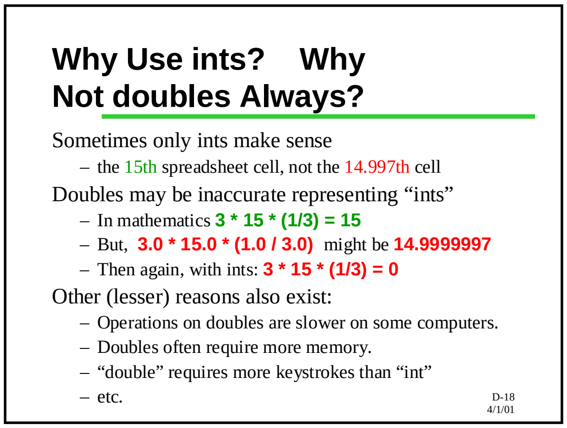


A Cautionary Example

```
int radius;
double volume;
double pi = 3.14159635;

volume = (4/3) * pi * radius * radius *
            radius;
```

Danger, Will Robinson:
4/3 is 1!



Why Use ints? Why Not doubles Always?

Sometimes only ints make sense

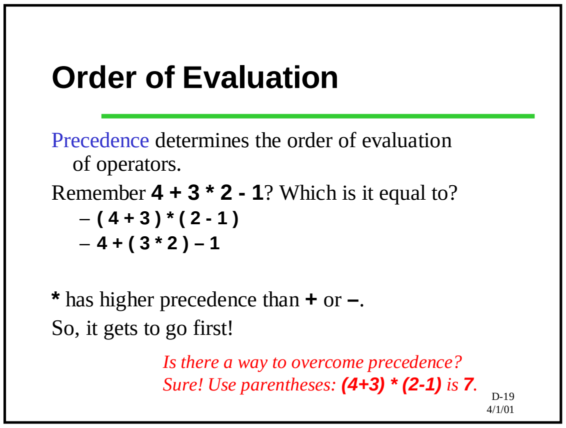
- the 15th spreadsheet cell, not the 14.997th cell

Doubles may be inaccurate representing "ints"

- In mathematics **3 * 15 * (1/3) = 15**
- But, **3.0 * 15.0 * (1.0 / 3.0)** might be **14.9999997**
- Then again, with ints: **3 * 15 * (1/3) = 0**

Other (lesser) reasons also exist:

- Operations on doubles are slower on some computers.
- Doubles often require more memory.
- "double" requires more keystrokes than "int"
- etc.



Order of Evaluation

Precedence determines the order of evaluation of operators.

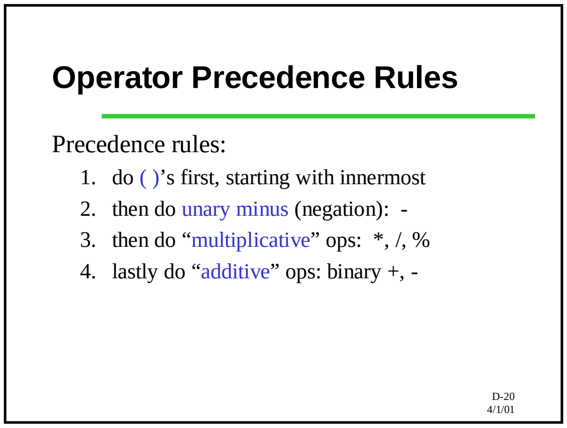
Remember **4 + 3 * 2 - 1**? Which is it equal to?

- **(4 + 3) * (2 - 1)**
- **4 + (3 * 2) – 1**

***** has higher precedence than **+** or **–**.

So, it gets to go first!

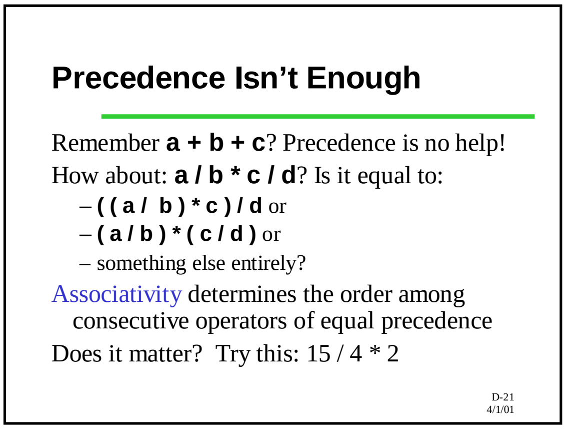
Is there a way to overcome precedence?
Sure! Use parentheses: ***(4+3) * (2-1)*** *is* ***7***.



Operator Precedence Rules

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 +, -



Precedence Isn't Enough

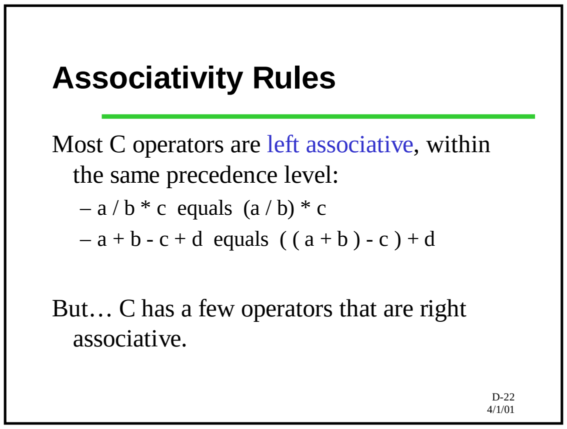
Remember **a + b + c**? Precedence is no help!

How about: **a / b * c / d**? Is it equal to:

- **((a / b) * c) / d** or
- **(a / b) * (c / d)** or
- something else entirely?

Associativity determines the order among consecutive operators of equal precedence

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

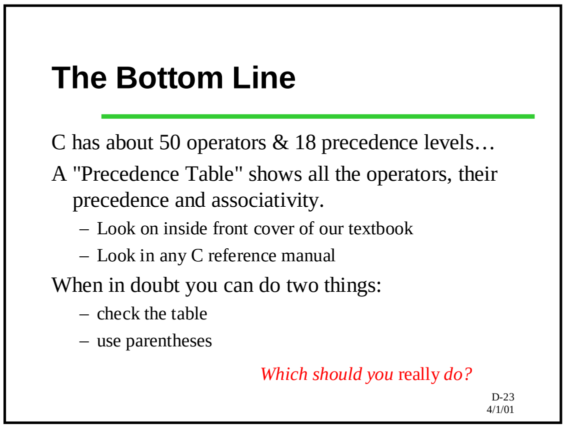


Associativity Rules

Most C 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

But… C has a few operators that are right associative.



The Bottom Line

C has about 50 operators & 18 precedence levels…

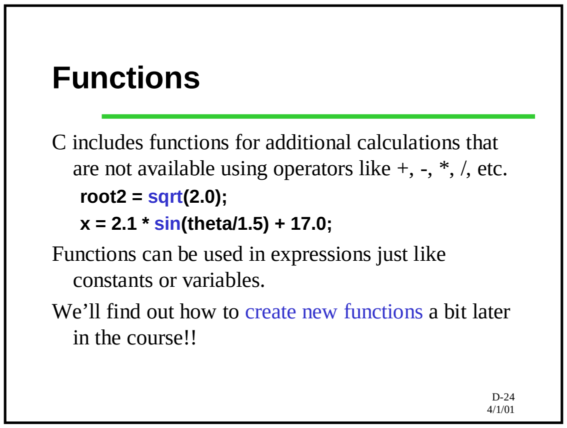
A "Precedence Table" shows all the operators, their precedence and associativity.

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

When in doubt you can do two things:

- check the table
- use parentheses

Which should you really *do?*



Functions

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

root2 = sqrt(2.0);
x = 2.1 * sin(theta/1.5) + 17.0;

Functions can be used in expressions just like constants or variables.

We'll find out how to create new functions a bit later in the course!!

Function Libraries - #include

Standard C functions are organized into libraries.

To use a library function, specify the library that contains it (using **#include**) at the top of the program.

Look in the textbook (appendix C) or a C manual for lists of available libraries and functions.

```
#include <math.h>
int main(void) {
   …
   root2 = sqrt(2.0);
   …
```

The <math.h> library contains sqrt, sin, cos, tan, etc.

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)

But this is bad… why?

Precedence and Associativity: Example

Mathematical formula:

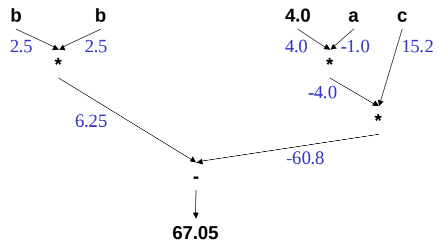
$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

C statements:

```
discriminant = b*b – 4.0 * a * c;
root = (-b + sqrt(discriminant)) / (2.0 * a);
```

Depicting Expressions

b = 2.5;
a = -1.0;
c = 15.2;



Choose Your Own Adventure

2 * 3.14

What happens when an integer meets a double?
You decide…

If you choose "int multiplication", go forward one slide.

If you choose "double multiplication", go forward two slides.

If you choose "syntax error", go forward three slides.

Otherwise, go forward four slides.

int Multiplication

2 * 3.14

Heading north, you realize that you've lost something important to you. It's your **.14**! What happened to it?

If we try to use integer multiplication, we'll have to make 3.14 an integer. When we do that, we *lose* data!

double Multiplication

2 * 3.14

You feel a change coming over you. You're the same… but different somehow! What's happened?

If we try to use double multiplication, we need to change the 2 into a double. What does it become? Will this work?

Syntax Error

2 * 3.14

You try to head north into the forest, but a mysterious force grabs you and hurtles you backward, saying:

"`adv.c(87): error C47: non-standard adventure detected`"

This *could* have been made a syntax error. But, it wasn't. That's a *design choice*.

"Else"

2 * 3.14

Frozen with indecision, you pause for one fateful moment.

In that time, a passel of *subexpressions* swarm over you and *evaluate* you repeatedly. Distracted, you don't notice the *assignment statement* lurking behind. Before you notice its presence, it has already *set* you.

You spend the rest of your life as "**6.28**".

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 = total*2.2;       /* bad news */
```

implicit conversion to double

implicit conversion to int: drops fraction with no warning

Explicit Conversions

Use a cast to explicitly convert the result of an expression to a *different* type

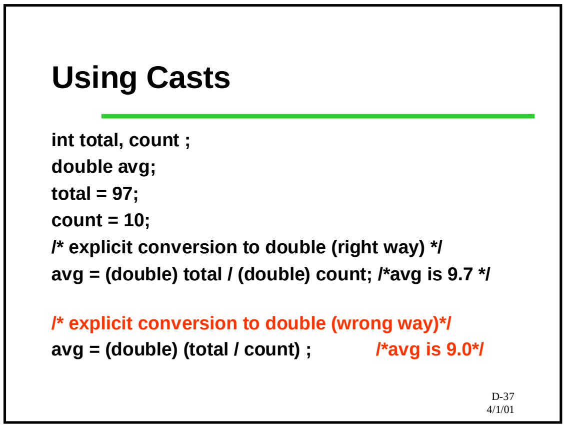
Format: **(type) expression**

Examples **(double) myage**
(int) (balance + deposit)

This *does not* change the rules for evaluating the expression itself (types, etc.)

The Way: It is good style to cast even if the conversion would happen anyway.

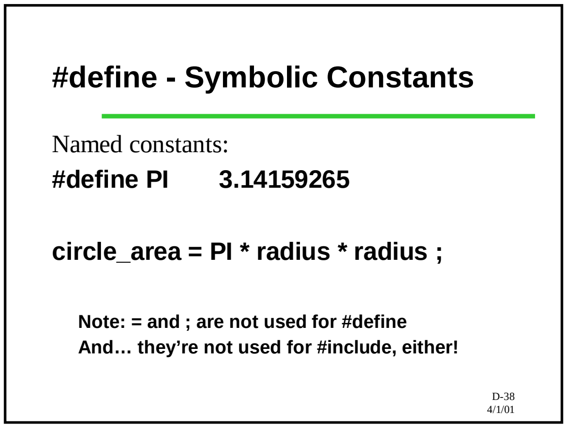
Why?



Using Casts

```
int total, count ;
double avg;
total = 97;
count = 10;
/* explicit conversion to double (right way) */
avg = (double) total / (double) count; /*avg is 9.7 */

/* explicit conversion to double (wrong way)*/
avg = (double) (total / count) ;          /*avg is 9.0*/
```



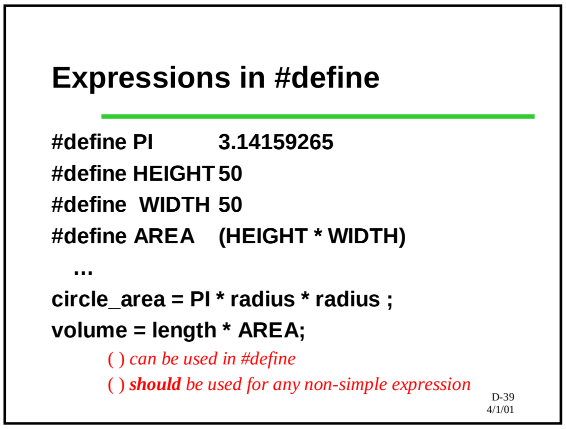
#define - Symbolic Constants

Named constants:

```
#define PI      3.14159265

circle_area = PI * radius * radius ;
```

Note: = and ; are not used for #define
And… they're not used for #include, either!

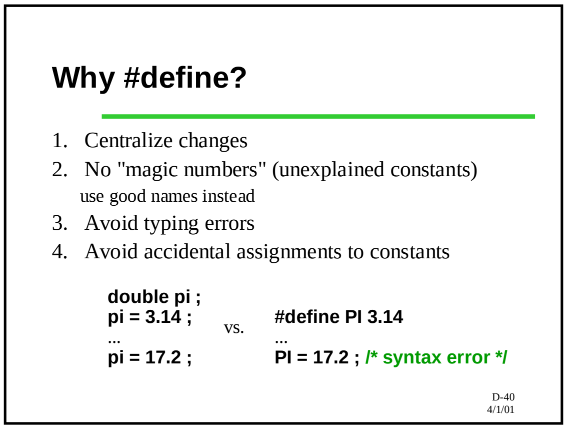


Expressions in #define

```
#define PI          3.14159265
#define HEIGHT 50
#define  WIDTH  50
#define AREA     (HEIGHT * WIDTH)
   …
circle_area = PI * radius * radius ;
volume = length * AREA;
```

( ) can be used in #define

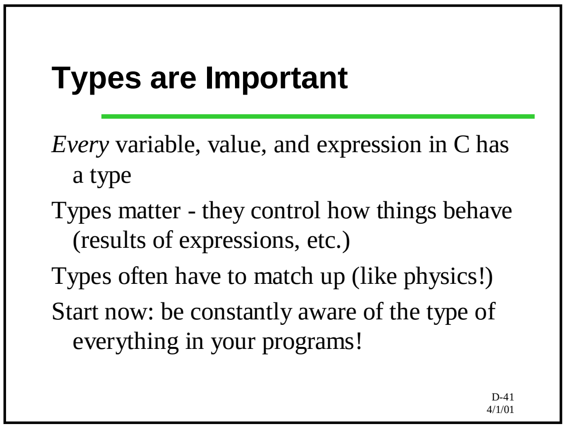
*( ) **should** be used for any non-simple expression*



Why #define?

1. Centralize changes
2. No "magic numbers" (unexplained constants)
   use good names instead
3. Avoid typing errors
4. Avoid accidental assignments to constants

```
double pi ;                #define PI 3.14
pi = 3.14 ;        vs.
...                        ...
pi = 17.2 ;                PI = 17.2 ; /* syntax error */
```



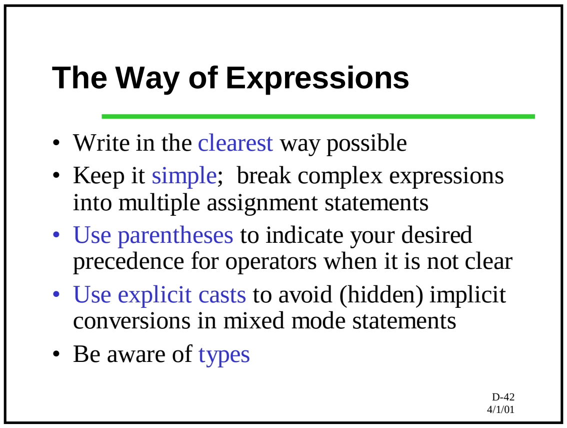
Types are Important

Every variable, value, and expression in C has a type

Types matter - they control how things behave (results of expressions, etc.)

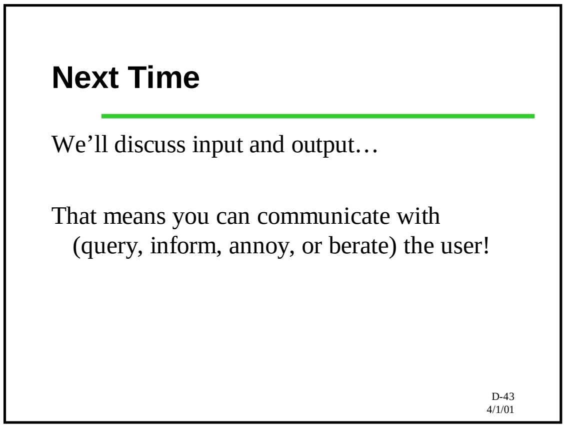
Types often have to match up (like physics!)

Start now: be constantly aware of the type of everything in your programs!



The Way of Expressions

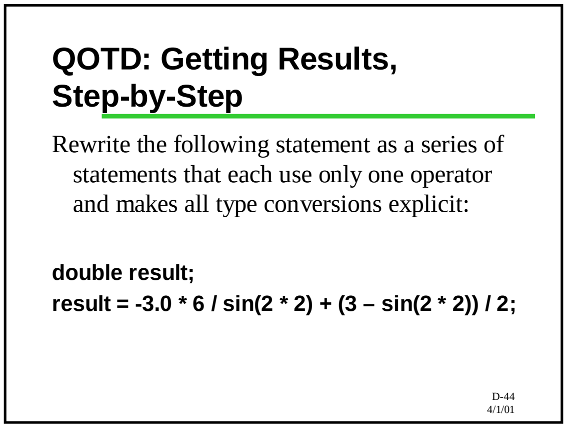
- Write in the clearest way possible
- Keep it simple; break complex expressions into multiple assignment statements
- Use parentheses to indicate your desired precedence for operators when it is not clear
- Use explicit casts to avoid (hidden) implicit conversions in mixed mode statements
- Be aware of types



Next Time

We'll discuss input and output…

That means you can communicate with (query, inform, annoy, or berate) the user!



QOTD: Getting Results, Step-by-Step

Rewrite the following statement as a series of statements that each use only one operator and makes all type conversions explicit:

```
double result;
result = -3.0 * 6 / sin(2 * 2) + (3 – sin(2 * 2)) / 2;
```