CSE103
Structured Programming
Lecture-3

Dr. Maheen Islam
Associate Professor
Dept. of CSE
East West University

Reserved Words and Identifiers

- Reserved word
  - Word that has a specific meaning in C
    - Ex: int, return
- Identifier
  - Word used to name and refer to a data element
    or object manipulated by the program.

Valid Identifier Names

- Begins with a letter or underscore symbol
- Consists of letters, digits, or underscores only
- Cannot be a C reserved word
- Case sensitive

  □Total ≠ total ≠ TOTAL
- Examples:

```
distance
milesPerHour
_voltage
goodChoice
high_level
MIN_RATE
```

Invalid Identifier Names

- Does not begin with a letter or underscore symbol or
- Contains other than letters, digits, and underscore or
- Is a C reserved word
- Examples

```
x-ray
2ndGrade
$amount
two&four
after five
return
```

Identifier Name Conventions

- Standard practice, not required by C language
  - Normally lower case
  - Constants upper case
- Multi-word
  - Underscore between words or
  - Camel case - each word after first is capitalized

```
distance
TAX_RATE
miles_per_hour
milesPerHour
```

CONSTANT

Variabl
e

- Name is a valid identifier name
- Is a memory location where a value can be stored for use by a program
- Value can change during program execution
- Can hold only one value
  - Whenever a new value is placed into a variable, the new value replaces the previous value.

Variables Names

- C: Must be a valid identifier name
- C: Variables must be declared with a name and a data type
  before they can be used in a program
- Should not be the name of a standard function or variable
- Should be descriptive; the name should be reflective of the variable's use in the program
  - For class, make that must be descriptive except subscripts
- Abbreviations should be commonly understood
  - Ex. `amt = amount`

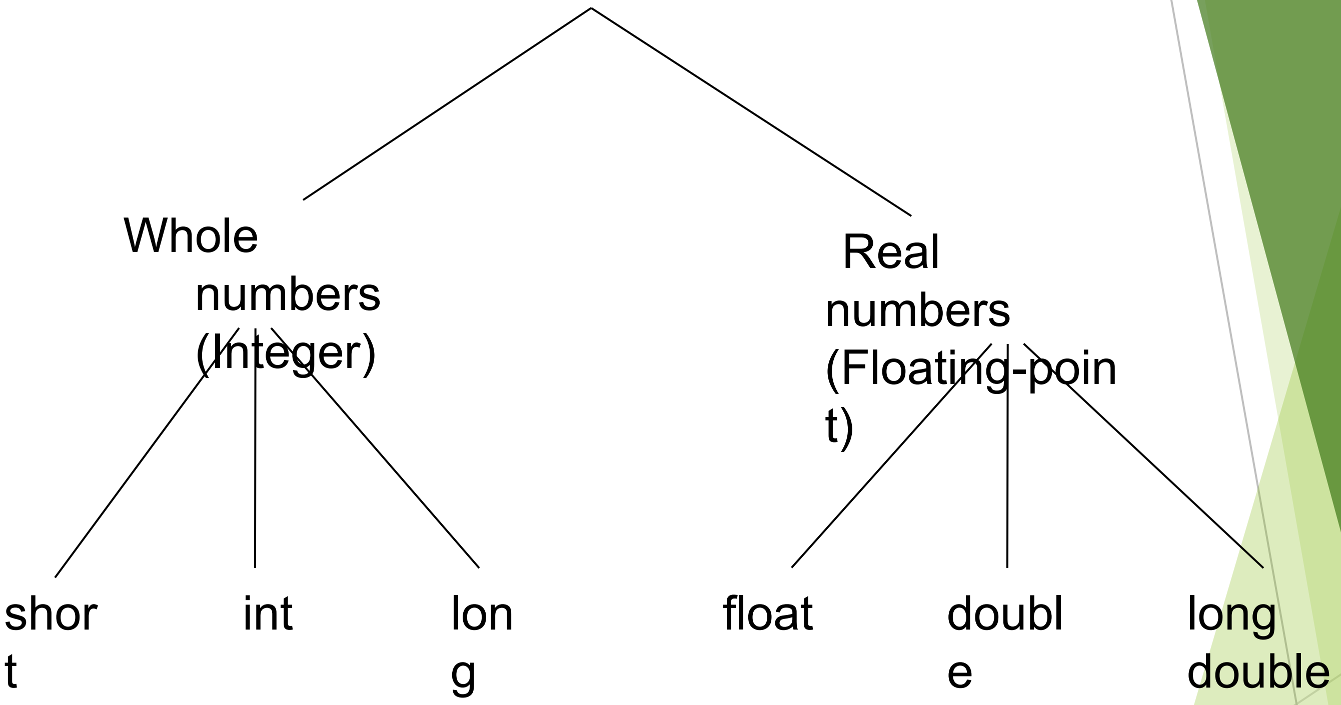
Variable/Named Constant Declaration Syntax

```
optional_modifier data_type  name_list;
```

- *optional_modifier* – type modifier
  - Used to distinguish between **signed** and **unsigned** integers
    - The default is signed
  - Used to specify size (**short**, **long**)
  - Used to specify named constant with **const** keyword
- *data_type* - specifies the type of value; allows the compiler to know what operations are valid and how to represent a particular value in memory
- *name_list* – program identifier names
- Examples:

```
int test-score;
const float TAX_RATE = 6.5;
```

Numeric Data Types



Whole
numbers
(Integer)

Real
numbers
(Floating-poin
t)

shor
t

int

lon
g

float

doubl
e

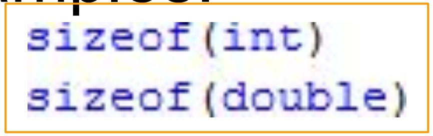
long
double

Data Types and Typical Sizes

Type Name	Memory Used	Size Range	Precision	Guarantee
short (= short int)	2 bytes	-32,768 to 32,767	N/A	16 bits
int	4 bytes	-2,147,483,648 to 2,147,483,647	N/A	16 bits
long (= long int)	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	N/A	32 bits
float	4 bytes	approximately 10^{-38} to 10^{38}	7 digits	6 digits
double	8 bytes	approximately 10^{-308} to 10^{308}	15 digits	10 digits
long double	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits	10 digits

Determining Data Type Size

- sizeof operator
  - Returns size of operand in bytes
  - Operand can be a data type
- Examples:

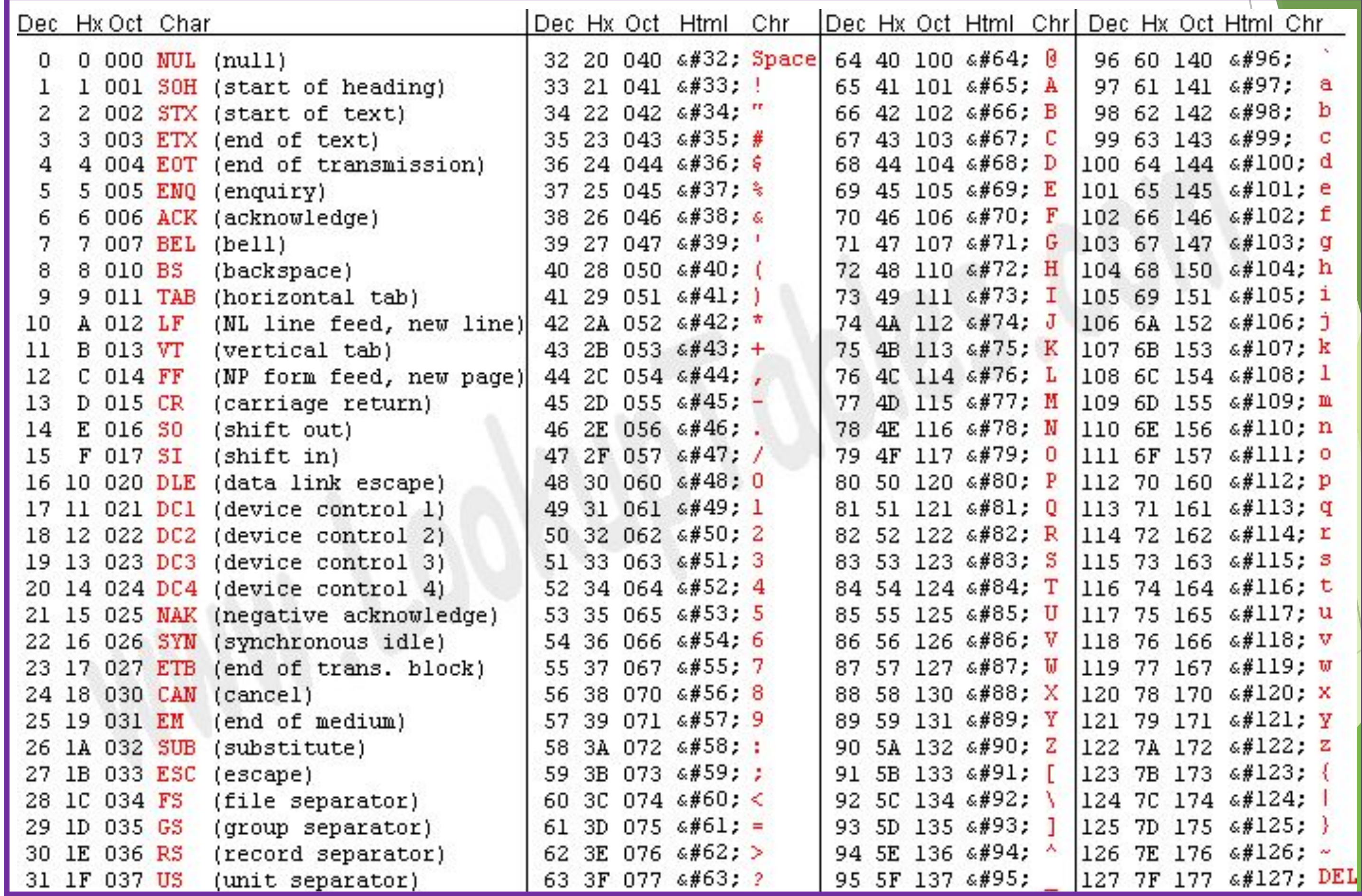


```
sizeof(int)
sizeof(double)
```

Characters

Type Name	Memory Used	Sample Size Range
char	1 byte	All ASCII characters

ASCII = American Standard Code for Information Interchange



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

www.LookupTables.com

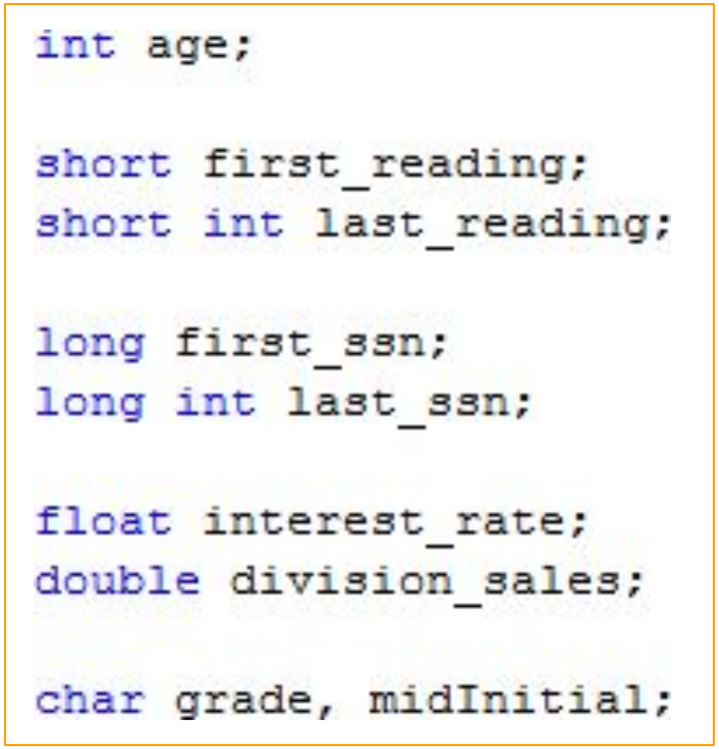
www.asciitable.com

Boolean Data Type

- Data type: _Bool
  - Can only store 0 & 1
  - Non zero value will be stored as 1
- Data type : bool
  - <stdbool.h> defines `bool`, `true`, and `false`

- Any expression
  - 0 is false
  - Non-zero is true

Variable Declaration Examples



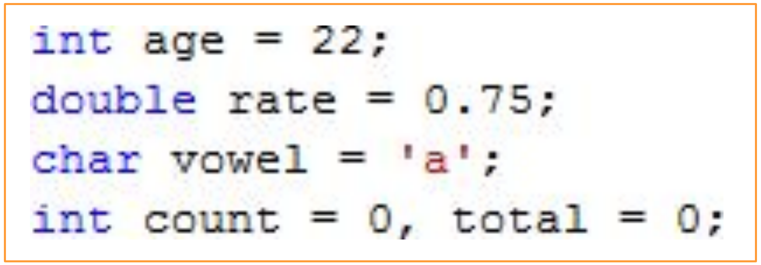
```
int age;

short first_reading;
short int last_reading;

long first_ssn;
long int last_ssn;

float interest_rate;
double division_sales;

char grade, midInitial;
```

Assigning Values to Variables

- Allocated variables without initialization have an undefined value.
- We will use three methods
  for assigning a value to a variable
  - Initial value
    - In the declaration statement
  - Processing
    - the assignment statement
  - Input
    - scanf function

Initializing Variables

- Initializing variables in declaration statements



```
int age = 22;
double rate = 0.75;
char vowel = 'a';
int count = 0, total = 0;
```

Assignment Operator =

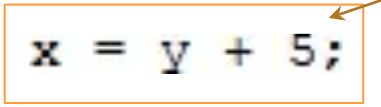
- Assigns a value to a variable
- Binary operator (has two operands)
- Not the same as "equal to" in mathematics
- General Form:

```
l_value = r_value
```

  - Most common examples of l_values (left-side)
    - A simple variable
    - A pointer dereference (in later chapters)
  - r_values (right side) can be any valid expression
- Assignment expression has value of assignment

Example Assignment Statement

- Statement



```
x = y + 5;
```

5 is literal value or constant

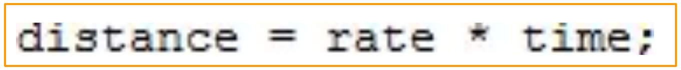
- Means:

  Evaluate the expression on the right and put the result in the memory location named x

- If the value stored in y is 18, then 23 will be stored in x

Other Example Assignments

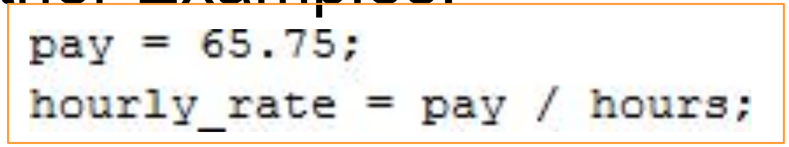
- Example:



```
distance = rate * time;
```

l_value: distance
r_value: rate *
time

- Other Examples:



```
pay = 65.75;
hourly_rate = pay / hours;
```

Terminal Output

Go Tigers!

What can be output?

- Any data can be output to standard output (stdout), the terminal display screen
  - Literal values
  - Variables
  - Constants
  - Expressions (which can include all of above)

- printf function:

  The values of the variables are passed to printf

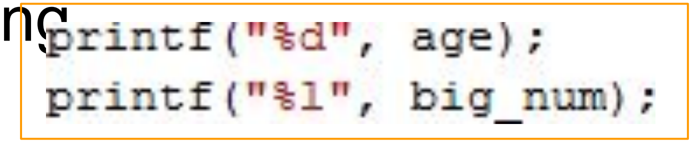
Syntax: printf function

```
printf(format_string, expression_list)
```

- Format_string specifies how expressions are to be printed
  - Contains placeholders for each expression
    - Placeholders begin with % and end with type
- Expression list is a list of zero or more expressions separated by commas
- Returns number of characters printed

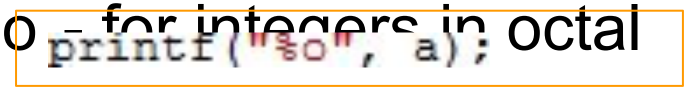
Typical Integer Placeholders

- %d or %i - for integers, %l for long



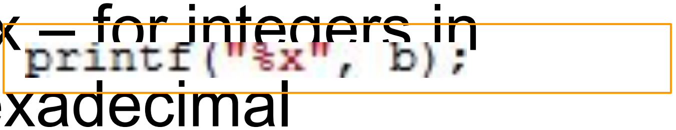
```
printf("%d", age);
printf("%l", big_num);
```

- %o – for integers in octal



```
printf("%o", a);
```

- %x – for integers in hexadecimal



```
printf("%x", b);
```

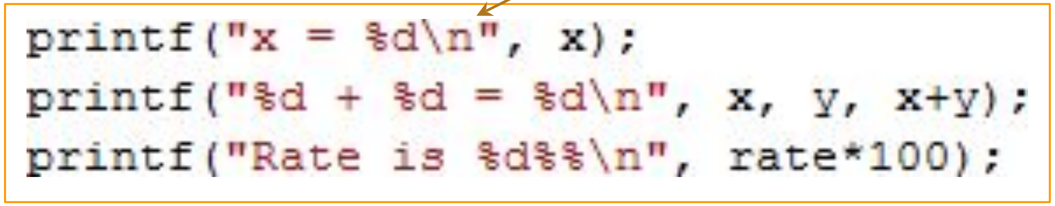
Floating-point Placeholders

- %f, %e, %g – for float
  - %f – displays value in a standard manner.
  - %e – displays value in scientific notation.
  - %g – causes printf to choose between %f and %e and to automatically remove trailing zeroes.
- %lf – for double (the letter l, not the number 1)

Printing the value of a variable

- We can also include literal values that will appear in the output.
  - Use two %’s to print a single percent

\n is new line



```
printf("x = %d\n", x);
printf("%d + %d = %d\n", x, y, x+y);
printf("Rate is %d%%\n", rate*100);
```

Output Formatting Placeholder

`%[flags][width][.precision][length]type`

- Flags
  - `-` left-justify
  - `+` generate a plus sign for positive values
  - `#` puts a leading 0 on an octal value and 0x on a hex value
  - `0` pad a number with leading zeros
- Width
  - Minimum number of characters to generate
- Precision
  - Float: Round to specified decimal places

Output Formatting Placeholder

`%[flags][width][.precision][length]type`

- Length

  l long

- Type

  d, i decimal unsigned

  int ffloat

  x hexadecimal

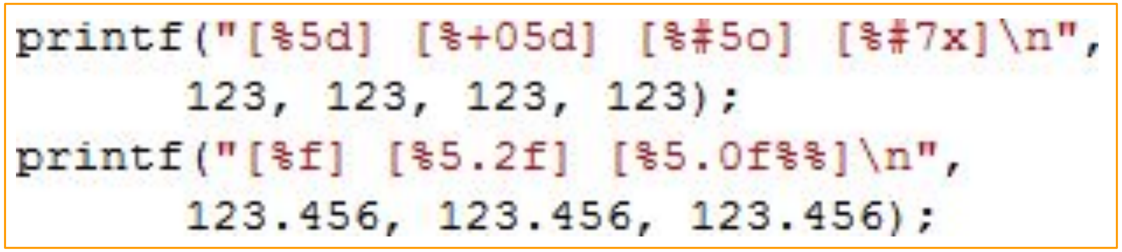
  o octal

  % print a %

Output Formatting Placeholder

%[flags][width][.precision][length]type

- Examples:



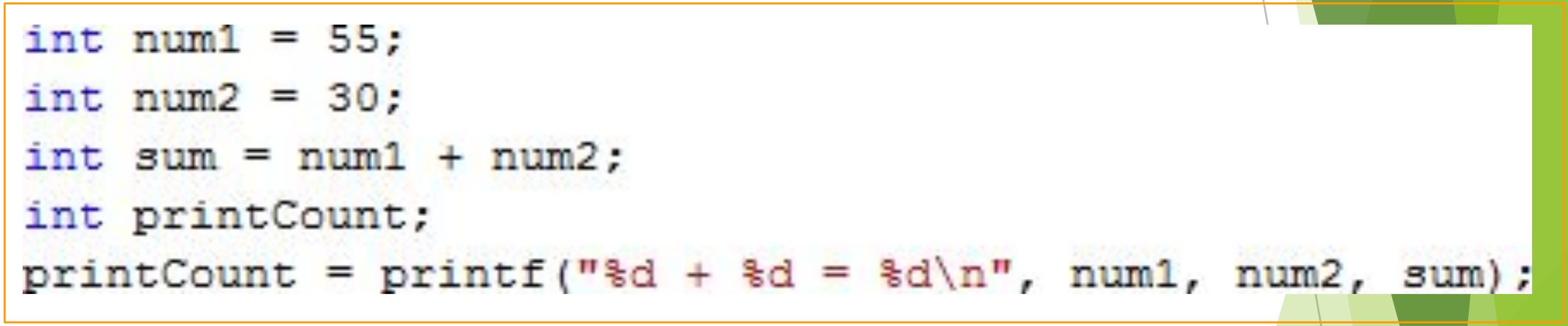
```
printf("[%5d] [%+05d] [%#5o] [%#7x]\n",
      123, 123, 123, 123);
printf("[%f] [%5.2f] [%5.0f%%]\n",
      123.456, 123.456, 123.456);
```

```
[  123] [+0123] [ 0173] [ 0x7b]
[123.456000] [123.46] [ 123%]
```

Format codes w/printf: http://en.wikipedia.org/wiki/Printf

Return from printf

- A successful completion of printf returns the number of characters printed. Consequently, for the following:



```
int num1 = 55;
int num2 = 30;
int sum = num1 + num2;
int printCount;
printCount = printf("%d + %d = %d\n", num1, num2, sum);
```

if printf() is successful,
the value in printCount should be 13.

Literals / Literal Constants

- Literal – a name for a specific value
- Literals are often called constants
- Literals do not change value

Literal

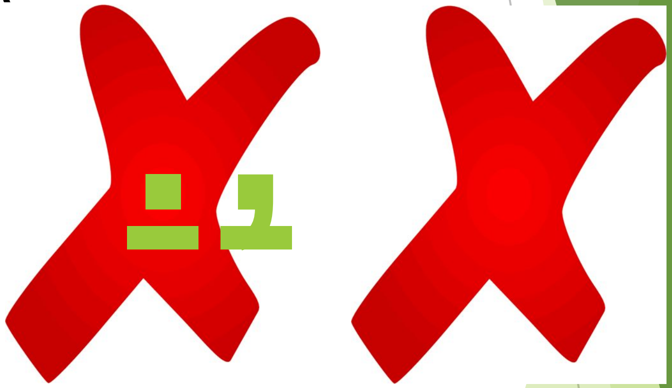
Integer Constants

- Must not contain a decimal point
- Must not contain a comma
- Examples

  -25

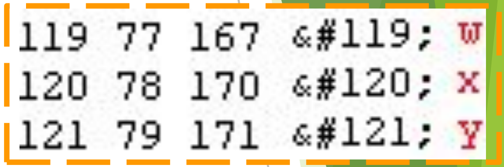
  68

  17895



Integer Constants

- May be expressed in several ways
  decimal number `120`
  hexadecimal `0x78`
  number octal `0170`
  number
  ASCII encoded character `'x'`

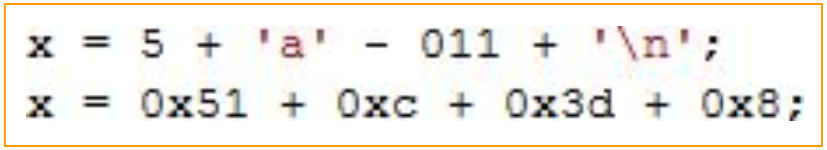


119	77	167	&#119;	w
120	78	170	&#120;	x
121	79	171	&#121;	y

- All of the above represent the 8-bit byte whose value is 01111000

Integer Constants

- Constants of different representations may be intermixed in expressions:
  - Examples



```
x = 5 + 'a' - 011 + '\n';
x = 0x51 + 0xc + 0x3d + 0x8;
```

Floating Point Constants

- Contain a decimal point.
- Must not contain a comma
- Can be expressed in two ways

decimal number:**23.8 4.0**

scientific notation: **1.25E10**

char Constants

- Enclosed in apostrophes, single quotes
- Example

  s: `'a'`

  `'A'`

  `'$'`

  `'2'`

  `'+'`
- Format specification: %c

String Constants

- Enclosed in quotes, double quotes
- Examples:

```
    "Hello"
    "The rain in Spain"
    "x"
```

- Format specification/placeholder: %s

Terminal Input

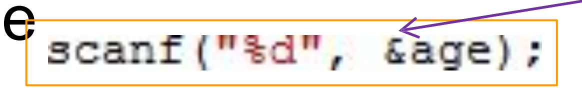
- We can put data into variables from the standard input device (stdin), the terminal keyboard
- When the computer gets data from the terminal, the user is said to be acting interactively.
- Putting data into variables from the standard input device is accomplished v the scanf function

Keyboard Input using scanf

- General format

  ```
  scanf(format-string, address-list)
  ```

- Exampl
  e

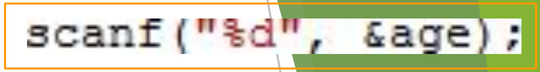


```
scanf("%d", &age);
```

& (address of operator) is

- The format string contains placeholders (one per address) to be used in converting the input.
  - □ %d – Tells *scanf* that the program is expecting an ASCII encoded integer number to be typed in, and that *scanf* should convert the string of ASCII characters to internal binary integer representation.
- Address-list: List of memory addresses to hold the input values

Addresses in scanf()



```
scanf("%d", &age);
```

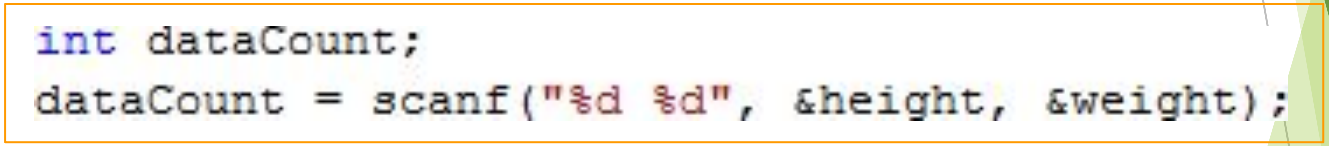
- Address-list must consist of addresses only
  - □ scanf() puts the value read into the memory address
  - □ The variable, age, is not an address; it refers to the *content* of the memory that was assigned to age
- & (address of) operator causes the address of the variable to be passed to *scanf* rather than the value in the variable
- Format string should consist of a placeholder for each address in the address-list

Format codes w/scanf: http://en.wikipedia.org/wiki/Scanf

Return from scanf()

- A successful completion of scanf() returns the number of input values read. Returns EOF if hits end-of-file reading one item.

  Consequently, we could have



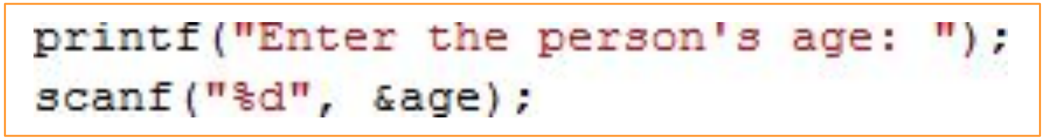
```
int dataCount;
dataCount = scanf("%d %d", &height, &weight);
```

  - If scanf() is successful,
    the value in dataCount should be 2

- Spaces or new lines separate one value from another

Keyboard Input using scanf

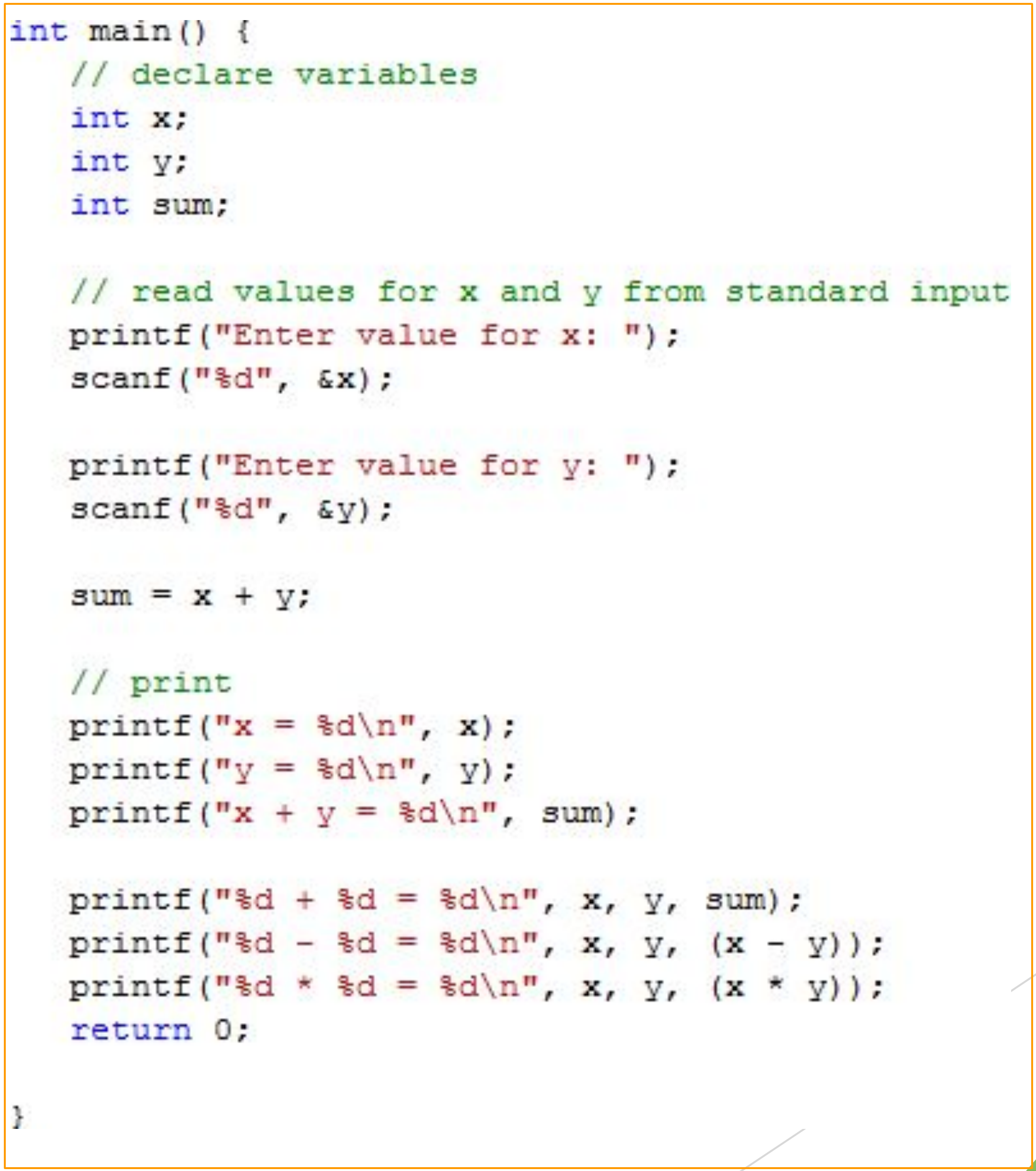
- When using scanf for the terminal, it is best to first issue a prompt



```
printf("Enter the person's age: ");
scanf("%d", &age);
```

- Waits for user input, then stores the input value in the memory space that was assigned to number.
- Note: ‘\n’ was omitted in printf
  - Prompt ‘waits’ on same line for keyboard input.
- Including printf prompt before scanf maximizes user-friendly input/output

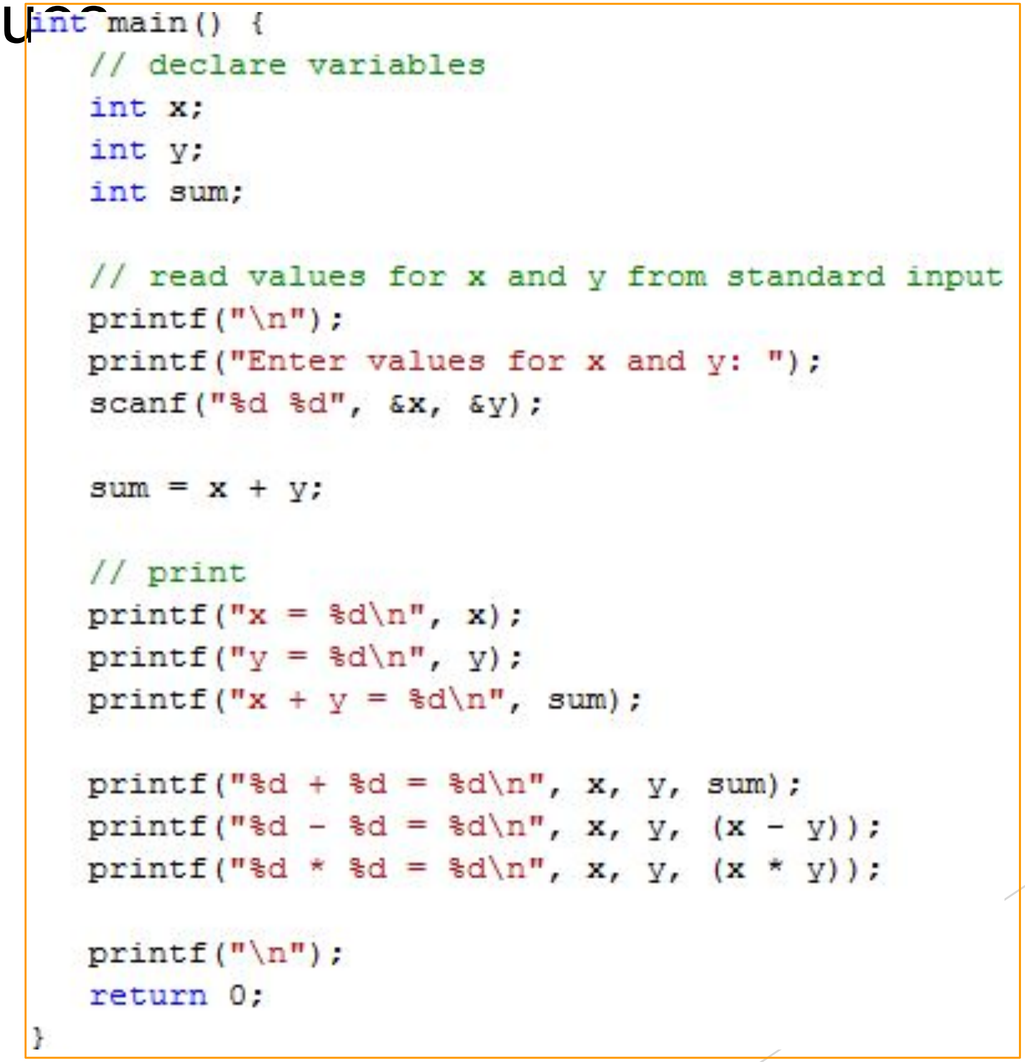
scanf Example



```
int main() {
   // declare variables
   int x;
   int y;
   int sum;

   // read values for x and y from standard input
   printf("Enter value for x: ");
   scanf("%d", &x);

   printf("Enter value for y: ");
   scanf("%d", &y);

   sum = x + y;

   // print
   printf("x = %d\n", x);
   printf("y = %d\n", y);
   printf("x + y = %d\n", sum);

   printf("%d + %d = %d\n", x, y, sum);
   printf("%d - %d = %d\n", x, y, (x - y));
   printf("%d * %d = %d\n", x, y, (x * y));
   return 0;

}
```

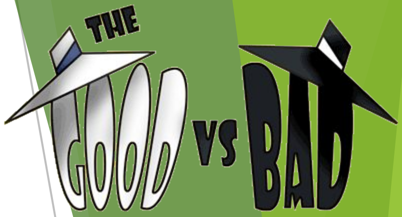
Input using scanf()

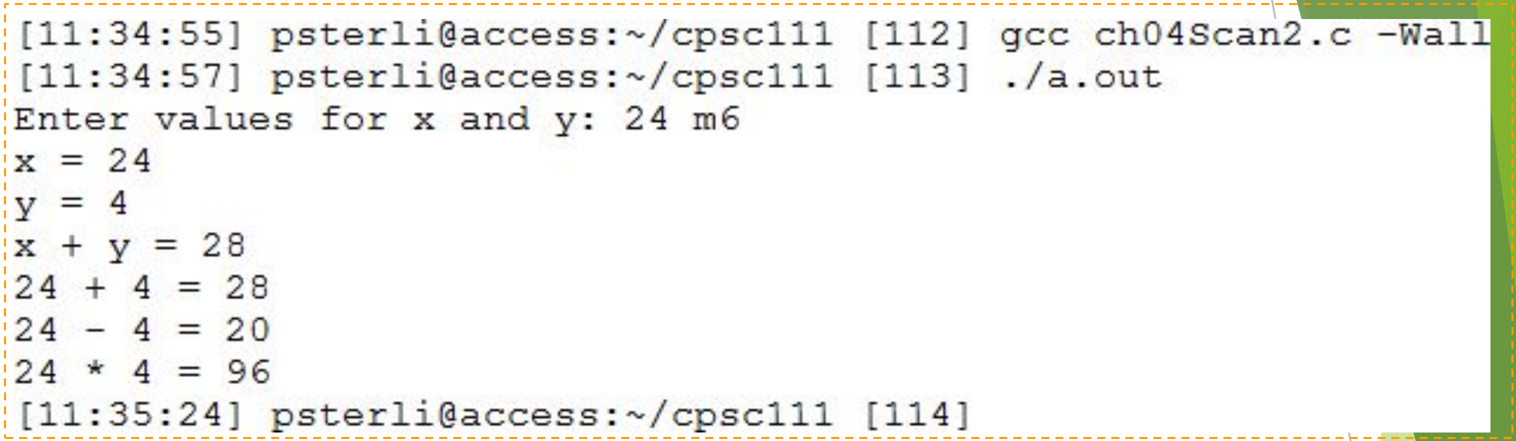
- Instead of using scanf() twice, we can use one scanf() to read both values



```c
int main() {
    // declare variables
    int x;
    int y;
    int sum;

    // read values for x and y from standard input
    printf("\n");
    printf("Enter values for x and y: ");
    scanf("%d %d", &x, &y);

    sum = x + y;

    // print
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("x + y = %d\n", sum);

    printf("%d + %d = %d\n", x, y, sum);
    printf("%d - %d = %d\n", x, y, (x - y));
    printf("%d * %d = %d\n", x, y, (x * y));

    printf("\n");
    return 0;
}
```

Bad Data





```
[11:34:55] psterli@access:~/cpsc111 [112] gcc ch04Scan2.c -Wall
[11:34:57] psterli@access:~/cpsc111 [113] ./a.out
Enter values for x and y: 24 m6
x = 24
y = 4
x + y = 28
24 + 4 = 28
24 - 4 = 20
24 * 4 = 96
[11:35:24] psterli@access:~/cpsc111 [114]
```

- scanf stops at the first bad character.
- The value of y was never set.   The value 4 is what was left in the memory location named num2 the last time the location was assigned a value.

Format Placeholder for Input

- When reading data, use the following format specifiers / placeholders

  %d - for integers, no octal or hexadecimal

  %i – for integers allowing octal and hexadecimal

  %f - for float

  %lf – for double (the letter l, not the number 1)

- Do not specify width and other special printf options

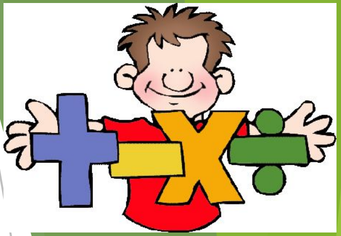
Executable Code

- Expressions consist of legal combinations of
  - constants
  - variables
  - operators
  - function calls

Executable Code

- Operators
  - Arithmetic: +, -, *, /, %
  - Relational: ==, !=, <, <=, >, >=
  - Logical: !, &&, ||
  - Bitwise: &, |, ~, ^
  - Shift: <<, >>
- See Expressions
  - 4th Edition: p. 443-450
  - 3rd Edition: p. 439-445

Arithmetic



- Rules of operator precedence (arithmetic ops):

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

- Average **`a + b + c / 3`** ?

Precedence Example

- Find the average of three variables a, b and c

Do not use:

```
a + b + c / 3
```

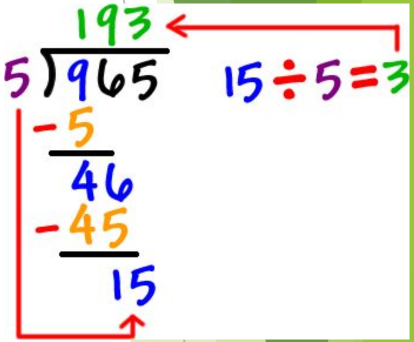
Use:

```
(a + b + c ) / 3
```

The Division Operator

- Generates a result that is the same data type of the largest operand used in the operation.
- Dividing two integers yields an integer result. Fractional part is truncated.

5 / 2 → 2
17 / 5 → 3



  - Watch out: You will not be warned!

The Division Operator

- Dividing one or more decimal floating-point values yields a decimal result.

5.0 / 2	→	**2.5**
4.0 / 2.0	→	**2.0**
17.0 / 5.0	→	**3.4**

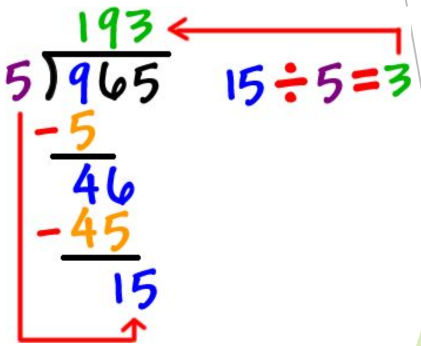
The modulus operator: %

- % modulus operator returns the remainder

7 % 5 → 2

5 % 7 → 5

12 % 3 → 0



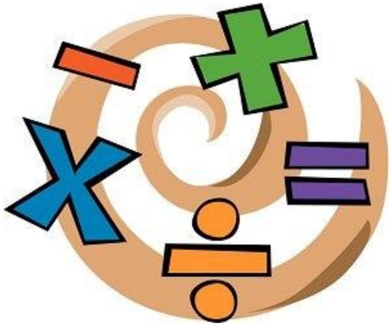
Evaluating Arithmetic Expressions

- Calculations are done ‘one-by-one’ using precedence, left to right within same precedence
  - 11 / 2 / 2.0 / 2 performs 3 separate divisions.

1. 11 / 2 → 5
2. 5 / 2.0 → 2.5
3. 2.5 / 2 → 1.2
   5

Arithmetic Expressions

math expression	C expression
$\frac{a}{b}$	a/b
2 x	2*x
$\frac{x - 7}{2 + 3y}$	(x-7)/(2 + 3*y)

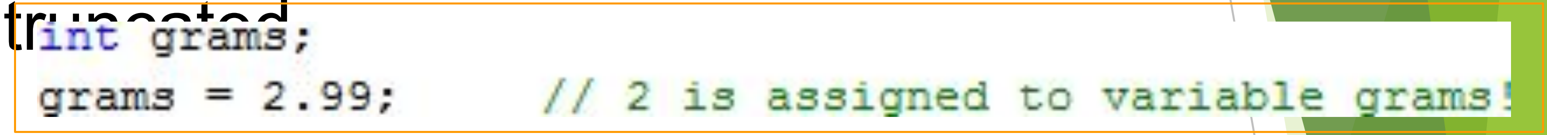


Evaluating Arithmetic Expressions

2 * (-3)	-6
4 * 5 - 15	5
4 + 2 * 5	14
7 / 2	3
7 / 2.0	3.5
2 / 5	0
2.0 / 5.0	0.4
2 / 5 * 5	0
2.0 + 1.0 + 5 / 2	5.0
5 % 2	1
4 * 5/2 + 5 % 2	11

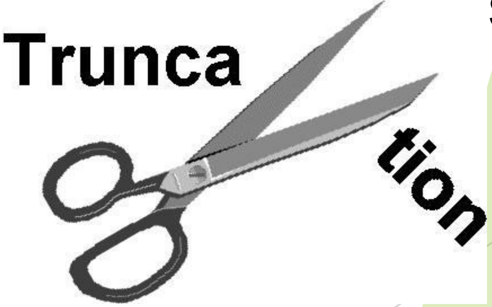
Data Assignment Rules

- In C, when a floating-point value is assigned to an integer variable, the decimal portion is truncated.



```
int grams;
grams = 2.99;      // 2 is assigned to variable grams!
```

- Only integer part 'fits', so that's all that goes
- Called 'implicit' or 'automatic type conversion'



Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C might not evaluate as you ‘expect’!
  - ‘Highest-order operand’ determines type of arithmetic ‘precision’ performed
  - Common pitfall!
  - Must examine each operation

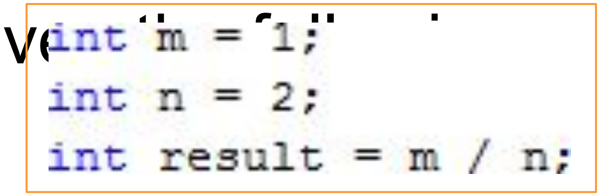
Type Casting

- Casting for Variables
  - Can add ‘.0’ to literals to force precision arithmetic, but what about variables?
    - We can’t use ‘`myInt.0`’!
- type cast – a way of changing a value of one type to a value of another type.
- Consider the expression 1/2: In C this expression evaluates to 0 because both operands are of type integer.

Type Casting

`1 / 2.0` gives a result of `0.5`

Given the following:

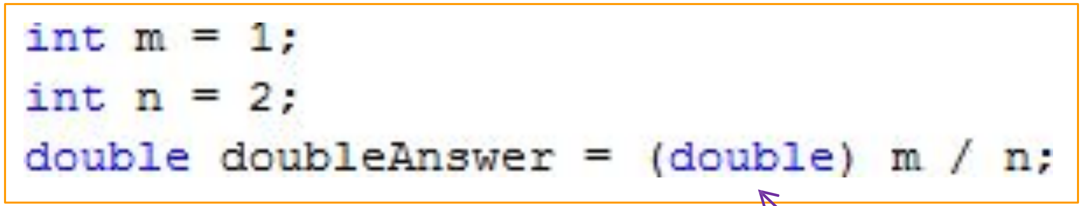


```
int m = 1;
int n = 2;
int result = m / n;
```

result is 0, because of integer division

Type Casting

- To get floating point-division, you must do a type cast from int to double (or another floating-point type), such as the following:



```
int m = 1;
int n = 2;
double doubleAnswer = (double) m / n;
```

Type cast operator

- This is different from `(double) (m/n)`

Type Casting

- Two types of casting
  - Implicit – also called ‘Automatic’
    - Done for you, automatically

      ```
      17 / 5.5
      ```

      This expression causes an ‘implicit type cast’ to take place, casting the 17 □ 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator

      ```
      (double)17 / 5.5

      (double) myInt / myDouble
      ```

Abreviated/Shortcut Assignment Operators

- Assignment expression abbreviations

  `a = a + 3;` can be abbreviated as `a += 3;` using the addition assignment operator

- Examples of other assignment operators include:

Assignment	Shortcut
d = d - 4	d -= 4
e = e * 5	e *= 5
f = f / 3	f /= 3
g = g % 9	g %= 9

Shorthand Operators

- Increment & Decrement Operators
  - Just short-hand notation
  - Increment operator, ++

```
intVar++;  is equivalent to
    intVar = intVar + 1;
```

  - Decrement operator, --

```
intVar--;  is equivalent to
    intVar = intVar – 1;
```

Shorthand Operators:Two Options

- Post-Increment

  `x++`

  - Uses current value of variable, THEN increments it

- Pre-Increment

  `++x`

  - Increments variable first, THEN uses new value

POST

PRE

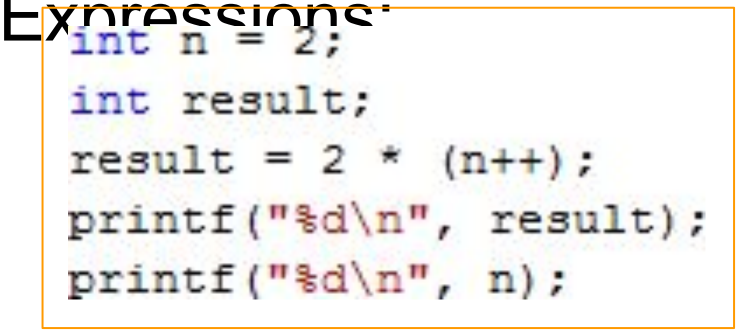
Shorthand Operators:Two Options

- ‘Use’ is defined as whatever ‘context’ variable is currently in
- No difference if ‘alone’ in statement:
  `x++;` and `++x;` identical result

Post-Increment in Action

POST

- Post-Increment in Expressions:



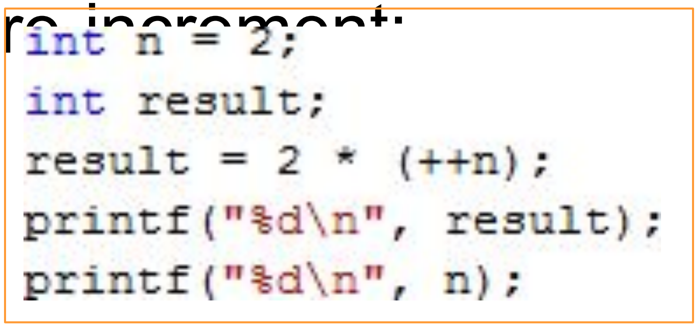
```
int n = 2;
int result;
result = 2 * (n++);
printf("%d\n", result);
printf("%d\n", n);
```

  - This code segment produces the output: 4
    3
  - Since post-increment was used

Pre-Increment in Action

PRE

- Now using pre-increment:



```
int n = 2;
int result;
result = 2 * (++n);
printf("%d\n", result);
printf("%d\n", n);
```

  - This code segment produces the output: 6
    3
  - Because pre-increment was used