

HIGH-LEVEL PROGRAMMING I

Intro to Expressions by Prasanna Ghali

Outline

2

- Variables
- Constants
- Review of Fundamental C Types
- Expressions
- Operators
 - ▣ Precedence
 - ▣ Associativity
- *lvalues* and *rvalues*
- Assignment Operators
 - ▣ Simple Assignment Operators
 - ▣ Compound Assignment Operators

Variables: Definition

3

```
#include <stdio.h>

double compute_bmi(double w, double h);

int main(void) {
    ① printf("Enter your weight in kg: ");
    double weight;
    scanf("%lf", &weight);
    ① printf("Enter your height in m: ");
    double height = 0.0;
    scanf("%lf", &height);
    ① double bmi = compute_bmi(weight, height);
    printf("Wt: %f | Ht: %f | BMI: %f\n",
           weight, height, bmi);
    return 0;
}
```

Every name must be declared (who am I?) before its first use!!!

①

These are **definitions** (where am I?) of variables of type **double**

Definition is declaration **plus** memory allocation

Variables: Initialization

4

```
#include <stdio.h>

double compute_bmi(double w, double h);

int main(void) {
    ① printf("Enter your weight in kg: ");
    double weight;
    scanf("%lf", &weight);
    ② printf("Enter your height in m: ");
    double height = 0.0;
    scanf("%lf", &height);
    ③ double bmi = compute_bmi(weight, height);
    printf("Wt: %f | Ht: %f | BMI: %f\n",
           weight, height, bmi);
    return 0;
}
```

①

Values of variables not initialized are *unspecified* and are said to contain *garbage*

②

`height` is defined and initialized to value `0.0`

③

`bmi` is defined and initialized to value returned by function `compute_bmi`

③

Identifiers (1 / 3)

5

- Names assigned to variables, functions, and other program elements are called *identifiers*
 - ▣ Can *only* contain letters: [a-z] & [A-Z], digits: [0-9], and underscore _
 - ▣ Cannot begin with a digit
 - ▣ Are *case-sensitive*

Identifiers (2/3)

6

- Examples of legal identifiers
 - ▣ Scorex4, opponents_slain, ThisAndThat, _JobDone
- Examples of illegal identifiers
 - ▣ 4xScore, opponents-killed, This&That, Job Done
- First 31 characters in identifier are significant; most compilers allow more

Identifiers (3/3)

7

- Use meaningful identifier names

Bad

```
int x = 80;  
int y = 50;  
int z = x * y;
```

Better

```
int width = 80;  
int height = 50;  
int area = width * height;
```

Character Data

8

- Character-based information and not numbers are most common type of information in programming
- Represented using information coding schemes such as ASCII or utf-8 (which includes ASCII plus other languages and symbols)
- ASCII uses 7-bit encoding of keyboard characters (which is expanded to 8-bits by adding 0 in most significant bit)
- In C, character data is represented by constants of type `int` and by variables of type `char`

Character Data: Constants

9

- Character constant is enclosed in single quotes, such as `'A'`, `'x'`, and `'6'` and has type `int`

Character	ASCII Code	Integer Equivalent
<code>'A'</code>	0100 0001	65
<code>'X'</code>	0101 1000	88
<code>'a'</code>	0110 0001	97
<code>'e'</code>	0110 0101	101
<code>'+'</code>	0010 1011	43
<code>'0'</code>	0011 0000	48
<code>'9'</code>	0011 1001	57
<code>'\n'</code>	0000 1010	10

Character Data: Variables

10

```
char ch;  
ch = 'A'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'X'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'a'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'e'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = '+'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = '\n'; printf("ch: %c | ch: %d\n", ch, ch);
```

ch: A		ch: 65
ch: X		ch: 88
ch: a		ch: 97
ch: e		ch: 101
ch: +		ch: 43
ch:		ch: 10

Since values of type **char** are stored as 8-bit binary values, **char** values can be interpreted as a character or an integer

```
int ch;  
ch = 'A'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'X'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'a'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = 'e'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = '+'; printf("ch: %c | ch: %d\n", ch, ch);  
ch = '\n'; printf("ch: %c | ch: %d\n", ch, ch);
```

Same output!!!

C Types

11

Bit size	Unsigned Integral Types	Signed Integral Types	Floating-Point Types
8-bit	unsigned char	signed char	–
16-bit	unsigned short int OR unsigned short	signed short int OR short	–
32-bit	unsigned int	signed int OR int	float
64-bit	unsigned long int OR unsigned long	signed long int OR long	double
64-bit	unsigned long long int OR unsigned long long	signed long long int OR long long	–
128-bit	–	–	long double

sizeof Operator: Memory Sizes in Bytes

12

```
printf("sizeof(char):           %lu\n", sizeof(char));
printf("sizeof(short):          %lu\n", sizeof(short));
printf("sizeof(int):             %lu\n", sizeof(int));
printf("sizeof(long):            %lu\n", sizeof(long));
printf("sizeof(long long):       %lu\n", sizeof(long long));
printf("sizeof(float):           %lu\n", sizeof(float));
printf("sizeof(double):           %lu\n", sizeof(double));
printf("sizeof(long double): %lu\n", sizeof(long double));
```

`sizeof(type)` or `sizeof(expr)` evaluates to number of bytes required to store value of type *type* or value that *expr* evaluates to

Constants

13

- *Constant* has fixed value with specific type
- Integer constants: see [here](#) for full range
 - ▣ Decimal: `42` (`int`), `42U` (`unsigned int`), `42L` (`long`), `42LU` (`unsigned long`), `42LL` (`long long`), `42LLU` (`unsigned long long`)
 - ▣ Octal: `052` (`int`), `052U` (`unsigned int`), `052L` (`long`), `052LU` (`unsigned long`), `052LL` (`long long`), `052LLU` (`unsigned long long`)
 - ▣ Hexadecimal: `0x2A` (`int`), `0x2AU` (`unsigned int`), `0x2AL` (`long`), `0x2ALU` (`unsigned long`), `0x2ALL` (`long long`), `0x2ALLU` (`unsigned long long`)
 - ▣ Character constants (type `int`): `'X'`, `'2'`, `'\n'`
- Floating-point constants: see [here](#) for full range
 - ▣ `1000.0` (`double`), `1000.0F` (`float`), `1000.0L` (`long double`)
 - ▣ `1.e+3` (`double`), `1.e+3F` (`float`), `1.e+3L` (`long double`)
- String constants (read-only array of `chars`):
`"Hello World!!!\n"`

Preprocessor `define` Directive

(1 / 2)

14

- Give symbolic names to “magic values” using the `#define` preprocessor directive, called *macro definitions*
 - ▣ `#define PI 3.14159265F`
 - ▣ `#define SALES_TAX 0.15f`
- Preprocessor performs basic text replacement

Preprocessor `define` Directive

(2/2)

15

```
#define KG_PER_LB      (0.453592)
#define METER_PER_INCH (1.0/39.3701)

double compute_bmi(double w, double h);

int main(void) {
    printf("Enter your weight in lbs: ");
    double weight;
    scanf("%lf", &weight);
    printf("Enter your height in inches: ");
    double height = 0.0;
    scanf("%lf", &height);

    double bmi = compute_bmi(KG_PER_LB*weight, METER_PER_INCH*height);
    printf("Weight: %f kg | Height: %f m | BMI: %f\n",
           KG_PER_LB*weight, METER_PER_INCH*height, bmi);
    return 0;
}
```

Keywords

16

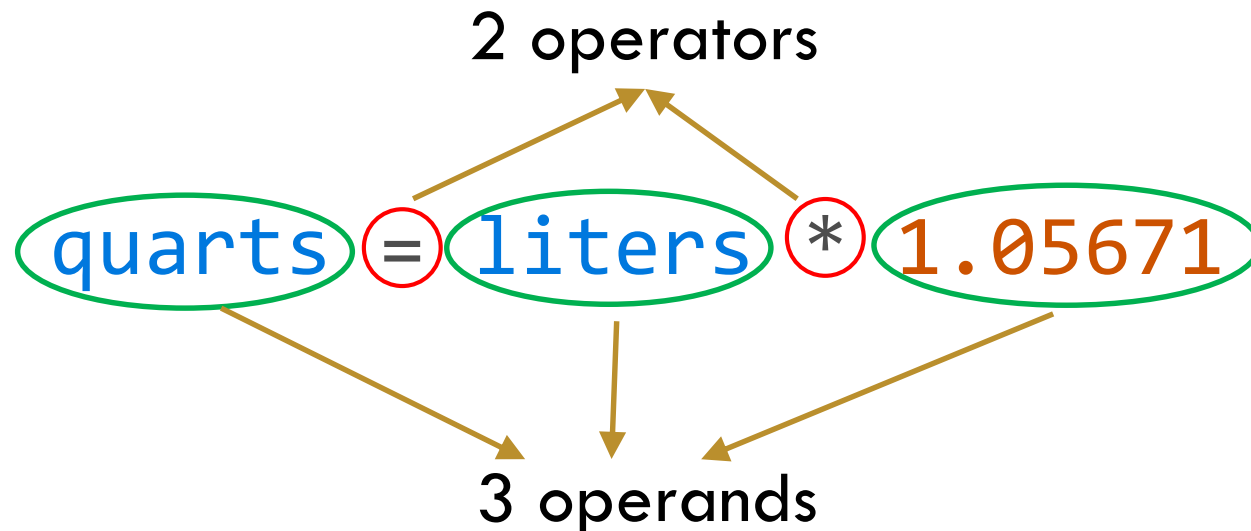
- Identifiers reserved for use by compiler called *keywords*
- Cannot be used by programmer as valid identifiers
- Keywords common to all C and C++ standards

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

What is an Expression?

17

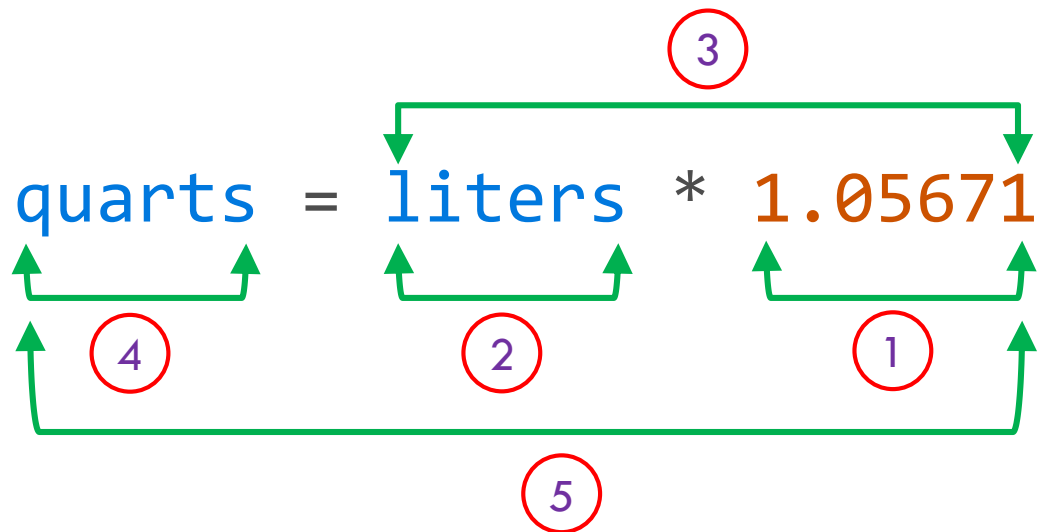
- Composed of one or more **operands** and zero or more **operators**
 - ▣ Operators represent actions
 - ▣ Operands represent objects on which actions are applied



Expression and Subexpressions

18

□ How many subexpressions?



((quarts) = ((liters) * (1.05671)))

What is Evaluation of Expression?

19

- ***Evaluation*** of expression implies application of operators on operands yielding ***result*** with a ***value*** and ***type***
 - If evaluation results in non-zero value, result is ***true***
 - Otherwise, ***false***

Assignment Operator


20

- Used to assign new values to variables (i.e., memory locations)

Memory location
assigned an identifier

Value specified by constant,
another variable, or result of
operation(s)

variable = expression



Assignment Statement

21

assignment operator

this is definition and initialization of new variable, NOT an assignment!!!

name of previously defined variable

```
double area = 0.0;  
.  
area = 0.5 * base * height;  
.  
.  
.  
expression
```

```
area = area + 2.0;
```

same name can occur on both sides of assignment operator

area 6.0

area = area + 2.0

area 8.0

8.0

Arithmetic Operators and Expressions

22

-5

+31

-5 - -5

8 % 7

5 - +31

3 + 4

2 + 3 * 5

5.6 + 6.2 * 3.0

7 + 2 * 5 + 6 / 3

Valid only for integral operands

Behaves differently for integral operands and for floating-point operands

Unary Operator	Meaning
+	Positive
-	Negate

Binary Operator	Meaning
+	Add
-	Subtract
%	Remainder
/	Division
*	Multiply

Precedence and Associativity

23

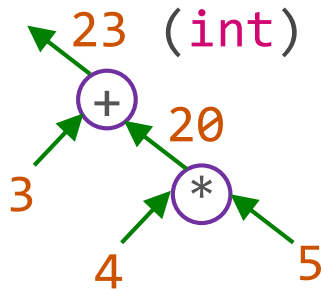
- If expression consists of zero or one operator, compiler evaluates expression **unambiguously**
 - ▣ $2 + 5$ evaluated unambiguously as 7
- Association between operands to operators can be disambiguated with parentheses:
 - ▣ $(2 + 3) * 4$ evaluates to 20
 - ▣ If parentheses are nested, expressions within innermost parentheses are disambiguated first: $2 + ((3 * (4 - 5)) / 6)$ evaluates to 2
- If *unparenthesized* expression contains more than one operator, it can be evaluated **ambiguously**
 - ▣ $2 + 3 * 4$ can evaluate to 20 or 14: $(2+3)*4$ or $2+(3*4)$
- To disambiguate unparenthesized expression evaluation, every operator has **precedence level** and **associativity order**

Precedence Level

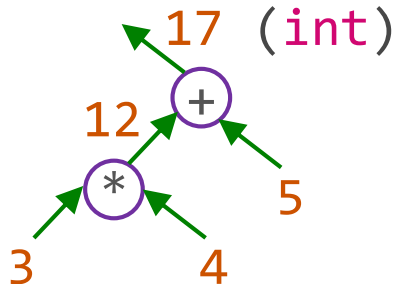
24

- How “tightly” does operator bind to operands in unparenthesized expression?

3 + 4 * 5



3 * 4 + 5



high to low
precedence order

Operator	Meaning
+ -	unary plus, unary minus
* / %	multiplication, division, remainder
+ -	addition, subtraction
=	assignment

Associativity Rule (1 / 4)

25

- Many operators have same precedence level!!!
 - ▣ $3 * 4 / 5$ can evaluate to 2 or 0
 - ▣ What about $3 / 4 * 5$?

precedence order high to low ↓	Operator	Meaning
	+ -	unary plus, unary minus
	* / %	multiplication, division, remainder
	+ -	addition, subtraction
	=	assignment

Associativity Rule (2/4)

26

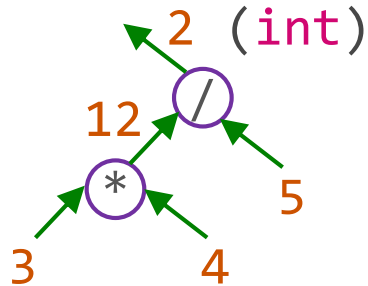
- Protocol for describing “real” precedence among operators with same precedence level
 - ▣ **Right** associative rule means group operands from *right to left*
 - ▣ **Left** associative rule means group operands from *left to right*

Associativity Rule (3/4)

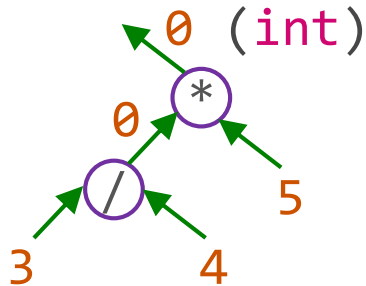
27

Really Meh

3 * 4 / 5



3 / 4 * 5



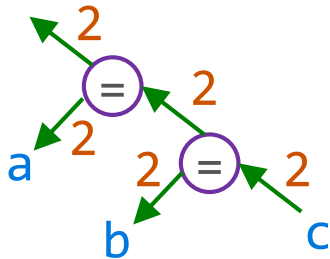
precedence order
high to low

Operator	Associativity
+ -	R-L
* / %	L-R
+ -	L-R
=	R-L

Associativity Rule (4/4)

28

```
int a, b = 1, c = 2;  
a = b = c;  
// is a assigned value 1 or 2?
```



precedence order
high to low

Operator	Associativity
+ -	R-L
* / %	L-R
+ -	L-R
=	R-L

Precedence and Associativity (1 / 2)

29

Arithmetic Expression	Result	Type
2+5		
13.1 + 89.2		
34 - 20		
45.12f + 90.34F		
5 / 2		
2u / 7U		
34L % 5l		
4.0L * 6.0l		
0x03 + 0x04 / 0x05		
3 * 7 - 6 + 2 * 5 / 4 + 6		

Precedence and Associativity (2/2)

30

```
int a=1, b=2, c=3;    double x=1.0, y=2.0, z=3.0;
```

Arithmetic Expression	Result	Type
$a = b + 2 = c = 5$		
$3 + 4 * 5$		
$3 + a - b / 7$		
$x + 2.0 * (y - z) + 18.0$		
$12.8 * 0.5 - 34.50$		
$x * 10.5 + y - 16.2$		
$3 * 7 - 6 + 2 * 5 / 4 + c$		
$x = x + 3.0 + z * y$		
$a = 4 - 3 * b / 8$		
$a + b = 8$		

Implicit Type Conversions (1 / 3)

31

- Expressions with operands of different data types are called ***mixed expressions***
 - ▣ Bad idea – source of many bugs!!!
- When evaluating mixed expressions, ***implicit type conversions*** take place according to set of pre-defined type conversion rules
 - ▣ C's idea of type conversion: Lossless - don't lose values!!!

```
int quarts = 4;  
double liters = 0.0;  
  
liters = quarts/1.05671;
```

Implicit Type Conversions (2/3)

32

Mixed Expression	Result	Type
$3 / 2 + 5.5$		
$15.6 / 2 + 5$		
$4 + 5 / 2.0$		
$4 * 3 + 7 / 5 - 25.5$		

Implicit Type Conversions (3/3)

33

- C/C++ rules for implicit type conversions and promotions are mind boggling
- Our advice: Avoid such conversions and promotions at all costs!!!

Cast Operator (1 / 2)

34

- To avoid and/or to document implicit type conversion, programmer can provide **explicit** type conversion thro' **cast operator**
 - ▣ *(type name) expression*

precedence order high to low ↓	Operator	Associativity
	+ - (<i>type</i>)	R-L
	* / %	L-R
	+ -	L-R
	=	R-L

Cast Operator (2/2)

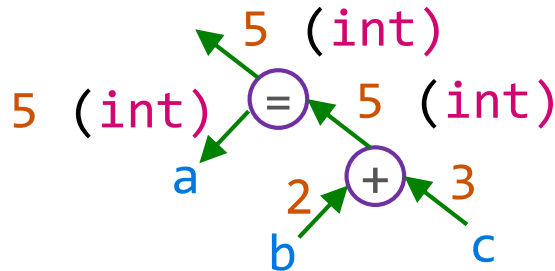
35

Expression	Evaluates to	Type
<code>(int) 7.9</code>		
<code>(double) 25</code>		
<code>(double) 5 + 3</code>		
<code>(double) 15/2</code>		
<code>(double) (15/2)</code>		
<code>(int) (7.8 + (double)(15)/2)</code>		
<code>(int) (7.8 + (double)(15/2))</code>		
<code>(int) 12.8 * 17.5 - 34.50</code>		
<code>(int)('A')</code>		
<code>(char)67</code>		

Side Effect with Assignment Operator

36

- Most operators do not **modify** their operands
- Assignment operator has **side effect** of modifying left operand
 - ▣ Left operand must therefore represent memory location



```
int a = 1, b = 2, c = 3;  
a = b + c;
```

lvalues and rvalues (1 / 6)

37

- Why is this code legal?

```
int a = 1, b = 2, c = 3;  
a = b + c;
```

- Why is this code also legal?

```
int a = 1, b = 2, c = 3;  
c = a + (b = c);
```

- But, not this code?

```
int a = 1, b = 2, c = 3;  
a + b = c;
```

lvalues and *rvalues* (2/6)

38

- Every expression is an *lvalue* or an *rvalue*
- ***lvalue*** (short for *locator value*) is expression that refers to identifiable memory location
- By exclusion, any non-*lvalue* expression is an ***rvalue*** - think of ***rvalue*** as “value resulting from expression”
- Useful to visualize a variable as name associated with certain memory locations `int x = 99;`
- Sometimes (as an *lvalue*) `x` means its memory locations and sometimes `x` (as an *rvalue*) means value stored in those memory locations

here, `x` means *lvalue*

while here, `x` means *rvalue*

```
int x = 99;  
x = 100;  
x = x + 2;
```

1000
1001
1002
1003

99

x

lvalues and rvalues (3/6)

39

- Every operator seen so far can take *rvalue(s)* as operand(s) except assignment operator ***whose left operand can only be lvalue***

```
// a, b, and c specify memory locations  
// and can be both lvalues and rvalues
```

```
int a = 1, b, c;
```

```
c = a; // expression a is rvalue; expression c is lvalue
```

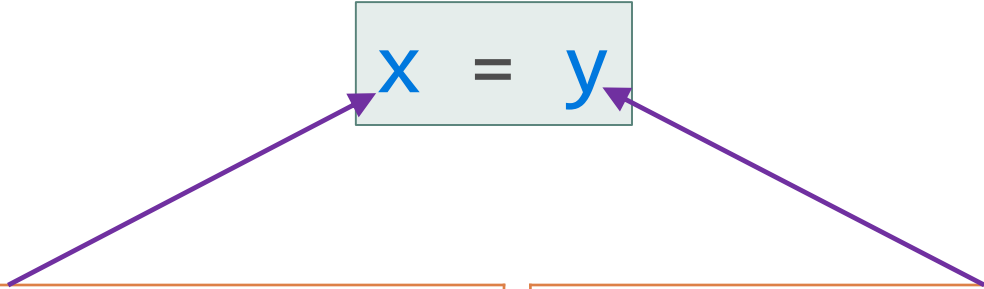
```
b = a + 3; // expr a + 3 is rvalue; expr b is lvalue
```

```
a = b + c; // expr b + c is rvalue; expr a is lvalue
```

```
b + a = c; // error: expr c is rvalue; expr b + a is rvalue
```

lvalues and *rvalues* (4/6)

40



$x = y$

Symbol x , in this context, means
“address that x represents”

This expression is termed ***lvalue***

lvalue means “ x ’s memory location”

lvalue is known at compile-time

Symbol y , in this context, means
“contents of address that y
represents”

This expression is termed ***rvalue***

rvalue means “value of y ”

rvalue is not known until run-time

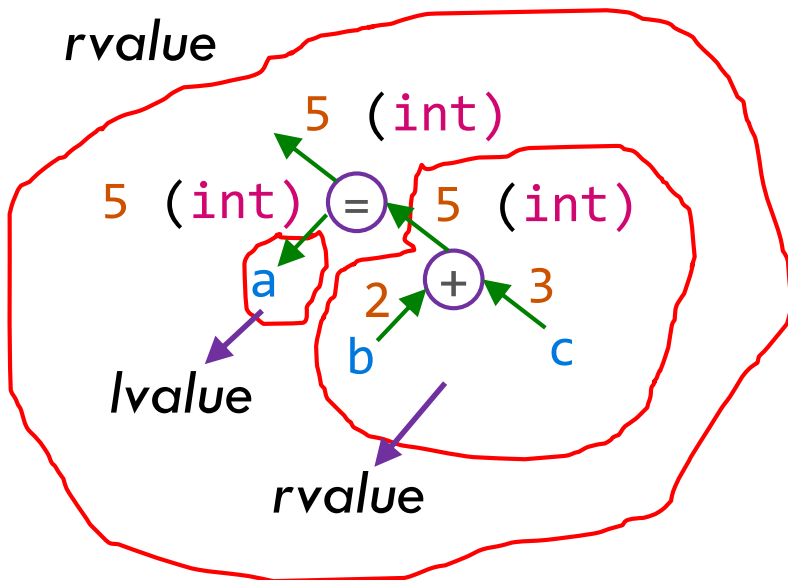
lvalues and rvalues (5/6)

41

- Every operator seen so far can take *rvalue*(s) as operand(s) except assignment operator **whose left operand can only be lvalue**
- All operators seen so far – including assignment operator - evaluate their expressions to *rvalue*

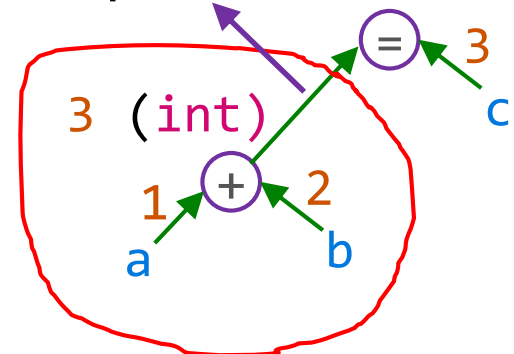
```
int a = 1, b = 2, c = 3;
```

```
a = b + c
```



```
a + b = c
```

rvalue expression but
lvalue required!!!



lvalues and rvalues (6/6)

42

□ Is expression in 2nd statement legal?

```
int a = 1, b = 2, c = 3;  
(((a = b) = c) = 5);
```

□ No!!!

▣ Because rvalue expression `a = b` in innermost parentheses is left operand in assignment expression `(a = b) = c`

Assignment Statement: Review

(1 / 2)

43

- Suppose `num1`, `num2`, and `num3` are `int` variables and following statements are executed in sequence

```
1. num1 = 18;  
2. num1 = num1 + 27;  
3. num2 = num1;  
4. num3 = num2 / 5;  
5. num3 = num3 / 4;
```

Line #	num1	num2	num3
1.			
2.			
3.			
4.			
5.			

- Show values of variables after execution of each statement

Assignment Statement: Review

(2/2)

44

Suppose you've following definitions:

```
int a = 1, b = 2, c = 3, d, x, y;
```

Further suppose that at a later point, you want to evaluate expressions $-b + b^2 - 4ac$ and $-b - b^2 + 4ac$ and assign values of these expressions to `x` and `y`, respectively.

How would you use variables `d`, `x`, and `y`?

`d` =

`x` =

`y` =

Compound Assignment Operator

(1 / 2)

45


- For each arithmetic operator, C provides *compound assignment operator* to write assignment expressions more concisely

Simple assignment statement	Compound assignment statement
<code>i = i + 5;</code>	<code>i += 5;</code>
<code>counter = counter + 1;</code>	<code>counter += 1;</code>
<code>health = health - 9;</code>	<code>health -= 9;</code>
<code>sum = sum % number;</code>	<code>sum %= number;</code>
<code>amount = amount * (interest + 1);</code>	<code>amount *= (interest + 1);</code>
<code>x = x / (y + 5);</code>	<code>x /= (y + 5);</code>

Compound Assignment Operator

(2/2)

46



Operator	Associativity
<code>+</code> <code>-</code> <code>(type)</code> <code>sizeof</code>	R-L
<code>*</code> <code>/</code> <code>%</code>	L-R
<code>+</code> <code>-</code>	L-R
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>	R-L

Tokens (1 / 3)

47

- After preprocessing, C source code is broken down into series of *tokens*
 - ▣ Groups of characters that cannot be split up without changing their meaning

Tokens (2/3)

48

- Tokens in source code are often separated by *whitespace*
 - ▣ E.g. spaces, tabs, newlines
 - ▣ Whitespace are **not tokens**
- *Comments* used to document source code
 - ▣ Replaced by preprocessor with space – therefore never seen by compiler
 - ▣ Single line comment begins with `//`
 - ▣ Multiline comment begins with `/*` and ends with `*/`

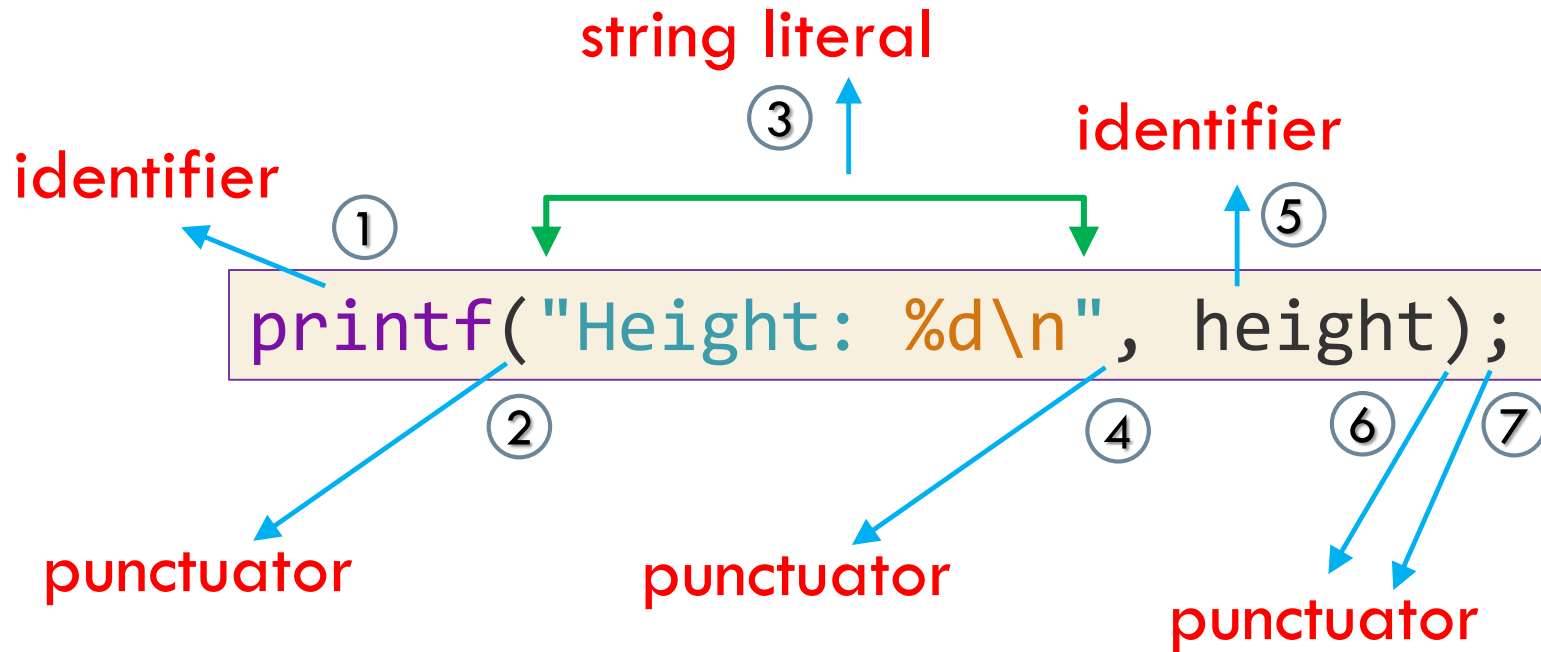
Tokens (3/3)

49

- Classes of tokens:
 - ▣ Identifiers
 - ▣ Keywords
 - ▣ Constants
 - ▣ String literals
 - ▣ Operators
 - ▣ Punctuators

How Many Tokens in Statement?

50



Summary

51

- Identifiers: Names for C/C++ objects; Naming rules enforced; Reserved identifiers called keywords
- Variables: Every variable must be defined before first use
- Constants: Literal value; Use suffixes to indicate type of literal type
- Review of Fundamental C Types
- Expression: Evaluation; Assignment operators; Arithmetic operators; `sizeof(type)`, `cast ()`,
- Precedence and associativity of operators: No need to memorize; will become part of your vocabulary thro' constant use
- Implicit type conversions: Mixed expressions are bad; don't mix types; cast operator
- *lvalues* and *rvalues*
- Preprocessor `define` directive
- Tokens: What are they? How to parse expression or statement for tokens?

Precedence and Associativity (1 / 2): Solutions

52

Arithmetic Expression	Result	Type
<code>2+5</code>	<code>7</code>	<code>int</code>
<code>13.1 + 89.2</code>	<code>102.3</code>	<code>double</code>
<code>34 - 20</code>	<code>14</code>	<code>int</code>
<code>45.12f + 90.34F</code>	<code>135.46</code>	<code>float</code>
<code>5 / 2</code>	<code>2</code>	<code>int</code>
<code>2u / 7U</code>	<code>0</code>	<code>unsigned int</code>
<code>34L % 5l</code>	<code>6</code>	<code>long int</code>
<code>4.0L * 6.0l</code>	<code>24.0</code>	<code>long double</code>
<code>0x03 + 0x04 / 0x05</code>	<code>3</code>	<code>int</code>
<code>3 * 7 - 6 + 2 * 5 / 4 + 6</code>	<code>23</code>	<code>int</code>

Precedence and Associativity (2/2): Solutions

53

```
int a=1, b=2, c=3;    double x=1.0, y=2.0, z=3.0;
```

Arithmetic Expression	Result	Type
$a = b + 2 = c = 5$	-	Illegal expression
$3 + 4 * 5$	23	int
$3 + a - b / 7$	4	int
$x + 2.0 * (y - z) + 18.0$	17.0	double
$12.8 * 0.5 - 34.50$	-28.1	double
$x * 10.5 + y - 16.2$	-3.7	double
$3 * 7 - 6 + 2 * 5 / 4 + c$	20	int
$x = x + 3.0 + z * y$	10.0	double
$a = 4 - 3 * b / 8$	4	int
$a + b = 8$	-	Illegal expression

Implicit Type Conversions (2/2): Solutions

54

Mixed Expression	Result	Type
$3 / 2 + 5.5$	6.5	double
$15.6 / 2 + 5$	12.8	double
$4 + 5 / 2.0$	6.5	double
$4 * 3 + 7 / 5 - 25.5$	-12.5	double

Cast Operator (2/2): Solution

55

Expression	Evaluates to	Type
<code>(int) 7.9</code>	7	int
<code>(double) 25</code>	25.0	double
<code>(double) 5 + 3</code>	8.0	double
<code>(double) 15/2</code>	7.5	double
<code>(double) (15/2)</code>	7.0	double
<code>(int) (7.8 + (double)(15)/2)</code>	15	int
<code>(int) (7.8 + (double)(15/2))</code>	14	int
<code>(int) 1.28 * 17.5 - 34.50</code>	-17.0	double
<code>(int)('A')</code>	65	int
<code>(char)67</code>	'C'	char

Assignment Statement: Review

(1 / 2) - Solutions

56

- Suppose `num1`, `num2`, and `num3` are `int` variables and following statements are executed in sequence

- 1. `num1 = 18;`
- 2. `num1 = num1 + 27;`
- 3. `num2 = num1;`
- 4. `num3 = num2 / 5;`
- 5. `num3 = num3 / 4;`

Line #	num1	num2	num3
1.	18	?	?
2.	45	?	?
3.	45	45	?
4.	45	45	9
5.	45	45	2

- Show values of variables after execution of each statement

Assignment Statement: Review

(2/2): Solutions

57

Suppose you've following definitions:

```
int a = 1, b = 2, c = 3, d, x, y;
```

Further suppose that at a later point, you want to evaluate expressions $-b + b^2 - 4ac$ and $-b - b^2 + 4ac$ and assign values of these expressions to `x` and `y`, respectively.

How would you use variables `d`, `x`, and `y`?

```
d = b*b - 4*a*c;
```

```
x = -b + d;
```

```
y = -b - d;
```
