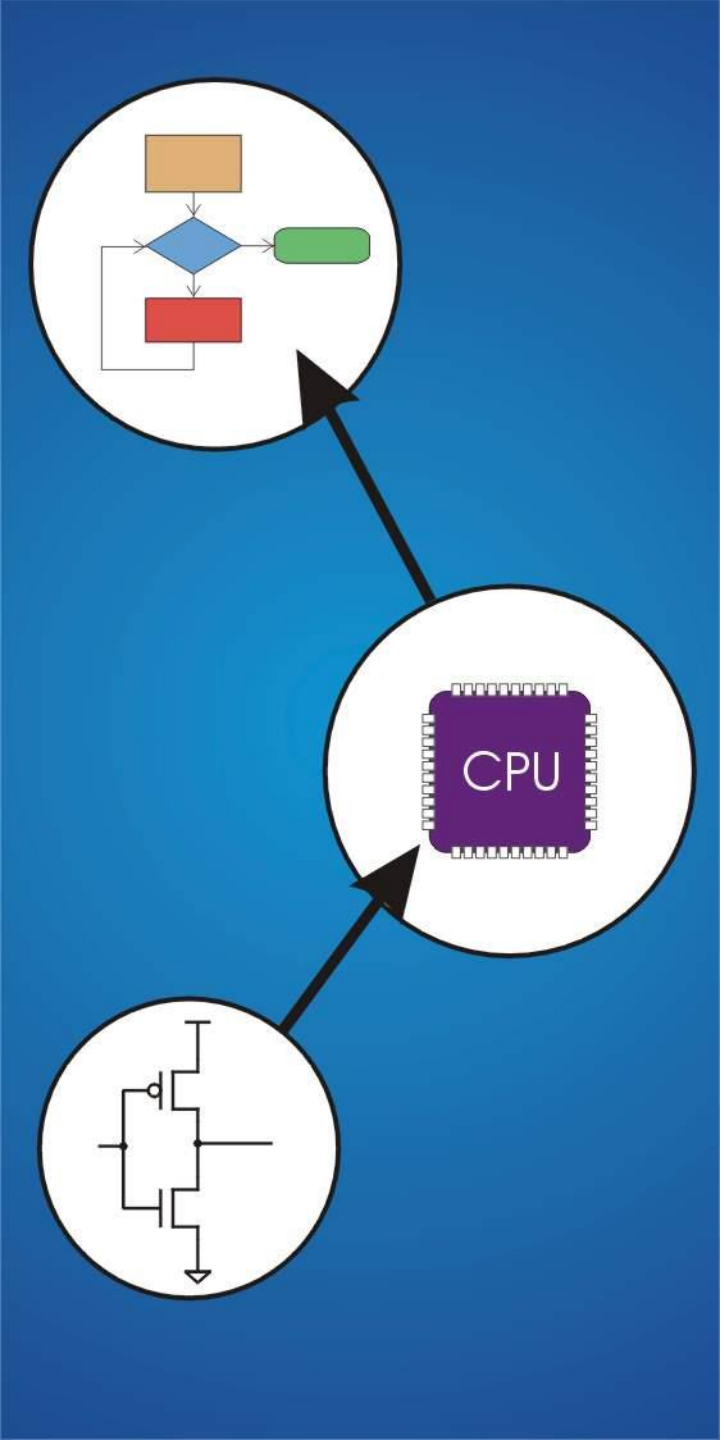
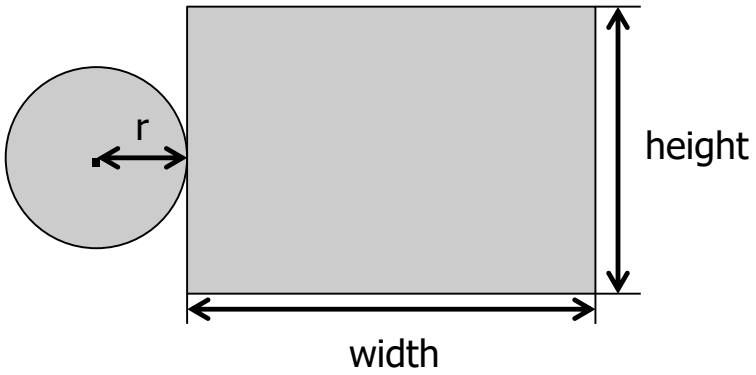
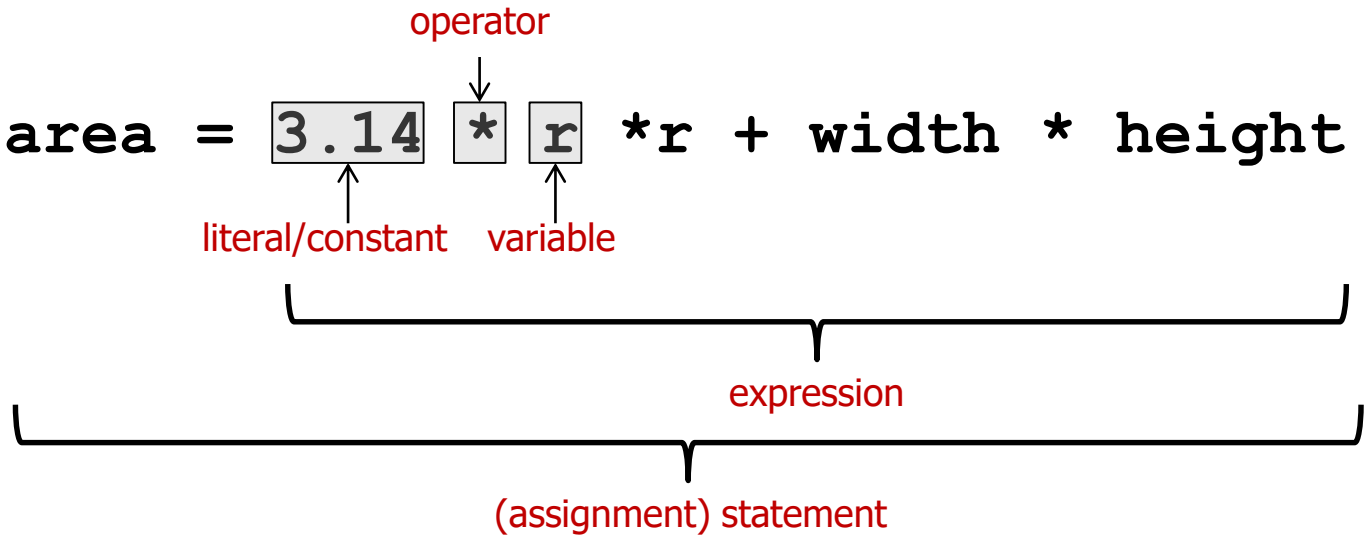


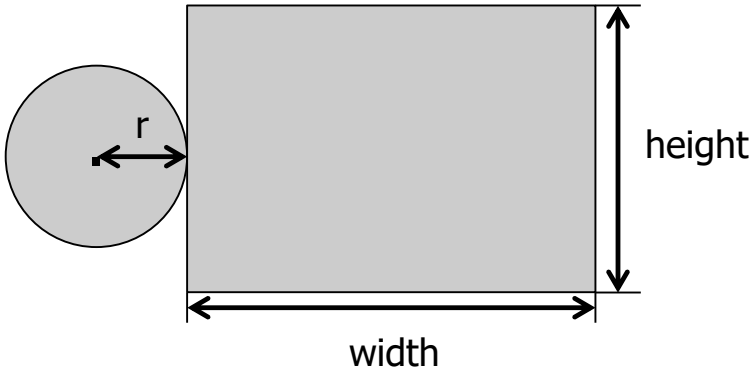
Chapter 12
Variables and Operators

Highlights (1)





Highlights (2)




```
area = 3.14 * r *r + width * height
```

- **Precedence:** 우선 순위

```
area = 3.14 * r *r + (width * height)
```

- **Associativity:** 결합 법칙

```
area = ((3.14 * r) *r) + width * height
```

- **also need to distinguish value (from evaluating an expression on the right-hand side) from action (assignment of a value to a variable on the left-hand side)**

Basic C Elements

Variable

- **holds a value upon which a program acts**
- **has a type (integer, floating-point, character, *etc*)**
- **is an abstraction (i.e., symbolic name) of a memory location but its value is moved to a register when subject to an operation and moved back to memory when registers are full**

Operators

- **predefined actions performed on data values**
- **combined with variables and constants (literals) to form expressions**
- **Is an abstraction of arithmetic / logic instructions**

Data Types

C has three basic data types

`int`	**integer (at least 16 bits)**
`double`	**floating point (at least 32 bits)**
`char`	**character (at least 8 bits)**

Exact size can vary, depending on ISA

- **int is supposed to be "natural" integer size;**
  for LC-3, that's 16 bits -- 32 bits or 64 bits for most modern processors

Variable Declaration

Syntax

- `int` <variable name list>;
- `double` <variable name list>;
- `char` <variable name list>;

Examples

- `int score;`
- `int score, sum;`
- `double area, volumn;`
- `char InputChar;`
- `int sum = 0;`

Variable Name (i.e., *identifier*)

Any combination of letters, numbers, and underscore (_)

Case matters

- **"sum" is different than "Sum"**

Cannot begin with a number

- **usually, variables beginning with underscore are used only in special library routines**

Only first 31 characters are meaningful

Examples

Legal

```
i
wordsPerSecond
words_per_second
_green
aReally_longName_moreThan31chars
aReally_longName_moreThan31characters
```

same identifier

Illegal

```
10sdigit
ten'sdigit
done?
double
```

reserved keyword

Literals (i.e., constants)

Integer

```
123   /* decimal */
-123
0x123 /* hexadecimal */
```

Floating point

```
6.023
6.023e23  /* 6.023 x 10^23 */
5E12      /* 5.0 x 10^12 */
```

Character

```
'c'
'\n'   /* newline (more on Page 567 */
'\xA'  /* ASCII 10 - hexadecimal */
'\012' /* ASCII 10 - octal */
```

Scope: Global and Local

Where is the variable accessible?

Global: accessed anywhere in program

Local: only accessible in a particular region

Compiler infers scope from where variable is declared

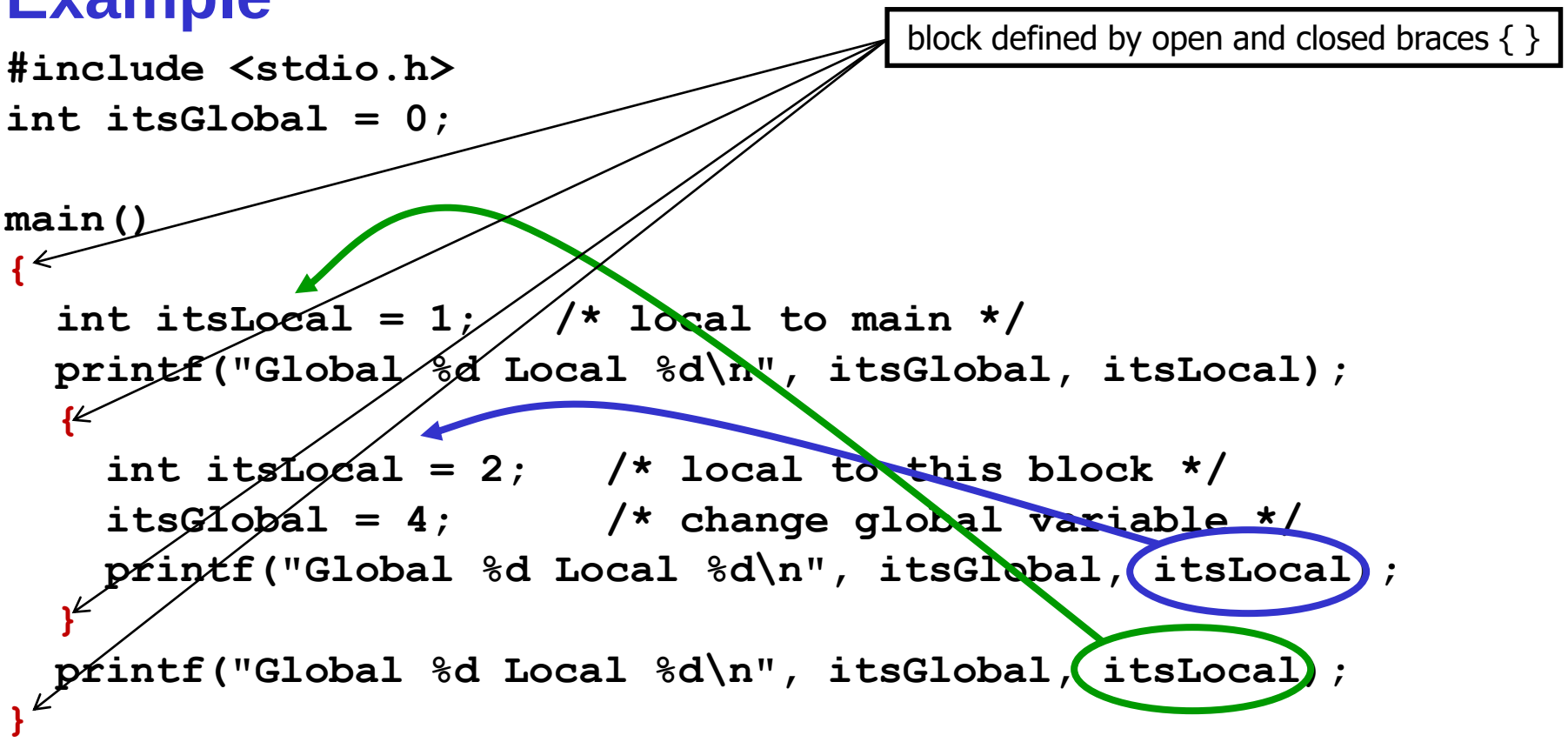
- **programmer doesn't have to explicitly state**

Variable is local to the block in which it is declared

- **block defined by open and closed braces { }**
- **can access variable declared in any "containing" block**

Global variable is declared outside all blocks

Example




```
#include <stdio.h>
int itsGlobal = 0;

main()
{
  int itsLocal = 1;    /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  {
    int itsLocal = 2;    /* local to this block */
    itsGlobal = 4;       /* change global variable */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
  }
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```

Operators

Programmers manipulate values (ones stored in variables, literals/constants and, return values from functions) using the operators provided by the high-level language.

Variables, constants, and operators combine to form an expression, which yields a value

Each operator may correspond to many machine instructions.

- **Example: The multiply operator (*) typically requires multiple LC-3 ADD instructions.**

Expression

Any combination of variables, literals/constants, operators, and function calls (i.e., their return values) that yield to a value

- **every expression has a type (integer, floating-point, character, boolean) derived from the types of its components (according to rules defined in the C programming language)**

Examples:

```
3.14
width
width*height
3.14*r*r + width*height
counter >= STOP
x + sqrt(y)
x & z + 3 || 9 - w-- % 6
```

Statement

Expresses a complete unit of work

- **executed in sequential order**

Simple statement ends with semicolon

```
z = x * y;
y = y + 1;
;  /* null statement */
```

Compound statement groups simple statements using braces (why needed?).

- **syntactically equivalent to a simple statement**

```
{  z = x * y; y = y + 1; }
```

Operators

Three things to know about each operator

(1) Function

- what does it do?

(2) Precedence

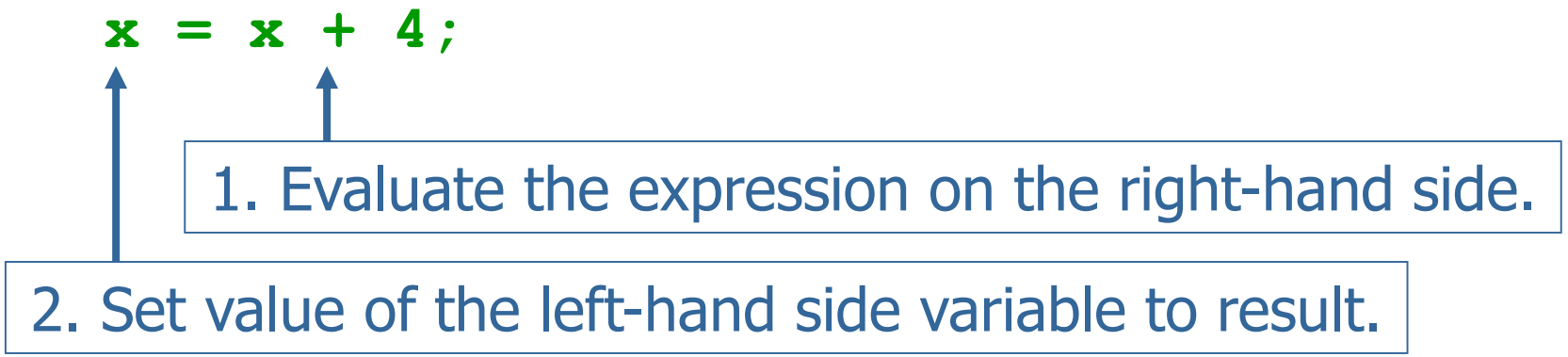
- in which order are operators combined?
- Example:
  "a * b + c * d" is the same as "(a * b) + (c * d)"
  because multiply (*) has a higher precedence than addition (+)

(3) Associativity

- in which order are operators of the same precedence combined?
- Example:
  "a - b - c" is the same as "(a - b) - c"
  because add/sub associate left-to-right

Assignment Statement (action)

Changes the value of a variable.



Assignment Operator (value) – unique in C

**All expressions evaluate to a value,
even ones with the assignment operator.**

For assignment, the result is the value assigned.

- **the value of the right-hand side**
- **the value of expression `y = x + 3` is the current value of variable `x` plus 3**

Assignment associates right to left.

```
y = x = 3; is equivalent to y = (x = 3);
```

y gets the value 3, because (x = 3) evaluates to the value 3.

Arithmetic Operators

Symbol	Operation	Usage	Precedence	Assoc
*	multiply	`x * y`	6	l-to-r
/	divide	`x / y`	6	l-to-r
%	modulo	`x % y`	6	l-to-r
+	addition	`x + y`	7	l-to-r
–	subtraction	`x - y`	7	l-to-r

All associate left to right.

`/` `%` have higher precedence than `+` `-`.

Operate on *values* -- neither operand is changed.

Arithmetic Expressions

If mixed types, smaller type is "promoted" to larger.

```
x + 4.3
```

if x is int, converted to double and result is double

Integer division -- fraction is dropped.

```
x / 3
```

if x is int and x=5, result is 1 (not 1.666666...)

Modulo -- result is remainder.

```
x % 3
```

if x is int and x=5, result is 2.

Bitwise Operators

Symbol	Operation	Usage	Precedence	Assoc
~	bitwise NOT	`~x`	4	r-to-l
<<	left shift	`x << y`	8	l-to-r
>>	right shift	`x >> y`	8	l-to-r
&	bitwise AND	`x & y`	11	l-to-r
^	bitwise XOR	`x ^ y`	12	l-to-r
|	bitwise OR	`x | y`	13	l-to-r

Operate on variables bit-by-bit.

- **Like LC-3 AND and NOT instructions.**

Relational Operators

Symbol	Operation	Usage	Precedence	Assoc
>	greater than	`x > y`	9	l-to-r
>=	greater than or equal	`x >= y`	9	l-to-r
<	less than	`x < y`	9	l-to-r
<=	less than or equal	`x <= y`	9	l-to-r
==	equal	`x == y`	10	l-to-r
!=	not equal	`x != y`	10	l-to-r

Result is 1 (TRUE) or 0 (FALSE): boolean type

Note: Don't confuse equality (==) with assignment (=).

Logical Operators

Symbol	Operation	Usage	Precedence	Assoc
!	logical NOT	`!x`	4	r-to-l
&&	logical AND	`x && y`	14	l-to-r
||	logical OR	`x || y`	15	l-to-r

Treats the operands as boolean type: TRUE (non-zero) or FALSE (zero).

Result is 1 (TRUE) or 0 (FALSE): boolean type

Special Operators: ++ and -- (unique in C)

Changes value of variable after (or before) its value is used in an expression.

Symbol	Operation	Usage	Precedence	Assoc
++	postincrement	`x++`	2	r-to-l
--	postdecrement	`x--`	2	r-to-l
++	preincrement	`++x`	3	r-to-l
--	predecrement	`--x`	3	r-to-l

Pre: Increment/decrement variable before using its value.

Post: Increment/decrement variable after using its value.

Using ++ and --

```
x = 4;
y = x++;
```

Results: x = 5, y = 4
(because x is incremented after using its value)

```
x = 4;
y = ++x;
```

Results: x = 5, y = 5
(because x is incremented before using its value)

Special Operators: +=, *=, etc. (unique in C)

Arithmetic and bitwise operators can be combined with assignment operator.

Statement	Equivalent assignment
`x += y;`	`x = x + y;`
`x -= y;`	`x = x - y;`
`x *= y;`	`x = x * y;`
`x /= y;`	`x = x / y;`
`x %= y;`	`x = x % y;`
`x &= y;`	`x = x & y;`
`x |= y;`	`x = x | y;`
`x ^= y;`	`x = x ^ y;`
`x <<= y;`	`x = x << y;`
`x >>= y;`	`x = x >> y;`

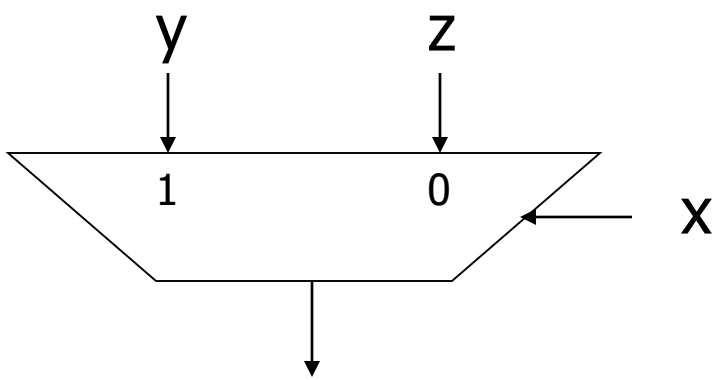
All have same precedence and associativity as = and associate right-to-left.

Special Operator: Conditional

Symbol	Operation	Usage	Precedence	Assoc
? :	conditional	`x?y:z`	16	l-to-r

If x is TRUE (non-zero), result is y;
else, result is z.

Like a MUX, with x as the select signal.



Practice with Precedence

Assume a=1, b=2, c=3, d=4.

```
x = a * b + c * d++ / 2;  /* x = 8 */
```

same as:

```
x = (a * b) + ((c * (d++)) / 2);
```

For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.

Summary: Operator Precedence and Associativity

Precedence	Associativity	Operators
1 (highest)	l(eft) to r(ight)	() (function call) [ ] (array index) . ->
2	r(ight) to l(eft)	++ - - (postfix versions)
3	r to l	++ - - (prefix versions)
4	r to l	* (indirection) & (address of)
		+ (unary) - (unary) ~ (logical NOT) ! (bitwise NOT) sizeof
5	r to l	(type) (type cast)
6	l to r	* (multiplication) / %
7	l to r	+ (addition) - (subtraction)
8	l to r	<< >> (shifts)
9	l to r	< > <= >= (relational operators)
10	l to r	== != (equality/inequality operators)
11	l to r	& (bitwise AND)
12	l to r	^ (bitwise exclusive OR)
13	l to r	| (bitwise OR)
14	l to r	&& (logical AND)
15	l to r	|| (logical OR)
16	l to r	?: (conditional expression)
17 (lowest)	r to l	= += -= *= etc. (assignment operators)

Symbol Table

Like assembler, compiler needs to know information associated with identifiers

- **in assembler, all identifiers were labels and information is address**

Compiler keeps more information

- **Name (identifier)**
- **Type**
- **Location in memory**
- **Scope**

Name	Type	Offset	`Scope`
amount	int	0	main
hours	int	-3	main
minutes	int	-4	main
rate	int	-1	main
seconds	int	-5	main
time	int	-2	main

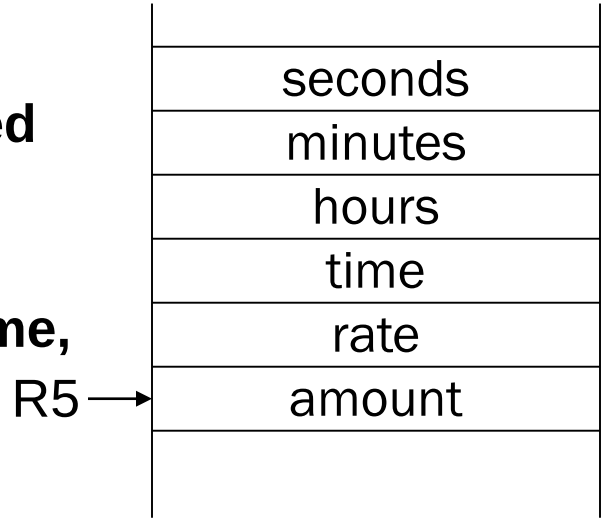
Local Variable Storage

Local variables are stored in user stack called an *activation record*, also known as a *stack frame*.

```
#include <stdio.h>

int main()
{
   int amount;
   int rate;
   int time;

   int hours;
   int minutes;
   int seconds;
…
```

Symbol table "offset" gives the distance from the base of the frame.

- **R5 is the frame pointer – holds address of the base of the current frame.**
- **A new frame is pushed on the run-time stack each time a function is called or a block is entered.**
- **Because stack grows downward, R5 contains the highest address of the frame, and variable offsets are <= 0.**



Allocating Space for Variables

Global data section

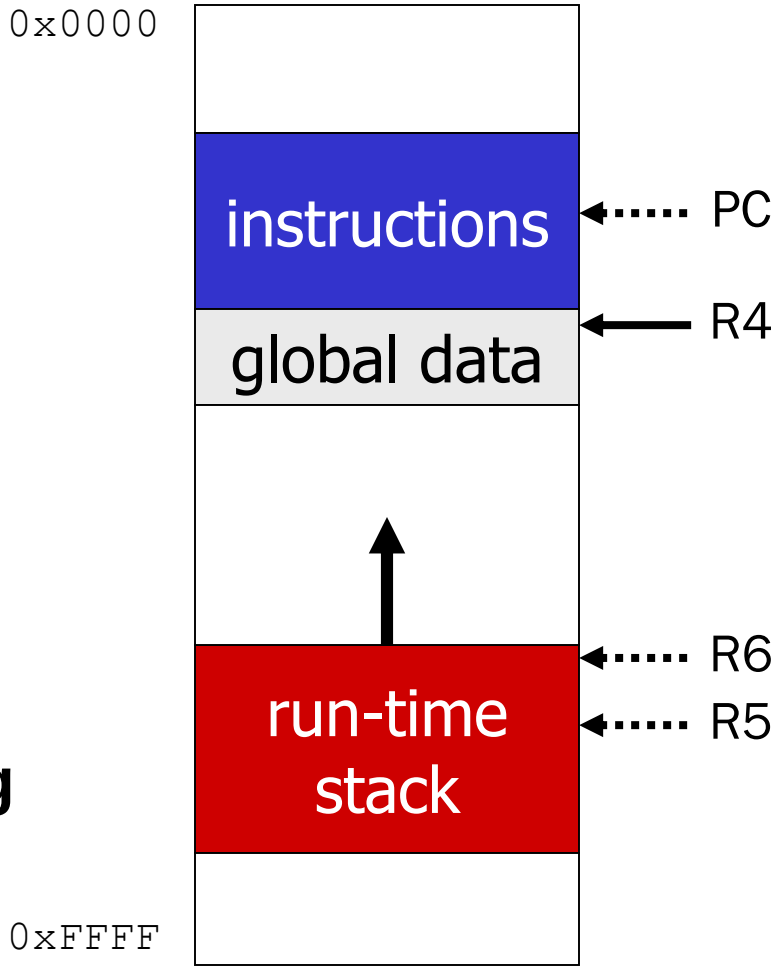
- All global variables stored here (actually all static variables)
- R4 points to beginning

Run-time stack

- Used for local variables
- R6 points to top of stack
- R5 points to top frame on stack
- New frame for each block (goes away when block exited)

Offset = distance from beginning of storage area

- Global: `LDR R1, R4, #4`
- Local: `LDR R2, R5, #-3`



Variables and Memory Locations

In our examples,
a variable is always stored in memory.

When assigning to a variable,
must store to memory location.

A real compiler would perform code optimizations that try to keep variables allocated in registers.

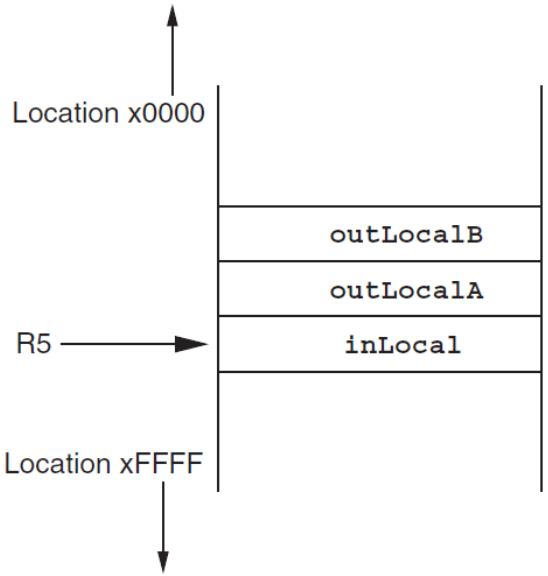
Why?

Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;   /* local to main */
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n",
         outLocalA, outLocalB);
}
```

Example: Symbol Table

Name	Type	Offset	Scope
`inGlobal`	`int`	`0 (R4)`	`global`
`inLocal`	`int`	`0 (R5)`	`local:main`
`outLocalA`	`int`	`-1 (R5)`	`local:main`
`outLocalB`	`int`	`-2 (R5)`	`local:main`



Example: Code Generation

```
; main
; initialize variables
        AND R0, R0, #0
        ADD R0, R0, #5   ; inLocal = 5
        STR R0, R5, #0   ; (offset = 0)

        AND R0, R0, #0
        ADD R0, R0, #3   ; inGlobal = 3
        STR R0, R4, #0   ; (offset = 0)
```

Example (continued)

```
; first statement:
; outLocalA = inLocal & ~inGlobal;
        LDR R0, R5, #0   ; get inLocal
        LDR R1, R4, #0   ; get inGlobal
        NOT R1, R1       ; ~inGlobal
        AND R2, R0, R1   ; inLocal & ~inGlobal
        STR R2, R5, #-1  ; store in outLocalA
                         ; (offset = -1)
```

Example (continued)

```
; next statement:
; outLocalB = (inLocal + inGlobal)
;             - (inLocal - inGlobal);
      LDR R0, R5, #0   ; inLocal
      LDR R1, R4, #0   ; inGlobal
      ADD R0, R0, R1   ; R0 contains (inLocal + inGlobal)
      LDR R2, R5, #0   ; inLocal
      LDR R3, R4, #0   ; inGlobal
      NOT R3, R3
      ADD R3, R3, #1
      ADD R2, R2, R3   ; R2 contains (inLocal - inGlobal)
      NOT R2, R2       ; negate
      ADD R2, R2, #1
      ADD R0, R0, R2   ; (inLocal + inGlobal) - inLocal - inGlobal)
      STR R0, R5, #-2  ; outLocalB (offset = -2)
```

꼭 기억해야 할 것

- variable
  - type
  - scope
- operator
  - precedence
  - associativity
- literal/constant
- expression
  - yields a value after being evaluated
  - has a type
- statement