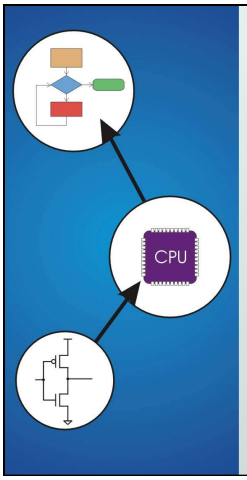


Chapter 12 Variables and Operators

Original slides from Gregory Byrd, North Carolina State University

Modified slides by Chris Wilcox, Colorado State University

Basic C Elements

- **Variables**
  - named, typed data items
- **Operators**
  - predefined actions performed on data items
  - combined with variables to form expressions, statements
- Rules and usage
- Implementation using LC-3 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 processor
  - **int** is supposed to be "natural" integer size, for LC-3 that's 16 bits, LC-3 does not have **double**
  - **int** on a modern processor is usually 32 bits, **double** is usually 64 bits

Variable Names

- Any combination of letters, numbers, and underscore (_)
- **Case matters**
  - "sum" is different than "Sum", this is also true of function names
- **Cannot begin with a number**
  - usually variables beginning with underscore are used only in special library routines
- **Only first 31 characters are used**
  - actually that's compiler dependent, so be careful not to create ambiguous variables!

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

- **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 */
'\xA' /* ASCII 10 (0xA) */
```


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 in the program
  - 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 variables are 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 variables using the **operators** provided by the high-level language.
- Variables and operators combine to form **expressions** and **statements**.
- These constructs denote the work to be done by the program.
- Each operator may correspond to many machine instructions.
  - Example: The multiply operator (*) typically requires multiple LC-3 ADD instructions.

Expression

- Any combination of variables, constants, operators, and function calls
  - every expression has a type, derived from the types of its components (according to C typing rules)
- Examples:
  - `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; /* assign product to z */`
  - `y = y + 1; /* after multiplication */`
  - `; /* null statement */`
- Compound statement groups simple statements using braces.
  - syntactically equivalent to a simple statement
  - `{ z = x * y; y = y + 1; }`

Operators

Three things to know about each operator:

- **(1) Function**
  - what does the operator do?
- **(2) Precedence**
  - in which order are operators combined?
  - Example: a * b + c * d" is the same as "(a * b) + (c * d)“
    since multiply has 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 and subtract associate left-to-right

Assignment Operator

- Changes the value of a variable.

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

1. Evaluate right-hand side.
2. Set value of left-hand side variable to result.

Assignment Operator

- All expressions evaluate to a value, even ones with the assignment operator.
- **For assignment, the result is the value assigned.**
  - usually (but not always) the value of right-hand side
  - type conversion might make assigned value different than computed value
- **Assignment associates right to left.**

```
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
+	add	`x + y`	7	l-to-r
-	subtract	`x - y`	7	l-to-r

- All associate left to right.
- `* / %` have higher precedence than `+ -`.
- Full precedence chart on page 602 of textbook

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.
- Shift operations are logical (not arithmetic).
  - Operate on *values* -- neither operand is changed.

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 entire variable (or value) as TRUE (non-zero) or FALSE (zero).
- Result of a logcial operation is always either TRUE (1) or FALSE (0).

Relational Operators

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

- Result is 1 (TRUE) or 0 (FALSE).
- **Note: Don't confuse equality (==) with assignment (=)!**

Special Operators: ++ and --

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

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

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

- Results: **x = 5, y = 5**
  (because x is incremented before assignment)

Practice with Precedence

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

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

- same as:

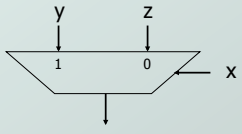
```
x = (a * b) + ((c * d) / 2);
```

- For long or confusing expressions, **use parentheses**, because reader might not have memorized precedence table.
- Note: Assignment operator has lowest precedence, so operations on the right-hand side are evaluated before assignment.

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.



Special Operators: +=, *=, etc.

- 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.

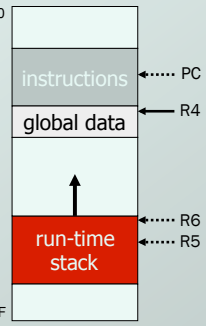
Symbol Table

- 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

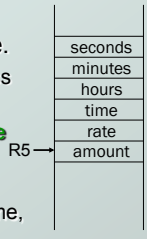
Allocating Space for Variables

- **Global data section**
  - All global variables stored here
    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`



Local Variable Storage

- Local variables are stored in an ***activation record***, also known as a ***stack frame***.
- 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 block is entered.
  - Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.



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	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	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
ADD R1, R0, #1   ; increment
STR R1, R5, #0   ; store

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 is sum
LDR R2, R5, #0  ; inLocal
LDR R3, R5, #0  ; inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3  ; R2 is difference
NOT R2, R2      ; negate
ADD R2, R2, #1
ADD R0, R0, R2  ; R0 = R0 - R2
STR R0, R5, #-2 ; outLocalB (offset = -2)
```