

Chapter 12 Variables and Operators

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

CIT 593

1/31

Basic C Elements

Variables

- A data item upon which the programmer performs an operation
- A named space in memory
- E.g. z, counter

Operators

- Predefined actions performed on data items
- E.g. *, ||, |, &&, &, +

Expressions

- Operators combined with variables/literals to form expressions
- E.g. x * y

Statements

- Expressions group together
- Constitutes a unit of work for high-level language
- Ends with a semicolon. E.g. z = x * y;

CIT 593

2/31

Variable Properties

Identifier: variable name

Type: how data is interpreted, and how much space it needs

Scope: is the region of the program in which the variable is alive and accessible

Storage: how C compiler allocates storage and whether or not the variable loses its value when the block that contains it has completed execution

CIT 593

3/31

Identifier: Variable Names

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

Case sensitive

- "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 definitely used

- Implementations can consider more characters if they like

CIT 593

4/31

Identifier Examples

Legal

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

same identifier

Illegal

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

reserved keyword

CIT 593

5/31

Types

C has several basic data types

```
int    integer (at least 16 bits, commonly 32 bits)
long   integer (at least 32 bits)
float  floating point (at least 32 bits)
double floating point (commonly 64 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 -- 32 bits for most modern processors
- So how do I know the size?
 - Use method called sizeof. E.g. sizeof(int), returns answer in bytes

Signed vs. unsigned:

- Default is 2's complement signed integers
- Use "unsigned" keyword for unsigned numbers

CIT 593

6/31

Additional to Data Type

Literal

- Unnamed constants that appear literally in source code.

Constants

- Variables whose values do not change during the execution of the program
- This done by appending 'const' before the type

Symbols

- Are values defined by using #define
- Values stay constant during single execution of the program but which might be different for different runs or from user to user
 - Like constants but is preprocessor directive

CIT 593

7/31

Literals

Integer

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

Floating point

```
6.023
6.023e23 /* 6.023 x 1023 */
5E12    /* 5.0 x 1012 */
```

Character

```
'c'
'\n' /* newline */
'\xA' /* ASCII 10 (0xA) */
```

CIT 593

8/31

Constants vs. Symbol

```
#define RADIUS 15.0

int main(){
  const double pi = 3.14159;
  double area;
  double circum;

  area = pi * RADIUS * RADIUS;
  circum = 2 * pi * RADIUS;
}
```

Annotations:

- `RADIUS`: symbol
- `pi`: constant
- `3.14159`: literal

CIT 593

9/31

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

CIT 593

10/31

Local vs. Global Example

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

int 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);
}
```

Annotation: → Not the same as `itsLocal`

Output:

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

CIT 593

11/31

Storage Class of a Variable

- Indicates how the C compiler allocates storage
- Whether or not the variable loses its value when the block that contains it has completed execution

Two kinds:

Static

- Retain their values between invocations
- Global variables are static storage class

Automatic

- Lose their values when their block terminates
- Local variables are automatic storage class by default
- Local variable can be made static by placing the keyword static before the variable
 - E.g. static int local

CIT 593

12/31

Tracking Variables

Tracking

- C compiler keeps tracks of variable name (along with additional information) in a symbol table
- This is similar to LC3 assembler's symbol table

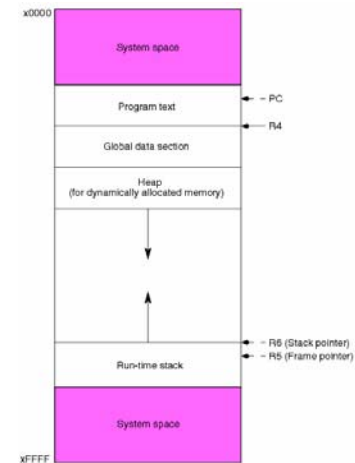
Identifier	Type	Location (offset)	Scope	Other Info

CIT 593

13/31

Memory Allocation in C for Variables

- Global Data section for global variables
- Run-time stack section for local variables
- Heap section for dynamically allocated memory
- Will go in detail how the Memory allocation works in chp 14. But for now skip section 12.5.2...



CIT 593

14/31

Expression

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
```

CIT 593

15/31

Statement

Expresses a complete unit of work

- Executed in sequential order

Simple statement ends with semicolon (note: “;” is not a comment in C)

```
z = x * y; /* assign product to z */
y = y + 1; /* update y */
; /* null statement */
```

Compound statement formed with braces

- Syntactically equivalent to a simple statement
- ```
{ z = x * y; y = y + 1; }
```

CIT 593

16/31

## 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

CIT 593

17/31

## 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.

LC3 code:

```
;assume addr of variable x is in R5 for now
LDR R0, R5, #0
ADD R0, R0, #4
STR R0, R5, #0
```

CIT 593

18/31

## Assignment Operator (contd..)

All expressions evaluate to a value (even ones with the assignment operator i.e.  $y = x = 3$ )

**For assignment, the result is the value assigned**

- Usually (but not always) the value of the right-hand side
  - Type conversion might make assigned value different than computed value e.g.  $\text{int } x = 15.6/3 = 5$

**Assignment associates right to left.**

**y = x = 3;**

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

**y = (x = 3);**

CIT 593

19/31

## 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 + -

Example

- $2 + 3 * 4$  vs.  $(2 + 3) * 4$
- $2 * 4 \% 5$

CIT 593

20/31

## Arithmetic Expressions

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

- Example: `x + 4.3`
- if x is int, converted to double and result is double

Integer division -- fraction is dropped

- Example: `x / 3`
- if x is int and x=5, result is 1 (not 1.666666...)

Modulo -- result is remainder

- Example: `x % 3`
- if x is int and x=5, result is 2

CIT 593

21/31

## Bitwise Operators

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

Operate on variables bit-by-bit

- Like LC-3 AND and NOT instructions

Shift operations

- Operate on *values* -- neither operand is changed
- `x = y << 1` same as `x = y+y`

CIT 593

22/31

## Logical Operators

| Symbol                  | Operation   | Usage                       | Precedence | Assoc  |
|-------------------------|-------------|-----------------------------|------------|--------|
| <code>!</code>          | logical NOT | <code>!x</code>             | 4          | r-to-l |
| <code>&amp;&amp;</code> | logical AND | <code>x &amp;&amp; y</code> | 14         | l-to-r |
| <code>  </code>         | logical OR  | <code>x    y</code>         | 15         | l-to-r |

Treats entire variable (or value) as

- TRUE (non-zero), or
- FALSE (zero)

Result is 1 (TRUE) or 0 (FALSE)

- `x = 15; y = 0; printf("%d", x || y);`

Bit-wise vs Logical

- `1 & 8 = 0` (000001 AND 001000 = 000000)
- `1 && 8 = 1` (True & True = True)

CIT 593

23/31

## ...C and the Right Shift Operator (>>)

Does right shift sign extend or not?

- Answer: Yes and No

Unsigned values: **zero extend**

- `unsigned int x = ~0;`
- Then, `(x >> 10)` will have 10 leading zeros

Signed values:

- "Right shifting a *signed* quantity will fill with sign bits ("arithmetic shift") *on some machines* and with 0-bits ("logical shift") *on others.*" - Kernighan and Ritchie
- In practice, it does sign extend
  - `int x = -0; /* signed */`
  - Then, `(x >> 10)` will still be all 1s

CIT 593

24/31

## 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)

CIT 593

25/31

## Assignment vs. Equality

Don't confuse equality (==) with assignment (=)

```
int x = 9;
int y = 10;
if (x == y) {
 printf("not executed\n");
}
if (x = y) {
 printf("x = %d y = %d", x, y);
}
```

**Result: "x = 10 y = 10" is printed. Why?**

Compiler will not stop you! (What happens in Java?)

CIT 593

26/31

## Special Operators: ++ and --

Changes value of variable before (or after) 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

CIT 593

27/31

## 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)

CIT 593

28/31

## Special Operators: +=, \*=, etc.

Arithmetic and bitwise operators can be combined with assignment operator

| Statement                   | Equivalent assignment          |
|-----------------------------|--------------------------------|
| <code>x += y;</code>        | <code>x = x + y;</code>        |
| <code>x -= y;</code>        | <code>x = x - y;</code>        |
| <code>x *= y;</code>        | <code>x = x * y;</code>        |
| <code>x /= y;</code>        | <code>x = x / y;</code>        |
| <code>x %= y;</code>        | <code>x = x % y;</code>        |
| <code>x &amp;= y;</code>    | <code>x = x &amp; y;</code>    |
| <code>x  = y;</code>        | <code>x = x   y;</code>        |
| <code>x ^= y;</code>        | <code>x = x ^ y;</code>        |
| <code>x &lt;&lt;= y;</code> | <code>x = x &lt;&lt; y;</code> |
| <code>x &gt;&gt;= y;</code> | <code>x = x &gt;&gt; y;</code> |

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

CIT 593

29/31

## Special Operator: Conditional

| Symbol          | Operation   | Usage              | Precedence | Assoc  |
|-----------------|-------------|--------------------|------------|--------|
| <code>?:</code> | conditional | <code>x?y:z</code> | 16         | l-to-r |

`x ? y : z`

- If x is non-zero, result is y
- If x is zero, result is z

Seems useful, but don't use it

- A normal "if" is almost always more clear
- You don't need to use every language feature

CIT 593

30/31

## 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 all the arithmetic operations on the right-hand side are evaluated first

CIT 593

31/31