EECS 22: Advanced C Programming

Lecture 2

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 2: Overview

- Review of the C Programming Language
  - Lexical Elements (Tokens)
  - Keywords
  - Basic Types and Constants
  - Operators and Expressions
    - Arithmetic, Increment, Decrement, Assignment
    - Relational, Logical, Bitwise, Shift, Conditional
    - Others
  - Operator Precedence and Associativity
  - Formatted Input and Output

Review of the C Programming Language

- A C program consists of one or more *translation units* (stored in files)
- A translation unit is formed by a sequence of *tokens*
- Tokens: Lexical Elements
    - Keywords   **int**, **while**, **return**
    - Identifiers   **x**, **MaxValue**, **f**, **main**
    - Constants   **42, 45.0, 123.456e-7, 'x'**
    - String Literals   **"Hello World!\n"**
    - Operators   **+, -, *, /, …**
    - Separators   *white space*, **/* comment */**

Keywords in C

- List of Keywords in ANSI-C

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

- These keywords are reserved!
- These cannot be used as identifiers.
- More keywords are reserved for C++

Identifiers and Separators

- Identifiers
  - Sequence of letters and digits
  - The underscore (_) counts as a letter
  - The first character must be a letter
  - Upper and lower case letters are significant (case-sensitive)
  - Identifiers may have any length
    - However, a compiler implementation may impose length limits
- Separators
  - White space
    - Blanks, tabs, newlines, form feeds
  - Comments
    - Start with `/*` and end with `*/`
    - May extend over multiple lines
    - Do not nest (no comment within a comment, neither in a string)

Basic Types and Constants

- Integer Types
  - **char**   Character, e.g. `'a'`, `'b'`, `'1'`, `'*'`
    - typical range 8 bit `[-128,127]`
  - **short int**   Short integer, e.g. `-7`, `0`, `42`
    - typical range 16 bit `[-32768,32767]`
  - **int**   Integer, e.g. `-7`, `0`, `42`
    - typical range 32 bit `[-2147483648,2147483647]`
  - **long int**   Long integer, e.g. `-99L`, `9L`, `123L`
    - typical range 32 bit `[-2147483648,2147483647]`
  - **long long int** Very long integer, e.g. `12345LL`
    - typical range 64 bit
      `[-9223372036854775808,9223372036854775807]`
- Integer Types can be
  - **signed**   negative and positive values (incl. 0)
  - **unsigned**   positive values only (incl. 0)

Basic Types and Constants

- Integer Constants
  - Decimal representation
    - Sequence of digits **0** to **9**, *not* starting with **0**
    - e.g. **1234567**
  - Octal representation
    - Sequence of digits **0** to **7**, starting with **0**
    - e.g. **0123**   (which is **83** in decimal notation)
  - Hexadecimal representation
    - Sequence of digits **0** to **9** and letters **A** to **F**, starting with **0x**
    - e.g. **0x1A2**   (which is **418** in decimal notation)
  - Suffixes
    - **U** indicates **unsigned** type
    - **L** indicates **long** type, **LL** indicates **long long** type
  - Note: Letters in integer constants are case-insensitive!

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   7

Basic Types and Constants

- ASCII Table: Numerical Representation of Characters
  - American Standard Code for Information Interchange

0 *NUL*	1 *SOH*	2 *STX*	3 *ETX*	4 *EOT*	5 *ENQ*	6 *ACK*	7 *BEL*
8 *BS*	9 *HT*	10 *NL*	11 *VT*	12 *NP*	13 *CR*	14 *SO*	15 *SI*
16 *DLE*	17 *DC1*	18 *DC2*	19 *DC3*	20 *DC4*	21 *NAK*	22 *SYN*	23 *ETB*
24 *CAN*	25 *EM*	26 *SUB*	27 *ESC*	28 *FS*	29 *GS*	30 *RS*	31 *US*
32	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 *DEL*

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   8

Basic Types and Constants

- Character String Constants: "Text strings"
  - Start and end with a double quote character (**"**)
  - May not extend over a single line
  - Subsequent string constants are concatenated
  - Text formatting using *Escape Sequences*
    - `\n`   newline
    - `\t`   horizontal tab
    - `\v`   vertical tab
    - `\b`   back space
    - `\r`   carriage return
    - `\f`   form feed
    - `\a`   alert / bell
    - `\\`   backslash character
    - `\?`   question mark
    - `\'`   single quote
    - `\"`   double quote character
    - `\ooo` octal character, e.g. `\0`
    - `\xhh` hexadecimal character, e.g. `\x41 = A`
  - Example: `"Hello" " \"EECS 22\"!\n"`
  - Note: Strings are of type `const char *`

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   9

Basic Types and Constants

- Floating Point Types
  - `float`   Floating point with single precision
    - Example `3.5f`, `-0.234f`, `10e8f`
  - `double`   Floating point with double precision
    - Example `3.5`, `-0.23456789012`, `10e88`
  - `long double`   Floating point with high precision
    - Example `12345678.123456e123L`

- Floating Point Values are in many cases *Approximations* only!
  - Storage size of floating point values is fixed
  - Many values can only be represented as approximations
  - Example: `1.0/3.0 = .333333`

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   10

Operators and Expressions

- Arithmetic Operators
- Increment and Decrement Operators
- Assignment Operator
- Augmented Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Shift Operators
- Conditional Operator
- Other Operators

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   11

Arithmetic Operators

- Arithmetic Operators
  - parentheses   (,)
  - unary plus, minus   +, -
  - multiplication, division, modulo   *, /, %
  - addition, subtraction   +, -
- Evaluation order of expressions
  - binary operators evaluate left to right
  - unary operators evaluate right to left
  - by operator precedence
    - ordered as in table above (higher operators are evaluated first)
- Arithmetic operators are available
  - for integer types: all
  - for floating point types: all except %

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   12

Increment and Decrement Operators

- Counting in steps of one
  - increment   (add 1)
  - decrement   (subtract 1)

- C provides special counting operators
  - increment operator: **++**
    - **count++**   post-increment   (**count = count + 1**)
    - **++count**   pre-increment   (**count = count + 1**)
  - decrement operator: **--**
    - **count--**   post-decrement   (**count = count - 1**)
    - **--count**   pre-decrement   (**count = count - 1**)

  - Note: Argument must be an integral *lvalue*!
    - **Lvalue**: an expression referring to an object (i.e. variable name)
    - An *lvalue* can be used as the *left* argument for an assignment!

Increment and Decrement Operators

- Difference between Pre- and Post- Operators
  - *pre-* increment/decrement
    - value returned is the incremented/decremented (new) value
  - *post-* increment/decrement
    - value returned is the original (old) value
  - Examples:
    - `int n = 5;`   `int n = 5;`
    - `int x = 0;`   `int x = 0;`
    - `x = n++;`   `x = ++n;`

    - ➢ `x = 5`   ➢ `x = 6`
    - ➢ `n = 6`   ➢ `n = 6`

Assignment Operator

- Assignment operator: =
  - evaluates right-hand argument
  - assigns result to left-hand argument
    - ➢ Evaluation order: right-to-left!
  - Left-hand argument must be a lvalue
  - Result is the new value of left-hand argument
- Example:
  - `int a, b, c;`
  - `int d = 5;   /* initialization,`
              `not an assignment */`
  - `a = 42;      /* assignment */`
  - `b = c = 0;   /* same as b = 0; c = 0; */`

Augmented Assignment Operators

- Augmented assignment operators: `+=`, `*=`, …
  - evaluates right-hand side as temporary result
  - applies operation to left-hand side and temporary result
  - assigns result of operation to left-hand side
    - ➢ Evaluation order: right-to-left!
  - Left-hand argument must be a lvalue
- Example: Counter
  - `int c = 0;`   /* counter starting from 0 */
  - `c = c + 1;`   /* counting by regular assignment */
  - `c += 1;`   /* counting by augmented assignment */
- Augmented assignment operators:
  - `+=, -=, *=, /=, %=, <<=, >>=, |=, ^=, &=`

Relational Operators

- Comparison of values
  - `<`   less than
  - `>`   greater than
  - `<=`   less than or equal to
  - `>=`   greater than or equal to
  - `==`   equal to   (remember, `=` means assignment!)
  - `!=`   not equal to
- Relational operators are defined for all basic types
  - integer   (e.g. `5 < 6`)
  - floating point   (e.g. `7.0 < 7e1`)
- Result type is Boolean, but represented as integer
  - false   `0`
  - true   `1` (or any other value *not* equal to zero)

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   17

Logical Operators

- Operation on Boolean (truth) values
  - `!`   "not"   logical negation
  - `&&`   "and"   logical and
  - `||`   "or"   logical or
- Truth table:

x	y	!x	x && y	x || y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

- Argument and result types are Boolean, but represented as integer
  - false   `0`
  - true   `1` (or any other value *not* equal to zero)

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   18

Logical Operators

- *Lazy* evaluation for logical *and* and logical *or*
  - Evaluation order left-to-right
  - Logical *and* has higher priority than logical *or*
  - Expression evaluation stops as soon as the result is known
    - Logical *and* evaluates right-hand argument only if left-hand is true (1)
    - Logical *or* evaluates right-hand argument only if left-hand is false (0)
  - Example:
    - `v = f() && g() || h();`
    - Function `f()` is called first
    - Function `g()` is called only if `f()` returned `1`
    - Function `h()` is called only if result of `f()&&g()` returned `0`
  - Exercise:
    - Is it possible that only `f()` and `h()` are called?

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   19

Bitwise Operators

- Operators for bit manipulation
  - `&`   bitwise "and"   `0xFF & 0xF0 = 0xF0`
  - `|`   bitwise inclusive "or"   `0xFF | 0xF0 = 0xFF`
  - `^`   bitwise exclusive "or"   `0xFF ^ 0xF0 = 0x0F`
  - `~`   bitwise negation   `~0xF0      = 0x0F`
         (one's complement)
  - `<<`   left shift   `0x0F << 4  = 0xF0`
  - `>>`   right shift   `0xF0 >> 4  = 0x0F`
  - ➢ Bitwise operators are only available for integral types
- Typical usage
  - Mask out some bits from a value
    - `c = c & 0x0F` extracts lowest 4 bits from `char c`
  - Set a set of bits in a value
    - `c = c | 0x0F` sets lowest 4 bits of `char c`

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   20

Shift Operators

- Left-shift operator: **x << n**
  - shifts *x* in binary representation *n* times to the left
  - ➤ multiplies *x* *n* times by 2
  - Examples
    - *2x*   = **x << 1**
    - *4x*   = **x << 2**
    - $x * 2^n$ = **x << n**
    - 2^n = **1 << n**
- Right-shift operator: **x >> n**
  - shifts *x* in binary representation *n* times to the right
  - ➤ divides *x* *n* times by 2
  - Examples
    - *x* / 2 = **x >> 1**
    - *x* / 4 = **x >> 2**
    - $x / 2^n$ = **x >> n**

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   21

Conditional Operator

- Conditional evaluation of values in expressions
- Question-mark operator:
  test **?** *true-value* **:** *false-value*
  - evaluates the *test*
  - if *test* is true, then the result is *true-value*
  - otherwise, the result is *false-value*
- Examples:
  - **(4 < 5) ? (42) : (4+8)**   evaluates to **42**
  - **(2==1+2) ? (x) : (y)**   evaluates to **y**
  - **(x < 0) ? (-x) : (x)**   evaluates to **abs(x)**
- Note: Exactly one of the two cases is evaluated
  - Example: **Test() ? f() : g();**
    If **Test()** returns true, **f()** is called, otherwise **g()**

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   22

Other Operators

- Comma operator: *expr1, expr2*
    - Left-to-right evaluation, result is result of right operand
- Array access operator: *expr1[expr2]*
    - Detailed discussion in Lecture 4
- Function call: *expr1(expr2)*
    - Detailed discussion in Lecture 5
- Member access: *expr1.expr2*,
                                              expr1->expr2
    - Detailed discussion in Lecture 11
- Pointer operators: *&expr*, **expr*
    - Detailed discussion in Lecture 12
- Type casting: *(typename) expr*
    - Detailed discussion in Lecture 18

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   23

Operator Precedence and Associativity

– parenthesis, array/member acc.	`()`, `[]`, `.`, `->`	left to right
– unary operators, pointer op.,	`!`, `~`, `++`, `--`, `+`, `-`, `*`, `&`,	right to left
size of, type cast	`sizeof`, `(typename)`	
– multiplication, division, modulo	`*`, `/`, `%`	left to right
– addition, subtraction	`+`, `-`	left to right
– shift left, shift right	`<<`, `>>`	left to right
– relational operators	`<`, `<=`, `>=`, `>`	left to right
– equality	`==`, `!=`	left to right
– bitwise and	`&`	left to right
– bitwise exclusive or	`^`	left to right
– bitwise inclusive or	`|`	left to right
– logical and	`&&`	left to right
– logical or	`||`	left to right
– conditional operator	`?:`	left to right
– assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, …	right to left
– comma operator	`,`	left to right

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   24

Formatted Input

- Formatted input using `scanf()`
  - standard format specifier for integral values
    - `(unsigned) long long`   `%llu %lld`
    - `(unsigned) long`   `%lu  %ld`
    - `(unsigned) int`   `%u   %d`
    - `(unsigned) short`   `%hu  %hd`
    - `(unsigned) char`   `%c` (reads a character)
  - standard format specifier for floating point values
    - `long double`   `%Lf`
    - `double`   `%lf`
    - `float`   `%f`
  - standard format specifier for character strings
    - `char *`   `%Ns` (e.g. `%20s`)
    - `N` indicates maximum string length accepted!
    - ➢ Never use `%s` (potential buffer overflow)!

Formatted Output

- Formatted output using `printf()`
  - standard format specifier for integral values
    - `(unsigned) long long`   `%llu %lld`
    - `(unsigned) long`   `%lu  %ld`
    - `(unsigned) int`   `%u   %d`
    - `(unsigned) short`   `%hu  %hd`
    - `(unsigned) char`   `%c` (prints a character)
  - standard format specifier for floating point values
    - `long double`   `%Lf`
    - `double`   `%f`
    - `float`   `%f`
  - standard format specifier for character strings
    - `char *`   `%s`
  - standard format specifier for pointers
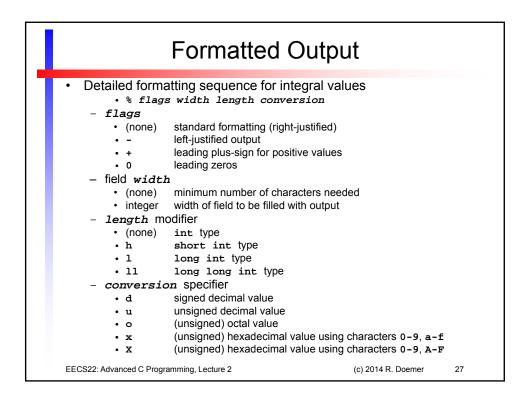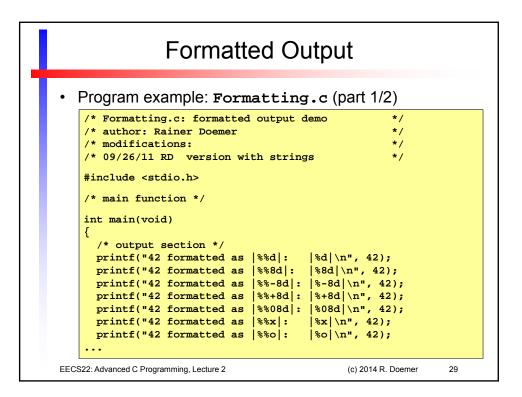   - *pointer*   `%p`

Formatted Output

- Detailed formatting sequence for integral values
  - **% *flags width length conversion***
  - *flags*
    - (none)   standard formatting (right-justified)
    - **-**   left-justified output
    - **+**   leading plus-sign for positive values
    - **0**   leading zeros
  - field *width*
    - (none)   minimum number of characters needed
    - integer   width of field to be filled with output
  - *length* modifier
    - (none)   **int** type
    - **h**   **short int** type
    - **l**   **long int** type
    - **ll**   **long long int** type
  - *conversion* specifier
    - **d**   signed decimal value
    - **u**   unsigned decimal value
    - **o**   (unsigned) octal value
    - **x**   (unsigned) hexadecimal value using characters **0-9**, **a-f**
    - **X**   (unsigned) hexadecimal value using characters **0-9**, **A-F**

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   27

Formatted Output

- Detailed formatting sequence for floating-point values
  - **% *flags width precision length conversion***
  - *flags*
    - (none)   standard formatting (right-justified)
    - **-**   left-justified output
    - **+**   leading plus-sign for positive values
    - **0**   leading zeros
  - field *width*
    - (none)   minimum number of characters needed
    - integer   width of field to be filled with output
  - *precision*
    - (none)   default precision (e.g. 6)
    - **.**int   number of digits after decimal point (for **f**, **e**, or **E**),
                    maximum number of significant digits (for **g**, or **G**)
  - *length* modifier
    - (none)   **float** or **double** type
    - **L**   **long double** type
  - *conversion* specifier
    - **f**   standard floating-point notation (fixed-point)
    - **e** or **E**   exponential notation (using **e** or **E**)
    - **g** or **G**   standard or exponential notation (using **e** or **E**)

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   28

Formatted Output

- Program example: **Formatting.c** (part 1/2)

```
/* Formatting.c: formatted output demo      */
/* author: Rainer Doemer                    */
/* modifications:                           */
/* 09/26/11 RD  version with strings        */

#include <stdio.h>

/* main function */

int main(void)
{
  /* output section */
  printf("42 formatted as |%%d|:    |%d|\n", 42);
  printf("42 formatted as |%%8d|:   |%8d|\n", 42);
  printf("42 formatted as |%%-8d|: |%-8d|\n", 42);
  printf("42 formatted as |%%+8d|: |%+8d|\n", 42);
  printf("42 formatted as |%%08d|: |%08d|\n", 42);
  printf("42 formatted as |%%x|:    |%x|\n", 42);
  printf("42 formatted as |%%o|:    |%o|\n", 42);
...
```

Formatted Output

- Program example: **Formatting.c** (part 2/2)

```
...
  printf("\n");
  printf("123.456 formatted as |%%f|:     |%f|\n", 123.456);
  printf("123.456 formatted as |%%e|:     |%e|\n", 123.456);
  printf("123.456 formatted as |%%g|:     |%g|\n", 123.456);
  printf("123.456 formatted as |%%12.4f|: |%12.4f|\n", 123.456);
  printf("123.456 formatted as |%%12.4e|: |%12.4e|\n", 123.456);
  printf("123.456 formatted as |%%12.4g|: |%12.4g|\n", 123.456);
  printf("\n");
  printf("\"abc\" formatted as |%%12s|:    |%12s|\n", "abc");

  /* exit */
  return 0;
} /* end of main */

/* EOF */
```

Formatted Output

- Example session: **Formatting.c**

```
% vi Formatting.c
% gcc Formatting.c -o Formatting -Wall -ansi
% Formatting
42 formatted as |%d|:    |42|
42 formatted as |%8d|:   |      42|
42 formatted as |%-8d|:  |42      |
42 formatted as |%+8d|:  |     +42|
42 formatted as |%08d|:  |00000042|
42 formatted as |%x|:    |2a|
42 formatted as |%o|:    |52|

123.456 formatted as |%f|:     |123.456000|
123.456 formatted as |%e|:     |1.234560e+02|
123.456 formatted as |%g|:     |123.456|
123.456 formatted as |%12.4f|: |    123.4560|
123.456 formatted as |%12.4e|: |  1.2346e+02|
123.456 formatted as |%12.4g|: |       123.5|

"abc" formatted as |%12s|:     |         abc|
%
```

EECS22: Advanced C Programming, Lecture 2   (c) 2014 R. Doemer   31