COMSW 1003-1

Introduction to Computer Programming in C

Lecture 5

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html
Announcements

• Exercise 1 solution out
• Exercise 2 out
• Read PCP Ch 6
Today

• Review of operators and printf()
• Binary Logic
• Arrays
• Strings
Review : printf

- **printf** is a function used to print to standard output (command line).

- **Syntax:**
  ```c
  printf("format1 format2 ...", variable1, variable2,...);
  ```

- **Format characters:**
  - `%d` or `%i` integer
  - `%f` float
  - `%lf` double
  - `%c` char
  - `%u` unsigned
  - `%s` string

- **Format**
  ```c
  %0n1. n2 t
  ```
  - `n1`: number of digits before the decimal point
  - `n2`: number of digits after the decimal point
  - `t`: type
    - `pad with zeros (optional)`
Review : printf

```
#include <stdio.h>

int main() {

    int a,b;
    float c,d;
    a = 15;
    b = a / 2;

    printf("%d\n",b);
    printf("%3d\n",b);
    printf("%03d\n",b);

    c = 15.3;
    d = c / 3;
    printf("%3.2f\n",d);

    return(0);
}
```

Output:

```
7
7
007
5.10
```
Review: printf

Escape sequences

\n  newline
\t  tab
\v  vertical tab
\f  new page
\b  backspace
\r  carriage return
Binary Logic
- In binary logic, variables can have only 2 values:
  - True (commonly associated with 1)
  - False (commonly associated with 0)

- Binary Operations are defined through TRUTH TABLES

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**AND**
\[ v = x \& y \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NOT**
\[ v = \lnot x \]

<table>
<thead>
<tr>
<th>x</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR**
\[ v = x \mid y \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**EXOR**
\[ v = x \pentodegree y \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion

\[ 6_{10} = 110_{2} \]
Binary Logic

- $1 = \text{true}, \ 0 = \text{false}$
- Decimal to binary conversion
  
  $$6_{10} = 110_2$$
  
  Divide by 2
  
  $$\begin{array}{c|cc}
  \text{Dividend} & 6 & 0 \\
  \hline
  \text{Quotient} & 3 & 1 \\
  \text{Remainder} & 1 & 1 \\
  \hline
  \text{Quotient} & 1 & 0 \\
  \text{Remainder} & 0 & 0
  \end{array}$$

- **AND**
  $$v = x \& y$$

- **OR**
  $$v = x \mid y$$
Binary Logic

- 1 = true, 0 = false

- Decimal to binary conversion

\[ 6_{10} = 110_2 \]

Most significant bit

Least significant bit

- Binary to decimal conversion

\[ 11001_2 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 25 \]
Binary Logic

- 1 = true, 0 = false
- Decimal to binary conversion
  \[6_{10} = 110_2\]
  Most significant bit
  Least significant bit
- Binary to decimal conversion
  \[11001_2 = 1\times2^0 + 0\times2^1 + 0\times2^2 + 1\times2^3 + 1\times2^4 = 25\]
- AND
  \[v = x \& y\]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
  \end{array}
  \]
- OR
  \[v = x | y\]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
  \end{array}
  \]
- NOT
  \[v = !x\]
  \[
  \begin{array}{cc}
  x & v \\
  0 & 1 \\
  1 & 0 \\
  \end{array}
  \]
- EXOR
  \[v = x ^ y\]
  \[
  \begin{array}{ccc}
  x & y & v \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
  \end{array}
  \]
Review: Operators

- **Assignment**: `=`
- **Arithmetic**: `*` `/` `%` `+` `-`
- **Increment**: `++` `- -` `+=` `-=`
- **Relational**: `<` `<=` `>` `>=` `==` `!=`
- **Logical**: `&&` `||` `!`
- **Bitwise**: `&` `|` `~` `^` `<<` `>>`
- **Comma**: `,`
Operators - Bitwise

- Work on the binary representation of data
- Remember: computers store and see data in binary format!

```c
int x, y, z, t, q, s, v;

x = 3; // 00000000000000000000000000000011
y = 16; // 00000000000000000000000000010000

z = x << 1; equivalent to z = x \times 2^1
00000000000000000000000000000110

// XOR
v = x ^ y;
00000000000000000000000000010011
```

```c
v = x \wedge y;

\quad \downarrow \quad \text{XOR}
```

```c
t = y >> 3; equivalent to t = y \cdot 2^{-3}
00000000000000000000000000010011

q = x \& y;
00000000000000000000000000010000

s = x | y;
00000000000000000000000000010011
```

```c
10011
```

```c
00000000000000000000000000000010
```

```c
00000000000000000000000000000000
```

```c
00000000000000000000000000010011
```

```c
00000000000000000000000000010011
```

```c
00000000000000000000000000010011
```

```c
00000000000000000000000000010011
```

```c
00000000000000000000000000010011
```
Operators - Arithmetic

• Arithmetic operators have a **precedence**
  ```c
  int x;
  x = 3 + 5 * 2 - 4 / 2;
  ```

• We can use parentheses () to impose our precedence order
  ```c
  int x;
  x = (3 + 5) * (2 - 4) / 2;
  ```

• % returns the module (or the remainder of the division)
  ```c
  int x;
  x = 5 % 3;  // x = 2
  ```

• We have to be careful with integer vs. float division: remember automatic casting!
  ```c
  int x = 3;
  float y;
  y = x / 2;  // y = 1.00
  ```

Possible fixes:
1) `float x = 3;`  
2) `y = (float) x / 2;`  
Then `y = 1.50`

```c
float y;

Possible fix: y = 1.0/2;
Then y = 0.50
```
Operators – Increment/Decrement

### Syntax

<table>
<thead>
<tr>
<th>Increment</th>
<th>Decrement</th>
<th>Add</th>
<th>Subtract</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>--</td>
<td>+=</td>
<td>-=</td>
</tr>
</tbody>
</table>

```c
int x = 3, y, z;

x++;  // x is incremented at the end of statement

++x;  // x is incremented at the beginning of statement

y = ++x + 3;  // x = x + 1;  y = x + 3;

z = x++ + 3;  // z = x + 3;  x = x + 1;

x -= 2;  // x = x - 2;
```
Operators - Relational

- Return 0 if statement is false, 1 if statement is true

```c
int x = 3, y = 2, z, k, t;

z = x > y;       // z = 1
k = x <= y;      // k = 0

z = x > y;       // z = 1
k = x <= y;      // k = 0
```

16
Operators - Logical

- A variable with value 0 is false, a variable with value !=0 is true

```c
int x = 3, y = 0, z, k, t, q = -3;

z = x && y;   // z = 0;   x is true but y is false

k = x || y;   // k = 1;   x is true

t = !q;       // t = 0;   q is true
```
Arrays

• “A set of consecutive memory locations used to store data” [PCP, Ch 5]

```c
int X[4];  // a vector containing 4 integers
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Address  

n  
n+4  
n+8  
n+12

• Indexing starts at 0!

```c
X[0] = 3;
X[2] = 7;
```

• Be careful not to access uninitialized elements!

```c
int c = X[7];
gcc will not complain about this, but the value of x is going to be random!
```
Arrays

• Multidimensional arrays

```c
int arr[4][3]; // a matrix containing 4x3 = 12 integers
```

<table>
<thead>
<tr>
<th>arr[0][0]</th>
<th>arr[0][1]</th>
<th>arr[0][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[1][0]</td>
<td>arr[1][1]</td>
<td>arr[1][2]</td>
</tr>
<tr>
<td>arr[2][0]</td>
<td>arr[2][1]</td>
<td>arr[2][2]</td>
</tr>
<tr>
<td>arr[3][0]</td>
<td>arr[3][1]</td>
<td>arr[3][2]</td>
</tr>
</tbody>
</table>

• Indexing starts at 0!

```c
arr[0][0] = 1;
arr[3][1] = 7;
```

• Initialize arrays

```c
int X[4] = { 3, 6, 7, 89};
int Y[2][4] = { {19, 2, 6, 99}, {55, 5, 555, 0} };
int Arr[] = { 3, 6, 77};
```

This automatically allocates memory for an array of 3 integers
Strings

• Strings are arrays of `char`
• ‘\0’ is a special character that indicates the end of a string

```c
char s[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

We need 6 characters because there is ‘\0’

```c
char s[10] = "Hello";
```

• Difference between string and char

```c
char c = 'a';
char s[2] = "a";
```
Strings functions

String specific functions are included in the library `string.h`

#include <string.h>

char s[6];
s = "Hello";

Illegal! String assignment can be done only at declaration!

• `strcpy()` : copy a string to another

```
strcpy( string1 , string2 );
```

Copy string2 to string1

char s[6];
strcpy(s, "Hello");
String functions

String specific functions are included in the library `string.h`

- `strcmp()`: compare two strings

```
strcmp( string1 , string2 );
```

Returns:
- 0 if `string1` and `string2` are the same value
- != 0 otherwise

```c
char s1[] = "Hi";
char s2[] = "Him";
char s3[3];
strcpy( s3, s1 );
int x = strcmp( s1, s2 );  // x != 0
int y = strcmp( s1, s3 );  // y = 0
```
Strings functions

String specific functions are included in the library `string.h`

- `strcat()` : concatenate two strings
  ```c
  strcat(string1, string2);
  ```
  Concatenate `string2` at the end of `string1`

  ```c
  char s1[] = "Hello ";
  char s2[] = "World!";
  strcat(s1, s2);
  ```

- `strlen()` : returns the length of a string (does not count `\0`)
  ```c
  strlen(string);
  ```

  ```c
  char s1[] = "Hello";
  int x = strlen(s1);  // x = 5
  ```
Reading Strings

Use functions from library stdio.h

• fgets() : get string from standard input (command line)

```c
char s1[100];
fgets( s1, sizeof(s1), stdin);
```

Reads a maximum of `sizeof(name)` characters of a string from stdin and saves them into string `name`

NOTE: fgets() reads the newline character ‘\n’, so we should substitute it with ‘\0’;
```
name[strlen(name)-1] = '\0';
```

• sizeof() : returns the size (number of bytes occupied in memory) of a variable (for strings it counts the number of elements, including ‘\0’)

```c
name[1] = 'H'
name[2] = 'e'
name[3] = 'l'
name[4] = 'l'
name[5] = 'o'
name[6] = '\0'
```
Reading numbers – Option 1

• First, read a string
• Then, convert string to number
• `scanf()` : get string from standard input (command line)

```c
char s1[100];
int x, y;
printf(“Please enter two numbers separated by a space\n”);
fgets(s1, sizeof(s1), stdin);

User enters: 3 18

scanf( s1, “%d %d”, &x, &y );

// x = 3; y = 18;
```
Reading numbers – Option 2

• Read directly the number
• `scanf()` : get string from standard input (command line) and automatically convert into a number

```c
int x, y;
printf("Please enter two numbers separated by a space\n")
scanf("%d %d", &x, &y);
// x = 3; y = 18;
```
Strings functions - recap

```c
char s1[] = "Hello";
char s2[] = "He";
int x;
char c;

• `strcmp(s1, s2)`
  ```c
  x = strcmp(s1, s2) // x != 0
  ```

• `strcpy(s1, s2)`
  ```c
  strcpy(s2, s1); // s2 = "Hello"
  ```

• `strcat(s1, s2)`
  ```c
  strcat(s2, s1); // s2 = "HelloHello"
  ```

• `strlen(s)`
  ```c
  x = strlen(s1); // x = 5;
  ```

• `sizeof(s)`
  ```c
  x = sizeof(s1); // x = 6;
  ```

• `fgets(s, sizeof(s1), stdin)`
  ```c
  fgets(s1, sizeof(s1), stdin);
  ```
  User enters "7R"

• `sscanf(s, "%d%c", &var)`
  ```c
  sscanf(s1, "%d%c", &x, &c);
  ```
  // x = 7; c = 'R';
```
Read PCP Ch 6
Homework 1 review

HOW TO COMPRESS/UNCOMPRESS folders in UNIX

• Compress folder ~/COMS1003/HW1 to HW1.tar.gz

  `tar -zcvf HW1.tar.gz ~/COMS1003/HW1`

• Uncompress HW1.tar.gz to folder ~/COMS1003/HW1new

  `tar -zxvf HW1.tar.gz -C ~/COMS1003/HW1new`

  (note: ~/COMS1003/HW1new must exist already)