CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Spring 2015

Announcements

• Lab 2 is out
• X Drive issues
Functions

Declaration and Use: Example 1

```c
int a = 5, b = 10, c;

int Maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}

int main(void)
{
    c = Maximum(a, b);
    printf("The max is %d\n", c)
}
```

Function is declared and defined before it is used in main()

Functions

Declaration and Use: Example 2

```c
int a = 5, b = 10, c;

int Maximum(int x, int y);

int main(void)
{
    c = Maximum(a, b);
    printf("The max is %d\n", c)
}

int Maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}
```

Function is declared with prototype before use in main() and defined after it is used in main()
printf()  
Standard Library Function

- Used to write text to the "standard output"
- Normally a computer monitor or printer
- Often the UART in embedded systems
- SIM Uart1 window in MPLAB X

```
printf("Hello, world\n"аратa\nреверсивный\nтранспонировать\nбыть\nникто\nУкраин
```
printf()  
Gotchas

- The value displayed is interpreted entirely by the formatting string:
  \texttt{printf("ASCII = \%d", 'a');}
  \textbf{will output:} ASCII = 97

- A more problematic string:
  \texttt{printf("Value = \%d", 6.02e23);}
  \textbf{will output:} Value = 26366

- Incorrect results may be displayed if the format type doesn't match the actual data type of the argument

\[\text{Maxwell James Dunne – Spring 2015} \quad \text{CMPE-013/L: "C" Programming}\]

printf()  
Output buffer

- printf() operates on lines of text.
- Output text may not be transmit until a newline is sent.

Example

\texttt{printf("a");}

\textbf{Output:}
printf()  
Output buffer

- printf() operates on lines of text.
- Output stored in a buffer until a newline triggers transmission.

Example

```c
printf("a\n");
```

Output:

"a\n"

printf()  
The output buffer

```
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

stdio.h

output buffer

UART
printf()  
Format specifiers
%
[flags][width][.precision][size]type

• Flags – Special printing options
• Width – The minimum size (in chars) of the output
• Precision – Field width
• Size – Convert from base types to longer/shorter types
• Type – The base variable type

printf()  
Format specifiers
%
[flags][width][.precision][size]type

• Flags – Special printing options
  – ‘-’ -> Left justify
  – ‘0’ -> Pad with zeros
  – ‘+’ -> Output ‘+’ for positive values
  – ‘ ’ -> Don’t output a sign symbol
  – ‘#’ -> Prefix integer value based on output type
\textbf{printf()} \\
\textit{Format specifiers}

\%[flags][width][.precision][size]type

- **Width** – The minimum size (in chars) of the output
  - Output is padded
  - ‘0’ flag specifies padding with ‘0’s instead of ‘ ‘s
**printf()**

Format specifiers

`%[flags][width][.precision][size]type`

- **Size** – Convert from base types to longer/shorter types
  - ‘h’ -> Converts to short
  - ‘l’ -> Converts to long/double
  - ‘ll’ -> Converts to long long/long double

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Single character</td>
</tr>
<tr>
<td>s</td>
<td>String (all characters until ‘\0’)</td>
</tr>
<tr>
<td>d</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer with lowercase digits (1a5e)</td>
</tr>
<tr>
<td>X</td>
<td>As x, but with uppercase digits (e.g. 1A5E)</td>
</tr>
<tr>
<td>f</td>
<td>Signed decimal value (floating point)</td>
</tr>
<tr>
<td>e/E</td>
<td>Signed decimal with exponent (e.g. 1.26e-5)</td>
</tr>
<tr>
<td>p</td>
<td>A pointer value indicating a memory address</td>
</tr>
<tr>
<td>g/G</td>
<td>As e or f, but depends on size and precision of value</td>
</tr>
<tr>
<td>8</td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
printf()
Format String Examples

• Print a hexadecimal:

```c
printf("0x806x\n", x);
```

0  Any unused spaces will be filled with zeros
6  Specifies that 6 characters must be output (including 0x prefix)

871587  0x0d4ca3

• Printing a double:

```c
printf("f = 806.3f\n", f);
```

0  Any unused spaces will be filled with zeros
6  Specifies that 6 characters must be output
.3  Specifies that 3 decimal places will be output

3.3  03.300
printf() 
Format String Examples

• Printing a double:

```c
printf("%.1f\n", percentCorrect);
```

.1 Specifies that 1 decimal place will be output

`%` Outputs a literal `%`

97.322 97.38

printf() 
Format String Examples

• Printing a double:

```c
printf("%.1f\n", (double)percentCorrect);
```

.1 Specifies that 1 decimal place will be output

`%` Outputs a literal `%`

97.322 97.38
**scanf()**
Standard Library Function

- Used to read input from the "standard input"
- Normally a keyboard or file
- Often the UART in embedded systems
- Input file in the simulator
- Entire family of functions:
  - `sscanf()` reads from a string
  - `fscanf()` reads from a file

---

**scanf()**
Standard Library Function

**Syntax**

```c
int scanf(FormatString, arg1, ..., argn);
```

- The format string tells `scanf` what kind of input.
- `arg1` through `argn` are **POINTERS** to variable of the right type.

**Example**

```c
int a, b;
printf("Input a and b\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b);
```
scanf ()

Gotchas

- Ignores blanks and tabs in format string
- Skips over white space (blanks, tabs, newline) as it looks for input
- Returns number of successful conversions
- Arguments **must** be pointers to variable types
- Arguments not processed in the input will be left in the input buffer.

scanf ()

The input buffer

stdio.h

output buffer

Input buffer

UART
**scanf()**

Standard Library Function

```c
int a, b;
printf("Input a and b\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "3140 56\n"
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "77 -3\n"
```

**scanf()**

The input buffer

```
  3 1 4 0  5 6 \n
```

```
scanf("%d %d", &a, &b)
```

```
  3 1 4 0  5 6 \n
```

a = 3140, b = 56
`scanf()`

The input buffer

\n  7  7  -  3  \n\n
`scanf("%d %d", &a, &b)`

Nothing!

`scanf("%d %d%c", &a, &b, &c)`

```
3 1 4 0 5 6 \n```

`a = 3140, b = 56`
**scanf()**

Format specifiers

`%[*][width][modifier]type`

- `*` – Ignores this field
- `Width` – The maximum number of characters to match
- `Modifier` – Convert from base types to longer/shorter types
- `Type` – The base variable type

**scanf()**

Examples

- Read input line with date in the format:
  - 25/12/2012

```c
scanf("%d/%d/%d", &day, &month, &year);
```

- `day` int, `&day` is pointer to day
- `month` int, `&month` is pointer to month
- `year` int, `&year` is pointer to year
**scanf()**

Examples

- Read input line with date in the format:
  - 25 Dec 2012

```
scanf("%d %s %d", &day, &month, &year);
```

- **day** int, &day is pointer to day
- **month** char[20], is a string for putting the month into, does not need "&" because name of array is already a pointer
- **year** int, &year is pointer to year

---

**scanf()**

Return value

```c
int a, b;
char c;
while (scanf("%d %d%c", &a, &b, &c) != 3) {
    printf("Please enter an integer pair!\n");
}
```
Arrays

Arrays are variables that can store many items of the same type. The individual items known as elements, are stored sequentially and are uniquely identified by the array index (sometimes called a subscript).

- Arrays:
  - May contain any number of elements
  - Elements must be of the same type
  - The index is zero based
  - Array size (number of elements) must be specified at declaration

Arrays

How to Create an Array

Arrays are declared much like ordinary variables:

**Syntax**

```c
type arrayName[size];
```

- size refers to the number of elements
- size can be a constant OR specified at runtime (c99)

**Example**

```c
int a[10];
char s[25];
char str[x];
```
Arrays

How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

**Syntax**

```c
(type arrayName[size] = {item1, ..., itemn});
```

- The items must all match the *type* of the array

**Example**

```c
int a[5] = {10, 20, 30, 40, 50};
char b[5] = {'a', 'b', 'c', 'd', 'e'};
```

---

Arrays

How to Use an Array

Arrays are accessed like variables, but with an index:

**Syntax**

```c
arrayName[index]
```

- *index* may be a variable or a constant
- The first element in the array has an index of 0
- C does not provide any bounds checking

**Example**

```c
int i, a[10]; // An array that can hold 10 integers

for(i = 0; i < 10; i++) {
    a[i] = 0; // Initialize all array elements to 0
}
a[4] = 42; // Set fifth element to 42
```
Arrays
Creating Multidimensional Arrays
Add additional dimensions to an array declaration:

**Syntax**

```c
type arrayName[size_1]...[size_n];
```

- Arrays may have any number of dimensions
- Three dimensions tend to be the largest used in common practice

**Example**

```c
int a[10][10];    // 10x10 array for 100 integers
float b[10][10][10];    // 10x10x10 array for 1000 floats
```

Arrays
Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

**Syntax**

```c
type arrayName[size_0]...[size_n] =
    
    {{item,...,item},
        ...
        {item,...,item}};
```

**Example**

```c
char a[3][3] = {{'X', '0', 'X'},
                {'0', '0', 'X'},
                {'X', 'X', '0'}};

int b[2][2][2] = {{{{0, 1}, {2, 3}}},{{4, 5}, {6, 7}}};
```
Arrays

Visualizing 2-Dimensional Arrays

```c
int a[3][3] = {{0, 1, 2},
               {3, 4, 5},
               {6, 7, 8}};
```

<table>
<thead>
<tr>
<th>Row, Column</th>
<th>a[y][x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td>a[0][0] = 0;</td>
</tr>
<tr>
<td></td>
<td>a[0][1] = 1;</td>
</tr>
<tr>
<td></td>
<td>a[0][2] = 2;</td>
</tr>
<tr>
<td>Row 1</td>
<td>a[1][0] = 3;</td>
</tr>
<tr>
<td></td>
<td>a[1][1] = 4;</td>
</tr>
<tr>
<td></td>
<td>a[1][2] = 5;</td>
</tr>
<tr>
<td>Row 2</td>
<td>a[2][0] = 6;</td>
</tr>
<tr>
<td></td>
<td>a[2][1] = 7;</td>
</tr>
<tr>
<td></td>
<td>a[2][2] = 8;</td>
</tr>
</tbody>
</table>

Arrays

Visualizing 3-Dimensional Arrays

```c
int a[2][2][2] = {{ {0, 1}, {2, 3} },
                   { {4, 5}, {6, 7} }};
```

<table>
<thead>
<tr>
<th>Plane, Row, Column</th>
<th>a[z][y][x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plane 0</td>
<td>a[0][0][0] = 0;</td>
</tr>
<tr>
<td></td>
<td>a[0][0][1] = 1;</td>
</tr>
<tr>
<td></td>
<td>a[0][1][0] = 2;</td>
</tr>
<tr>
<td></td>
<td>a[0][1][1] = 3;</td>
</tr>
<tr>
<td></td>
<td>a[1][0][0] = 4;</td>
</tr>
<tr>
<td></td>
<td>a[1][0][1] = 5;</td>
</tr>
<tr>
<td></td>
<td>a[1][1][0] = 6;</td>
</tr>
<tr>
<td></td>
<td>a[1][1][1] = 7;</td>
</tr>
<tr>
<td>Plane 1</td>
<td>a[2][0][0] = 8;</td>
</tr>
<tr>
<td></td>
<td>a[2][0][1] = 9;</td>
</tr>
<tr>
<td></td>
<td>a[2][1][0] = 10;</td>
</tr>
<tr>
<td></td>
<td>a[2][1][1] = 11;</td>
</tr>
</tbody>
</table>
Arrays
Example of Array Processing

```c
/****************************
 * Print out 0 to 90 in increments of 10
 ****************************/
int main(void)
{
    int i = 0;
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    while (i < 10) {
        a[i] *= 10;
        printf("%d\n", a[i]);
        i++
    }
    while (1);
}
```

Strings
Character Arrays and Strings

**Definition**

Strings are arrays of char whose last element is a null character `\0` with an ASCII value of 0. C has no native string data type, so strings must always be treated as character arrays.

- Strings:
  - Are enclosed in double quotes "string"
  - Are terminated by a null character `\0`
  - Must be manipulated as arrays of characters (treated element by element)
  - May be initialized with a string literal
Strings

Creating a String Character Array

Strings are created like any other array of char:

```c
char arrayName[length];
```

- `length` must be one larger than the length of the string to accommodate the terminating null character '\0'
- A char array with n elements holds strings with n-1 char

```c
char str1[10]; // Holds 9 characters plus '\0'
char str2[6]; // Holds 5 characters plus '\0'
```

Strings

How to Initialize a String at Declaration

Character arrays may be initialized with string literals:

```c
char arrayName[] = "Microchip";
```

- Array size is not required
- Size automatically determined by length of string
- NULL character '\0' is automatically appended

```c
char str1[] = "Microchip"; // 10 chars "Microchip\0"
char str2[6] = "Hello"; // 6 chars "Hello\0"
```

// Alternative string declaration
```c
char str3[] = {'P', 'I', 'C', '\0'};
```