CMPE-013/L

Introduction to “C” Programming

Gabriel Hugh Elkaim

Winter 2015

Functions

Function Definitions: Return Data Type

- The function returns nothing if:
  - The `return` statement has no *expression*
  - The `return` statement is not present at all
- The function should be declared as `void`

Example

```c
void identifier(type_1 arg_1, ..., type_n arg_n)
{
    declarations
    statements
    return;  // return; may be omitted if nothing is being returned
}
```
Functions
Function Definitions: Parameters

- A function's parameters are declared just like ordinary variables, but in a comma delimited list inside the parentheses
- The parameter names are only valid inside the function (local to the function)

Syntax

\[
\text{type identifier(type}_1 \text{ arg}_1, \ldots, \text{type}_n \text{ arg}_n) \\
\begin{align*}
&\{ \\
&\quad \text{declarations} \\
&\quad \text{statements} \\
&\quad \text{return expression;}
\end{align*}
\]

Functions
Function Definitions: Parameters

- Parameter list may mix data types
  - `int Foo(int x, float y, char z)`
- Parameters of the same type must be declared separately – in other words:
  - `int Maximum(int x, y)` will **not** work
  - `int Maximum(int x, int y)` is correct

Example

```c
int Maximum(int x, int y) \\
{ \\
\quad \text{return } (x \geq y) ? x : y;
}
```
Functions
Function Definitions: Parameters

• If no parameters are required, use the keyword `void` in place of the parameter list when defining the function

```
foo();    mybar(a, b)
```

### Example
```
type identifier(void)
{
    declarations
    statements
    return expression;
}
```

Functions
How to Call / Invoke a Function

**Function Call Syntax**

- No parameters and no return value
  ```
  void Foo();
  ```
- No parameters, but with a return value
  ```
  x = Foo();
  ```
- With parameters, but no return value
  ```
  Foo(a, b);
  ```
- With parameters and a return value
  ```
  x = Foo(a, b);
  ```
Functions
Function Prototypes

• Just like variables, a function must be declared before it may be used
• Declaration must occur before main() or other functions that use it
• Declaration may take two forms:
  – The entire function definition
  – Just a function prototype – the function definition itself may then be placed anywhere in the program

Example – Function Prototype 1
```c
int Maximum(int x, int y); // Good
```

– Like the function header, but without the parameter names – only the types need be present for each parameter (bad form!):

Example – Function Prototype 2
```c
int Maximum(int, int); // BAD, VERY BAD
```

Function prototypes may be take on two different formats:
– An exact copy of the function header:
**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);
}
```

**Example 1**

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);
}
```

**Example 2**

Function is **declared** with prototype before use in main().

Function is **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 Uart** window in MPLAB X

**Syntax**

```c
printf(ControlString, arg1, ..., argn);
```

- Everything printed verbatim within string except %d's which are replaced by the argument values from the list

**Example**

```c
int a = 5, b = 10;
printf("a = %d \n b = %d\n", a, b);
```

**Result:**

```
a = 5
b = 10
```

**NOTE:** the 'd' in %d is the *conversion character*. (See next slide for details)
**printf()**

**Gotchas**

- The value displayed is interpreted entirely by the formatting string:
  ```c
  printf("ASCII = %d", 'a');
  ```
  will output: ASCII = 97

- A more problematic string:
  ```c
  printf("Value = %d", 6.02e23);
  ```
  will output: Value = 26366

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

**printf()**

**Output buffer**

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

**Example**

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

Output:

```

```
\textbf{printf()}

Output buffer

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

\textbf{Example}

\begin{verbatim}
printf("a\n");
\end{verbatim}

Output:

\begin{verbatim}
"a\n"
\end{verbatim}
**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

**Flags**

- `-` -> Left justify
- `'0'` -> Pad with zeros
- `'+'` -> Output `+` for positive values
- `' '` -> Don’t output a sign symbol
- `'#'` -> Prefix integer value based on output type
printf()  
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

- **Precision** – Field width
  - For integers, minimum number of digits
  - For floats, number of fractional digits/significant figures
  - For strings, number of characters
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

---

### Conversion Character Meaning

<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>Unaligned hexadecimal integer with lowercase digits</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 (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>%</td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
• Print a hexadecimal:

```c
printf("0x%06x\n", x);
```

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

```
0x000d4ca3
```

871587

0x0d4ca3

• Printing a double:

```c
printf("f = %06.3f\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

```
3.300
```

03.300
• Printing a double:

\[
\text{printf(""%.1f%%\n", percentCorrect);}
\]

.1  Specifies that 1 decimal place will be output

% Outputs a literal ‘%’

97.322  97.3%

\[
\text{printf(""%.1f%%\n", (double)percentCorrect);}
\]

.1  Specifies that 1 decimal place will be output

% Outputs a literal ‘%’  "%.1f\n"

97.322  97.3%
**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

**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.

**stdio.h**

- **output buffer**
- **Input buffer**
- **UART**
**scanf()**

*Standard Library Function*

**Example**

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

```
3 1 4 0 5 6
```

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

```
3 1 4 0 5 6
```

```
a = 3140, b = 56
```

**The input buffer**
The input buffer

```
\n7 7 - 3 \n```

\texttt{scanf("%d %d", \&a, \&b)}

\texttt{Nothing!}

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

\texttt{scanf("%d %d%c", \&a, \&b, \&c)}

\texttt{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

**Examples**

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

  ```
  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

```c
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");
}
```

```c
'\n'
```
Arrays

Definition

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

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

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

Example

```
int a[10];   // a[0]
char s[25];  // ...
char str[x]; // str[0] to str[x-1]
```
Arrays

How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

**Syntax**

```
type arrayName[size] = {item_1, ..., item_n};
```

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

**Example**

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

Arrays are accessed like variables, but with an index:

**Syntax**

```
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**

```
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**

```
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 may be initialized with lists within a list:

**Syntax**

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

**Example**

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

**Arrays**

**Visualizing 3-Dimensional Arrays**

```c
int a[2][2][2] = {{{0, 1}, {2, 3}},
                   {{4, 5}, {6, 7}}};
```
Arrays
Example of Array Processing

/*****************************/
* 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

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:

**Syntax**

```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

**Example**

```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:

**Syntax**

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

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

**Example**

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

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

Lab Tutoring —
Sara Cadelano — scadelano@usc

T 12:00 - 1:05
T 2 - 3
W 2 - 3

Scan (...) &a, &b
void myVoid (void)

int myAvg (int x, int y);

C = myAvg (a, b)
return myAvg;

STACK