CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Spring 2016
scontf("\%\f")

"foober"

While (getc(stdin) != '\n')

flush(stdin);

LF
3.0 \leq 3
\frac{3.0}{3.0 \cdot 10^3}

- Wall

3 + 4
3 \times 4
3 - 4

Print(f("(0/0 if 0% < %1f)", 3, '+', 4)

Literals & Constants

Literals

Constants
A Simple C Program

Literal Constants

```c
unsigned int a;
unsigned int c;
#define b 2

void main(void)
{
    a = 5;
    c = a + b;
    printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```
Literal Constants

Definition

A literal or a literal constant is a value, such as a number, character or string, which may be assigned to a variable or a constant. It may also be used directly as a function parameter or an operand in an expression.

• Literals
  – Are "hard-coded" values
  – May be numbers, characters or strings
  – May be represented in a number of formats (decimal, hexadecimal, binary, character, etc.)
  – Always represent the same value (5 always represents the quantity five)
Constant vs. Literal

What's the difference?

- Terms are used interchangeably in most programming literature
- A literal is a constant, but a constant is not a literal
  - `#define MAXINT 32767`
  - `const int maxint = 32767;`
- For purposes of this presentation:
  - Constants are labels that represent a literal
  - Literals are values, often assigned to symbolic constants and variables
Literal Constants

- Four basic types of literals:
  - Integer
  - Floating Point
  - Character
  - String

- Integer and Floating Point are *numeric type* constants:
  - Commas and spaces are not allowed
  - Value cannot exceed type bounds
  - May be preceded by a plus or minus sign
Integer Literals

Decimal (Base 10)

- Cannot start with 0 (except for 0 itself)
- Cannot include a decimal point
- Valid Decimal Integers:
  0  5  127  -1021  65535

- Invalid Decimal Integers:
  32,767  25.0  1 024  0552
Integer Literals
Hexadecimal (Base 16)

- Must begin with \texttt{0x} or \texttt{OX}
- May include digits 0-9 and A-F / a-f
- Valid Hexadecimal Integers:
  \texttt{0x 0X1 0x0A2B 0xBEEF}

- Invalid Hexadecimal Integers:
  \texttt{0x5.3 0E1A2 0xE0G 53h}
Integer Literals
Octal (Base 8)

• Must begin with 0
• Only include digits 0 - 7
• Valid Octal Integers:
  0  01  012  073125
• Invalid Octal Integers:
  05.3  0o12  080  530
Integer Literals

Binary (Base 2)

- Must begin with 0b or 0B
- May include digits 0 and 1
- Valid Binary Integers:
  0b  0b1  0b01010011000001111

- Invalid Binary Integers:
  0b1.0  01100  0b12  10b

ANSI C does not specify a format for binary integer literals. However, this notation is supported by almost all compilers.
Integer Literals

Qualifiers

• Like variables, literals may be qualified
• A suffix is used to specify the modifier
  – ‘U’ or ‘u’ for unsigned: 25u
  – ‘L’ or ‘l’ for long: 25L
  – ’F’ or ’f’ for float: 10f or 10.25F
• Suffixes may be combined: 0xF5UL
  – Note: U must precede L
Integer Literals

32-bit

- Numbers without a suffix are assumed to be signed int
- Automatic promotion based on constant type/suffix:

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>
## Integer Literals

### Unqualified

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal</th>
<th>Octal/Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>u/U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>I/L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>long long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>LI/LL</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>
Floating Point Literals

Decimal (Base 10)

- Like integer literals, but the decimal point is allowed
- Exponential notation can be used:
  \[ k \pm n \Rightarrow k \cdot 10^{\pm n} \]
- Valid Floating Point Literals:
  \[ 2.56e-5 \quad 10.4378 \quad 48e8 \quad 0.5 \quad 10f \]
- Invalid Floating Point Literals:
  \[ 0x5Ae-2 \quad \text{02.41} \quad F2.33 \]
Character Literals

- Specified within single quotes (`'`) - \n
- May include any single printable character

- May include any single non-printable character using escape sequences (e.g. `'\0'` = NULL) (also called digraphs)

- Valid Characters: `'a', 'T', '\n', '5', ' @', ' ' ` (space)

- Invalid Characters: `me', '23', '0'`
String Literals

- Specified within double quotes ("")
- May include any printable or non-printable characters (using escape sequences)
- Terminated by a null character \0
- Valid Strings: "Microchip", "Hi\n", "PIC", "2500", "rob@microchip.com", "He said, \"Hi\""
- Invalid Strings: "He said, "Hi""
String Literals

Declarations

- Strings are a special case of **arrays**
- The null character is automatically appended to the end of the string:

Example 1 – Wrong Way

```c
char color[3] = "RED";
```

Is stored as:

```c
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

**NOT** a complete string – no '\0' at end

Example 2 – Right Way

```c
char color[] = "RED";
```

Is stored as:

```c
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```
# String Literals

How to Include Special Characters in Strings

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\000</td>
<td>Octal number</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

- Equivalent characters
  - ' \0', ' \x0'
  - ' \41', ' \x21', '!'
  - ' \144', ' \x64', 'd'

\x7: bell
Hallelujah
Symbolic Constants

Definition

A constant or a symbolic constant is a label that represents a literal. Anywhere the label is encountered in code, it will be interpreted as the value of the literal it represents.

• Constants
  – Once assigned, never change value
  – Make development changes easy
  – Eliminate the use of "magic numbers"
  – Two types of constants
    • Text Substitution Labels
    • const Variables
Symbolic Constants

Text Substitution Labels Using `#define`

- Defines a text substitution label

**Syntax**

```c
#define label text
```

- Each instance of `label` will be replaced with `text` by the **preprocessor** unless `label` is inside a string
- Requires no memory

**Example**

```c
#define PI 3.14159
#define mol 6.02E23
#define MCU "PIC32MX320F128H"
#define COEF 2 * PI
```
Symbolic Constants

#define Gotchas

• Note: a `#define` directive is **NEVER** terminated with a semi-colon `;`, unless you want that to be part of the text substitution.

**Example**

```
#define MyConst 5;

c = MyConst + 3;

c = 5; + 3;
```
Symbolic Constants
Constant Variables Using `const`

- Declaring constants can be done with `const`:
  
  **Example**
  
  ```c
  const float pi = 3.141593;
  ```

- This variable is allocated in program memory, but it cannot be changed due to the `const` keyword.

- In the majority of cases, it is better to use `#define` for constants.
```c
const int x = 3;

int y;

y = y + 3;

t_0 = y

t_1 = x

add t_1, t_1, t_2

#define X3
```
Symbolic Constants

Initializing Variables When Declared

- A constant declared with `const` may not be used to initialize a global or static variable when it is declared (though it may be used to initialize local variables...)

Example

```c
#define CONSTANT1 5
const CONSTANT2 = 10;

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```
Structs
Structures are collections of variables grouped together under a common name. The variables within a structure are referred to as the structure’s **members**, and may be accessed individually as needed.

- **Structures:**
  - May contain any number of members
  - Members may be of **any** data type
  - Allow a group of related variables to be treated as a single unit, even if different types
  - Ease the organization of complicated data
Structures
Declaring

Syntax

```
struct StructName {
    type_1 memberName_1;
    ...
    type_n memberName_n;
};
```

Members are declared just like ordinary variables

Example

```
// Structure to handle complex numbers
struct Complex {
    float re;  // Real part
    float im;  // Imaginary part
};
```
Structures
Instantiating

Syntax

```c
(struct StructName {
    type₁ memberName₁;
    ...
    typeₙ memberNameₙ;
}) varName₁,...,varNameₙ;
```

Example

```c
// Structure to handle complex numbers
struct Complex {
    float re;
    float im;
} x, y; // Declare x and y of type complex
```
If `StructName` has already been defined:

```c
struct StructName varName_1, ..., varName_n;
```

Example:

```c
struct Complex {
    float re;
    float im;
}
...
struct Complex x, y;  // Declare x and y of type complex
```
Structures
Accessing members

Syntax

`structVariableName.memberName`

Example

```c
struct Complex {
    float re;
    float im;
} x, y; // Declare x and y of type 'struct complex'

int main(void)
{
    x.re = 1.25; // Initialize real part of x
    x.im = 2.50; // Initialize imaginary part of x
    y = x; // Set struct y equal to struct x
    ...
```
Structures
Initialization

Syntax

If `StructName` has already been defined:

```c
struct  StructName  varName = {const_1, ..., const_n};
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex x = {1.25, 2.50};
```
1) your test harness
    your matrix math

2) your test harness
    your matrix math

3) both of your
Structures
Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{1.2, 7.6}, {38.5, 17.8}};
    ...

    m.a.x =
```
Structures

Nesting Structures

Example

```c
struct point {
    float x;
    float y;
};

struct line {
    struct point a;
    struct point b;
};

int main(void)
{
    struct line m = {{{1.2, 7.6}, {38.5, 17.8}}};
    printf("Line (%f, %f) <-> (%f, %f)",
            m.a.x, m.a.y, m.b.x, m.b.y);
    ...
}
```

```c
struct circle {
    struct point c;
    float r;
}
```
X \text{YZ point}

3

float x;
float y;
float z;

X Y Z point foo[h];

foo[t].X = foo[t][0]
Structures
Arrays and Pointers with Strings

- Strings:
  - May be assigned directly to `char` array member only at declaration
  - May be assigned directly to a pointer to `char` member at any time

**Example: Structure**

```c
struct Strings {
    char a[4];
    char *b;
} str = {"Bad", "Good"};
```

**Example: Initializing Members**

```c
int main(void)
{
    struct Strings str;
    str.a[0] = 'B';
    str.a[1] = 'a';
    str.a[2] = 'd';
    str.a[3] = '\0';
    str.b = "Good";
}
```
**Structures**

Creating Arrays of Structures

**Syntax**

If `StructName` has already been defined:

```c
struct  StructName arrName[n];
```

**Example**

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3];
```
Structures
Initializing Arrays of Structures at Declaration

Syntax

If `StructName` has already been defined:

```c
struct StructName arrName[n] = {{list_1}, ..., {list_n}};
```

Example

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex a[3] = {{1.2, 2.5}, {3.9, 6.5}, {7.1, 8.4}};
```
Structures
Using Arrays of Structures

If `arrName` has already been defined:

**Syntax**

```
arrName[n].memberName
```

**Example: Definitions**

```c
typedef struct {
    float re;
    float im;
} Complex;
...;
struct Complex a[3];
```

**Example: Usage**

```c
int main(void)
{
    a[0].re = 1.25;
    a[0].im = 2.50;
    ...
}
```
Structures
Creating a Pointer to a Structure

Syntax

If *StructName* has already been defined:

```
struct  StructName  *ptrName;
```

Example

```
struct  Complex  {
    float  re;
    float  im;
};
...
struct  Complex  *a;
```
Structures

How to Use a Pointer to Access Structure Members

If `ptrName` has already been defined:

**Syntax**

```
ptrName->memberName
```

Pointer must first be initialized to point to the address of the structure itself: `ptrName = &structVariable;`

**Example: Definitions**

```c
struct Complex {
    float re;
    float im;
};
...
struct Complex x;
struct Complex *p;
```

**Example: Usage**

```c
int main(void)
{
    p = &x;
    // Set x.re = 1.25 via p
    p->re = 1.25;
    // Set x.im = 2.50 via p
    p->im = 2.50;
}
```
Structures
How to Pass Structures to Functions

Example

typedef struct {
    float re;
    float im;
} Complex;

void Display(struct Complex x)
{
    printf("(%f + j%f)\n", x.re, x.im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(a);
    Display(b);
}
Structures

How to Pass Structures to Functions

Example

typedef struct {
    float re;
    float im;
} Complex;

void Display(struct Complex *x)
{
    printf("(%f + j%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
Structures
How to Pass Structures to Functions

Example

typedef struct {
    float re;
    float im;
} Complex;

void Display(const struct Complex *x)
{
    printf("(\%f + j\%f)\n", x->re, x->im);
}

int main(void)
{
    struct Complex a = {1.2, 2.5};
    struct Complex b = {3.7, 4.0};

    Display(&a);
    Display(&b);
}
struct complex
typedef
typedef

- Assign new names to existing datatypes
- Interpreted by the compiler (unlike \texttt{#define})

**Syntax**

```c
typedef datatype \texttt{typeName};
```

- `typedef int Length;
- typedef float single;
  Length x;
```
typedef

How to Create a Structure Type with typedef

Syntax

typedef struct 
    { 
        type1 memberName1;
        ...
        typen memberName_n;
    } TypeName;

Example

// Structure type to handle complex numbers
typedef struct 
    { 
        float re;  // Real part
        float im;  // Imaginary part
    } Complex;

Complex x;
typedef
Declaring structs

Example

```c
struct Complex {
    float re;
    float im;
};

struct {
    float re;
    float im;
} Complex;

struct Complex {
    float re;
    float im;
} Complex;

typedef struct {
    float re;
    float im;
} Complex;
```
typedef

Declaring structs

Example

typedef struct Complex {
    float re;
    float im;
} Complex;
Text I/O
Text I/O

- Within `<stdio.h>`:
  - Formatted text: `scanf()`/`printf()`
  - Characters: `getchar()`/`putchar()`
  - Strings/Lines: `fgets()`/`puts()`

  • NEVER EVER EVER USE `gets()`
Text I/O
fgets()

Syntax
```
char *fgets(char *str, int count, FILE *stream);
```

- `str` is where received data is stored
  - Needs to be an array
- `count` is how many characters to process
  - Stops when \n or (count - 1) chars are received
- `stream` is stdin
Text I/O

fgets() example

Example

```c
#include <stdio.h>

int main(void)
{
    // Create enough memory for a 50 char string
    char inputData[50 + 1];

    fgets(inputData, sizeof(inputData), stdin);
}
```
String Processing
String Processing

- Within `<string.h>`:
  - Examination
    - Length: `strlen()`
    - Comparing: `strcmp()`/`strncmp()`
    - Splitting: `strtok()`
  - Manipulation
    - Copying: `strncpy()` *(Don't use `strcpy()`!)*
    - Appending: `strncat()`
char x[4];

x = -1;

char y[

char z[] = "Bod";
String Processing

`strlen()`

**Syntax**

```c
size_t strlen(const char *str);
```

- `str` is the string to calculate the length of
- `size_t` can be treated as an `int`

**Examples**

```c
int x = strlen("My string"); // x = 9

char str[] = "asdf\0"
int y = strlen(str); // y = 4
```
String Processing

`strcmp()`

**Syntax**

```c
int strcmp(const char *s1, const char *s2);
```

- Ignores size of the strings, purely alphabetical comparison
- Return value is > 0 if `s1` alphabetically before `s2`, 0 if they're equal, < 0 if `s2` alphabetically before `s1`

**Examples**

```c
char *s1 = "apple", *s2 = "zed";
int cmpResult = strcmp(s1, s2);
if (cmpResult > 0) {
    printf("apple > zed\n");
} else if (cmpResult == 0) {
    printf("apple == zed\n")
} else {
    printf("apple < zed\n");
}
```
String Processing

strtok()

**Syntax**

```c
char *strtok(char *s1, const char *s2);
```

- `s1` (input/output), string to be tokenized
  - Will be modified!
- `s2` (input) – Delimiters

**Examples**

```c
char s1[] = "This is an example!";

char *firstToken = strtok(s1, " "); // firstToken = "This"

char *secondToken = strtok(NULL, " "); // secondToken = "is"

char *thirdToken = strtok(NULL, " "); // thirdToken = "an"

char *fourthToken = strtok(NULL, " "); // fourthToken = "example!"
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
```

```
This is an example!
```

s1
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
```

firstToken

s1

```
This is an example! 
```

Maxwell James Dunne – Spring 2016
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
```

- `firstToken`: "This is an example!"
- `s1`: "This is an example!"
- `secondToken`: Empty string (after processing)
String Processing
strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
```

```
<table>
<thead>
<tr>
<th>firstToken</th>
<th>s1</th>
<th>thirdToken</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>\0 is \0 an \0 example \0</td>
<td>! \0</td>
</tr>
</tbody>
</table>
```

secondToken
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
char *fourthToken = strtok(NULL, " ");
```

```
This is an example!
```

```
\0 is an example!
```

```
This is an example!
```

```
secondToken
```

```
thirdToken
```

```
fourthToken
```
String Processing

strncpy()

Syntax

char *strncpy(char *s1, const char *s2, size_t n);

- **s1** (output) – where the string will be copied to
- **s2** (input) - the string that to be copied
- **n** - how many characters can be copied
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50];
strncpy(s1, "asdf", 4); // s1 = "asdf\0"
strncpy(s1 + strlen(s1), "asdf", 4); // s1 = "asdfsasdf\0"
```
String Processing

strncat()

Syntax

```
char *strncat(char *s1, const char *s2, size_t n);
```

- **s1** (input/output) - is the base string
- **s2** (input) - the string that will be appended
- **n** - how many characters can be appended
- Undefined if s1 and s2 overlap!

Examples

```
char s1[50] = "This is an example!";
strncat(s1, "asdf", 4);
```
String Processing

- Within `<stdlib.h>`:
  - Conversion
    - Integer: `atoi()`, `xtoi()`
    - Floats: `atof()`
- Within `<stdio.h>`:
  - Conversion
    - Any: `sscanf()`
String Processing

atof()

**Syntax**

```c
double atof(const char *s);
```

- **s** (input) – The string to parse
- Returns the converted value or 0.0

**Examples**

```c
char s1[] = "1.03";
double x = atof(s); // y = 1.03

char s2[] = "efg";
double y = atof(s); // y = 0.0
```
Pointers

Pointers and memory
Pointer/array equivalency
Pointer arithmetic
Pointers and the stack
Pointers and strings
Arrays of pointers
Pointers
Address versus value

• In some situations, we will want to work with a variable's address in memory, rather than the value it contains...

```
int x;
```

```
Value of variable x = 0x0123
```

```
Address of variable x = 0x0804
```

32-bit Data Memory (RAM)
Pointers

What are pointers?

- A pointer holds the address of another variable or function
Pointers
What do they do?

- A pointer allows us to indirectly access a variable (just like indirect addressing in assembly language)
Pointers

Why would I want to do that?

• Pointers make it possible to write a very short loop that performs the same task on a range of memory locations / variables.

Example: Data Buffer

```c
// Point to RAM buffer starting address
char *bufPtr = &buffer;

while ((DataAvailable()) && (receivedCharacter != '\0')) {
    // Read byte from UART and write it to RAM buffer
    ReadUart(bufPtr);
    // Point to next available byte in RAM buffer
    bufPtr++;
}
```
Pointers
Why would I want to do that?

Example: Data Buffer
RAM buffer allocated over a range of addresses (perhaps an array)

Pseudo-code:
1. Point arrow to first address of buffer
2. Write data from UART to location pointed to by arrow
3. Move arrow to point to next address in buffer
4. Repeat until data from UART is 0, or buffer is full (arrow points to last address of buffer)
Pointers
Where else are they used?

- Provide method to pass arguments by reference to functions
- Provide method to pass more than one piece of information out of a function
- Another means of accessing arrays and dealing with strings
- Used in conjunction with dynamic memory allocation (creating variables at runtime)
Pointers
How to Create a Pointer Variable

Syntax

```
type *ptrName;
```

- In the context of a declaration, the * merely indicates that the variable is a pointer
- `type` is the type of data the pointer may point to
- Pointer usually described as “a pointer to `type`”

Example

```
int *iPtr;       // Create a pointer to int
int *iPtr, x;    // Create a pointer to int and an int
float *fPtr1, *fPtr2; // Create 2 float pointers
```