C: A High Level Programming Language

- Gives symbolic names to values
  - Don’t need to know which register or memory location

- Provides abstraction of underlying hardware
  - operations do not depend on instruction set
  - example: can write “a = b * c”, even though underlying hardware may not have a multiply instruction
C: A High Level Programming Language

- Provides expressiveness
  - use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time
Compilation vs. Interpretation

- Different ways of translating high-level language

  - Interpretation
    - interpreter = program that executes program statements
    - generally one line/command at a time
    - limited processing
    - easy to debug, make changes, view intermediate results
    - languages: BASIC, LISP, Perl, Java, Matlab, Python
Compilation vs. Interpretation

- **Compilation**
  - translates statements into machine language
  - does not execute, but creates executable program
  - performs optimization over multiple statements
  - change requires recompilation
    - can be harder to debug, since executed code may be different
  - languages: C, C++, Fortran, Pascal, Ada
Compilation vs. Interpretation

• Consider the following algorithm:
  
  Get \( W \) from the keyboard.
  
  \[
  X = W + W \\
  Y = X + X \\
  Z = Y + Y
  \]
  
  Print \( Z \) to screen.

• If interpreting, how many arithmetic operations occur?

• If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?

\[
Z = W \ll 3
\]
Compilation

• **Source Code Analysis**
  – "front end"
  – parses programs to identify its pieces
  – variables, expressions, statements, functions, etc.
  – depends on language (not on target machine)

• **Code Generation**
  – "back end"
  – generates machine code from analyzed source
  – may optimize machine code to make it run more efficiently
  – very dependent on target machine

• **Symbol Table**
  – map between symbolic names and items
  – like assembler, but more kinds of information
“Hello World”

- The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
  
  * Print the words blink on LED
  * hello, world

- This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went.

- With these mechanical details mastered, everything else is comparatively easy.
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n"); // Uses the I/O library to print
    return 0;
}

Embedded C Code

#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");

    while (1); // Loop forever and never return
}

reset

H H H H H
int main(void)
{
    while (1) {
        // Read inputs
        // Perform calculations
        // Update outputs
    }
}
Setting up the IDE
Configuring the Simulator

Set the Debug simulator to wait at the beginning of the main() function.
Resetting MPLAB®X windows

As you will see MPLABX has numerous adjustable windows. New MPLABX users can get a little confused about where and how the set the windows.

If you get confused

**Windows -> Reset Window**

Restores MPLABX Windows back to their original locations
Opening a Project

Select the Open Project button
Opening a Project

1) Navigate to the Project Directory

2) Select the Project

3) Select Open Project
Opening a Project

Project will Open in MPLAB X
Building a Project

To build the project and send it to the Debugger select the **Debug Project Button**
Running the Simulation

To run the project, push the **Continue** button.
Pausing the Simulation

To pause execution of the simulation hit the Pause button.
Windows used in Examples

Variable Window displays a particular set of program variables.

To Open the Variables window:

Select: Windows->Debugging->Variables
Windows used in Examples

Variables Window

**Variable Window** displays several columns of data

You may find it convenient to alter the columns displayed.

“right click” on the column heading
Windows used in Examples

UART1 Output Window

prints out text from C programs

To clear this window:

Right click *inside* of the window then select Clear
Windows used in Examples

Watches Window is similar to the Variables window but displays a different set of data.

To Open the Watches Window:

Select:
Windows->Debugging->Watches
**Windows used in Examples**

**Watches Window**

*Watches Window* needs to be ‘told’ what data to watch.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Decimal</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalVar1</td>
<td>int</td>
<td>0x850</td>
<td>1</td>
<td>0x0001</td>
</tr>
<tr>
<td>globalVar2</td>
<td>int</td>
<td>0x852</td>
<td>2</td>
<td>0x0002</td>
</tr>
<tr>
<td>&lt;Enter new wat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

“Right click” while in the Watches Window to add or delete watches.

**Column configuration is identical to Variables Window**
1 Stop the simulation by pushing Finish Debugger Session button
Closing a Project

2) "Right Click" on the project name then select Close
A Simple C Program

Preprocessor Directives

#include <stdio.h>
#define PI 3.14159

Example

int main(void)
{
    float radius, area;
    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}

Function

Constant Declaration (Text Substitution Macro)

Variable Declarations

Comment
**Comments**

**Definition**

Comments are used to document a program's functionality and to explain what a particular block or line of code does. Comments are ignored by the compiler, so you can type anything you want into them.

- **Two kinds of comments may be used:**
  - Block Comment
    
    /* This is a comment */

  - Single Line Comment
    
    // This is also a comment
Comments
Using Block Comments

• Block comments:
  – Begin with /* and end with */
  – May span multiple lines

```c
/*************************
* Program: hello.c
* Author: R. Ostapiuk
/***************************/
#include <stdio.h>

/* Function: main() */
int main(void)
{
    printf("Hello, world!\n"); /* Display "Hello, world!" */
}
```
Comments
Using Single Line Comments

- Single line comments:
  - Begin with // and run to the end of the line
  - May *not* span multiple lines

```c
#include <stdio.h>

// Function: main()
int main(void)
{
    printf("Hello, world!\n"); // Display "Hello, world!"
}
```
Comments

Nesting Comments

- Block comments may not be nested within other delimited comments
- Single line comments may be nested

Example: Single line comment within a delimited comment.

```c
/*
   code here    // Comment within a comment
 */
```

Example: Delimited comment within a delimited comment.

```c
/*
   code here    /* Comment within a comment
   code here    /* Comment within a comment... oops!
 */
```

- Delimiters don’t match up as intended!
- Dangling delimiter causes compile error
/**
 * @file
 * @author R. Ostapiuk
 * @section DESCRIPTION
 * This is an example Hello World program
 */
#include <stdio.h>

/**
* Main, the entrypoint for this C program.
* @return A success code, where non-zero values indicate failure
*/
int main(void)
{
    int i;        // Loop counter variable
    char *p;      // Pointer to text string

    // Display greeting
    printf("Hello, world!\n");
}
Variable Declarations
Variables and Data Types

A Simple C Program

Example

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    // Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```
Variables

**Definition**

A **variable** is a name that represents one or more memory locations used to hold program data.

- A variable may be thought of as a container that can hold data used in a program

```c
int myVariable;
myVariable = 5;
```
Variables

- Variables are names for storage locations in memory

```c
int warp_factor;
char first_letter;
float length;
```
Variables

- Variable declarations consist of a unique identifier (name)...

```c
int warp_factor;
char first_letter;
float length;
```
Variables

- ...and a **data type**
  - Determines size
  - Determines how values are interpreted

```c
int warp_factor;
char first_letter;
float length;
```
Identifiers

• Names given to program elements:
  – Variables, Functions, Arrays, Other elements

Example of Identifiers in a Program

```c
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```
Identifiers

Valid characters in identifiers:

- First Character
  - '_' (underscore)
  - 'A' to 'Z'
  - 'a' to 'z'

- Remaining Characters
  - '_' (underscore)
  - 'A' to 'Z'
  - 'a' to 'z'
  - '0' to '9'

- Case sensitive!
- Only first 31 characters significant*
ANSI C Keywords

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

- Some compiler implementations may define additional keywords
Data Types

Fundamental Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>single character</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>single precision floating point number</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double precision floating point number</td>
<td>64</td>
</tr>
</tbody>
</table>

The size of an int varies from compiler to compiler.
- XC16 int as 16-bits
- XC32 defines int as 32-bits

If you need precise length variable types, use stdint.h
- uint8_t is unsigned 8 bits
- int16_t is signed 16bits, etc.
Data Type Qualifiers

Modified Integer Types

Qualifiers: unsigned, signed, short and long

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Min</th>
<th>Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char</td>
<td>0</td>
<td>255</td>
<td>8</td>
</tr>
<tr>
<td>char, signed char</td>
<td>-128</td>
<td>127</td>
<td>8</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td>short int, signed short int</td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td>int, signed int</td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>$2^{32}$-1</td>
<td>32</td>
</tr>
<tr>
<td>long int, signed long int</td>
<td>-$2^{31}$</td>
<td>$2^{31}$-1</td>
<td>32</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>0</td>
<td>$2^{64}$-1</td>
<td>64</td>
</tr>
<tr>
<td>long long int, signed long long int</td>
<td>-$2^{63}$</td>
<td>$2^{63}$-1</td>
<td>64</td>
</tr>
</tbody>
</table>
# Data Type Qualifiers

Modified Floating Point Types

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Absolute Min</th>
<th>Absolute Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>$\pm \approx 10^{-44.85}$</td>
<td>$\pm \approx 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>double(^{(1)})</td>
<td>$\pm \approx 10^{-44.85}$</td>
<td>$\pm \approx 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>long double</td>
<td>$\pm \approx 10^{-323.3}$</td>
<td>$\pm \approx 10^{308.3}$</td>
<td>64</td>
</tr>
</tbody>
</table>

MPLAB-X XC32 Uses the IEEE-754 Floating Point Format

\[3.14\] float

\[3\] integer
Variables
How to Declare a Variable

Syntax

```
type identifier_1, identifier_2, ..., identifier_n;
```

- A variable must be declared before it can be used.
- The compiler needs to know how much space to allocate and how the values should be handled.

Example

```
int x, y, z;
float warpFactor;
char text_buffer[10];
unsigned index;
int
```
Variables
How to Declare a Variable

Variables may be declared in a few ways:

Syntax

One declaration on a line
```
type identifier;
```

One declaration on a line with an initial value
```
type identifier = InitialValue;
```

Multiple declarations of the same type on a line
```
type identifier_1, identifier_2, identifier_3;
```

Multiple declarations of the same type on a line with initial values
```
type identifier_1 = Value_1, identifier_2 = Value_2;
```
Variables
How to Declare a Variable

Examples

```c
unsigned int x;
unsigned y = 12;
int a, b, c;
long int myVar = 0x12345678;
long z;
char first = 'a', second, third = 'c';
float big_number = 6.02e+23;
```

It is customary for variable names to be spelled using "camel case", where the initial letter is lower case. If the name is made up of multiple words, all words after the first will start with an upper case letter (e.g. myLongVarName).
Variables
How to Declare a Variable

• Sometimes, variables (and other program elements) are declared in a separate file called a header file.

• Header file names customarily end in .h.

• Header files are associated with a program through the #include directive.
#include Directive

- Three ways to use the `#include` directive:

**Syntax**

```c
#include <file.h>
Look for file in the compiler search path
The compiler search path usually includes the compiler's directory and all of
its subdirectories.
For example: C:\Program Files\Microchip\MPLABX\XC\*.*

#include "file.h"
Look for file in project directory only

#include "c:\MyProject\file.h"
Use specific path to find include file
```
#include Directive

main.h Header File and main.c Source File

```c
#include "main.h"

int main(void)
{
    unsigned int a;
    unsigned int b;
    unsigned int c;
    a = 5;
    b = 2;
    c = a+b;
}
```

The contents of main.h are effectively pasted into main.c starting at the #include directive’s line.
#include Directive

Equivalent main.c File

- After the preprocessor runs, this is how the compiler sees the main.c file.
- The contents of the header file aren't actually copied to your main source file, but it will behave as if they were copied.

```c
#include "header.h"

unsigned int a;
unsigned int b;
unsigned int c;

int main(void)
{
    a = 5;
    b = 2;
    c = a+b;
}
```

Equivalent main.c file without #include
Header Guards
Duplicate #includes

```c
#include "main.h"
#include "main.h"

int main(void)
{
    a = 5;
    b = 2;
    c = a + b;
}
```

The contents of main.h are effectively pasted twice into main.c starting at the #include directive’s line
Header guards
Equivalent main.c File

- Duplicate declarations will occur.
- Which will give compilation errors as there cannot exist multiple declarations of the same variable in the same scope.

```c
#include "main.h"

int main(void)
{
    unsigned int a;
    unsigned int b;
    unsigned int c;

    ... Equivalent main.c file without #include
```
Header guards

Realistic example

```
#include "OledDriver.h"

main.c
#include "OledDriver.h"
#include "Oled.h"
```
Header guards
How do you write/use them

- Declare a macro when a header file is processed.
- Check for that macro before including the code.

```c
#ifndef OLED_H
#define OLED_H

#include "OledDriver.h";

uint q;
...
uint b;
#endif // OLED_H
```
Questions?
Expressions and Control

Maxwell James Dunne
Spring 2016
Expressions

• Represents a single data item (e.g. character, number, etc.)

• May consist of:
  – A single entity (a constant, variable, etc.)
  – A combination of entities connected by operators
    (+, -, *, / and so on)
Expressions

Example

\[ a + b \]
\[ x = y \]
\[ \text{speed} = \frac{\text{dist}}{\text{time}} \]
\[ z = \text{ReadInput}() \]
\[ c \leq 7 \]
\[ x == 25 \]
\[ ++\text{count} \]
\[ d = a + 5 \]
Statements

• Cause an action to be carried out

• Three kinds of statements in C:
  – Expression Statements
  – Compound Statements
  – Control Statements
Expression Statements

- An expression followed by a semi-colon
- Execution of the statement causes the expression to be evaluated

**Examples**

\[
i = 0;
i++;\]
\[
a = 5 + i;\]
\[
y = (m \times x) + b;\]
\[
printf("Slope = %f", m);\]
Compound Statements

- A group of individual statements enclosed within a pair of curly braces { and }
- Individual statements within may be any statement type, including compound
- Allows statements to be embedded within other statements
- Does NOT end with a semicolon after }
- Also called Block Statements
Compound Statements

Example

```c
{
    float start, finish;

    start = 0.0;
    finish = 400.0;
    distance = finish - start;
    time = 55.2;
    speed = distance / time;
    printf("Speed = %f m/s", speed);
}
```
Control Statements

- Used for loops, branches and logical tests
- Often require other statements embedded within them

Example

```c
while (distance < 400.0) {
    printf("Keep running!");
    distance += 0.1;
}
```

(while syntax: `while expr statement`)
Boolean Expressions

- Boolean data type added in C99
- Boolean expressions return integers:
  - 0 expressions evaluate as false
  - non-zero expressions evaluate as true (generally 1)

```c
{  
    int x = 5;  
    bool y, z;  
    y = (x > 4);  
    z = (x > 6);  
    while (1);  
} 
```

- y = true (1)
- z = false (0)
Boolean Expressions

Equivalent Expressions

• If a variable, constant, or function call is used alone as the conditional expression:
  \[(\text{myVar}) \text{ or } (\text{Foo}())\]

• This is the same as saying:
  \[(\text{myVar }\neq 0) \text{ or } (\text{Foo}() \neq 0)\]

• In either case, if \(\text{myVar} \neq 0\) or \(\text{Foo}() \neq 0\), then the expression evaluates as true (non-zero)
**if Statement**

**Syntax**

```plaintext
if (expression) statement
```

- *expression* is evaluated for boolean true (≠0) or false (=0)
- If true, then *statement* is executed

**Note**

Whenever you see *statement* in a syntax guide, it may be replaced by a compound (block) statement.

Remember: spaces and new lines are not significant.
if Statement

Flow Diagram

Syntax

```plaintext
if (expression) statement
```

Start

Expression

- Expression ≠ 0 (true)
- Expression = 0 (false)

Statement

End
if Statement

Example

```c

int x = 5;

if (x) {
    printf("x = %d\n", x);  // If x is TRUE (non-zero)...
}
while (1);

```

- What will print if x = 5? ... if x = 0?
- ...if x = -82?
- ...if x = 65536?
**if** Statement

Testing for TRUE

- **if** (x) **vs. if** (x == 1)
  - if (x) only needs to test for not equal to 0
  - if (x == 1) needs to test for equality with 1
  - Remember: true is defined as non-zero, false is defined as zero

---

**Example: if (x)**

```assembly
if (x)
8:
011B4  E208C2  if (x)
011B6  320004  cp 0.w 0x08c2
```

**Example: if (x == 1)**

```assembly
if (x == 1)
11:
011C0  804610  mov.w 0x08c2, 0x0000
011C2  500FE1  sub.w 0x0000, #1,[0x001e]
011C4  3A0004  bra nz, 0x0011ce
```
int power = 10;
float band = 2.0;
float frequency = 146.52;

if (power > 5) {
    if (band == 2.0) {
        if ((frequency > 144) && (frequency < 148)) {
            printf("Yes, it's all true!\n");
        }
    }
}
**if-else Statement**

**Syntax**

```
if (expression) statement_1
else statement_2
```

- **expression** is evaluated for boolean true (≠0) or false (=0)
- If true, then **statement_1** is executed
- If false, then **statement_2** is executed
**if-else Statement**

Flow Diagram

**Syntax**

```plaintext
if (expression) statement₁
else statement₂
```

[Diagram showing the flow of an if-else statement with decision points and flow paths based on the condition of the expression.]

---

Maxwell James Dunne – Spring 2016
if-else Statement

Example

```c
float frequency = 146.52; // Frequency in MHz

if ((frequency > 144.0) && (frequency < 148.0)) {
    printf("You're on the 2 meter band\n");
} else {
    printf("You're not on the 2 meter band\n");
}
```
if-else if Statement

Syntax

```
if (expression₁) statement₁
else if (expression₂) statement₂
else statement₃
```

- `expression₁` is evaluated for boolean true (≠0) or false (=0)
- If true, then `statement₁` is executed
- If false, then `expression₂` is evaluated
- If true, then `statement₂` is executed
- If false, then `statement₃` is executed
if-else if Statement

Flow Diagram

Syntax

```
if (expression₁) statement₁
else if (expression₂) statement₂
else statement₃
```

START

true

expression₁

false

false

statement₁

statement₂

statement₃

END
if-else if Statement

Example

if ((freq > 144) && (freq < 148)) {
    printf("You're on the 2 meter band\n");
} else if ((freq > 222) && (freq < 225)) {
    printf("You're on the 1.25 meter band\n");
} else if ((freq > 420) && (freq < 450)) {
    printf("You're on the 70 centimeter band\n");
} else {
    printf("You're somewhere else\n");
}
while Loop

Syntax

**while** (expression) **statement**

- If expression is true, statement will be executed and then expression will be re-evaluated to determine whether or not to execute statement again.

- It is possible that statement will never execute if expression is false when it is first evaluated.
while Loop

Flow Diagram

Syntax

while (expression) statement
**while Loop**

**Example**

```c
int i = 0;  // Loop counter initialized outside of loop

while (i < 5) {  // Condition checked at start of loop iterations
    printf("Loop iteration %d\n", i++);
}
```

**Expected Output:**

- Loop iteration 0
- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4
**while Loop**

- Primary looping mechanism
- Completely generic
- Frequently used for main loop of program

**Generic loop:**

```c
while (HaveData()) {
    PrintData();
}
```

**Main loop:**

```c
while (1) {
    ...
}
```
Functions

Program Structure

main()
{
    ...  
    eat();
    ...  
}

be_merry()
{
    ...
    return;
}

eat()
{
    ...
    return;
}

drink()
{
    ...
    be_merry();
    return;
}
Functions
What is a function?

Definition

Functions are self contained program segments designed to perform a specific, well defined task.

- All C programs have one or more functions
- The main() function is required
- Functions can accept parameters from the code that calls them
- Functions return a single value (but can export more data)
- Functions help to organize a program into logical, manageable segments
Functions

Remember Algebra Class?

- Functions in C are conceptually like an algebraic function from math class...

  \[ f(x) = x^2 + 4x + 3 \]

  Function Definition

  Function Parameter

- If you pass a value of 7 to the function: \( f(7) \), the value 7 gets "copied" into \( x \) and used everywhere that \( x \) exists within the function definition: \( f(7) = 7^2 + 4 \times 7 + 3 = 80 \)
Functions

Definitions

Syntax

Data type of
return expression

Name

type identifier(type₁ arg₁,...,typeₙ argₙ)
{
  declarations
  statements
  return expression;
}

Parameter List (optional)

Body

Header

Return Value (optional)
Functions
Function Definitions: Syntax Examples

Example

```c
int Maximum(int x, int y)
{
    int z;

    z = (x >= y) ? x : y;
    return z;
}
```

Example – A more efficient version

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

Function Definitions: Return Data Type

Syntax

```
type identifier(type_1 arg_1, ..., type_n arg_n)
{
    declarations
    statements
    return expression;
}
```

- A function's `type` must match the type of data in the return `expression`
Functions

Function Definitions: Return Data Type

• A function may have multiple return statements, but only one will be executed and they must all be of the same type.

Example

```c
int bigger(int a, int b)
{
    if (a > b) {
        return 1;
    } else {
        return 0;
    }
}
```
Functions

Function Definitions: Return Data Type

• The function type is **void** if:
  – The **return** statement has no **expression**
  – The **return** statement is not present at all

• This is sometimes called a **procedure function** since nothing is returned

---

**Example**

```c
void identifier(type_1 arg_1, ..., type_n arg_n)
{
    declarations
    statements
    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

```c
typedef identifier(type1 arg1,...,type_n arg_n) {
    declarations               Function Parameters
    statements
    return expression;
}
```
Functions

Function Definitions: Parameters

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

Example

```c
int Maximum(int x, int y)
{
    return ((x >= 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

**Example**

```c
type identifier(void)
{
    declarations
    statements
    return expression;
}
```
Functions
How to Call / Invoke a Function

Function Call Syntax

- No parameters and no return value
  \[ \text{Foo}() ; \]

- No parameters, but with a return value
  \[ x = \text{Foo}() ; \]

- With parameters, but no return value
  \[ \text{Foo}(a, b) ; \]

- With parameters and a return value
  \[ x = \text{Foo}(a, b) ; \]
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
Functions

Function Prototypes

- Function prototypes may be take on two different formats:
  - An exact copy of the function header:

  ```c
  int Maximum(int x, int y);
  ```

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

  ```c
  int Maximum(int, int);
  ```
Functions

Declaration and Use: Example 1

**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

Example 2

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

int Maximum(int x, int y);  // Function is declared with prototype before use in main()

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 defined after it is used in main()
}
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Spring 2016
Functions

Declaration and Use: Example 1

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

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** 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 Uart1 window in MPLAB X
printf()  
Standard Library Function

Syntax

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\nb = %d\n", a, b);
```

Result:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

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:
  printf("ASCII = %d", 'a');
  will output: ASCII = 97

A more problematic string:
  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() operates on lines of text.
Output text may not be transmit until a newline is sent.

Example

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

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

`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
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
printf()
Format specifiers

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

- **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
### `printf()`

#### Format specifiers

<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>%</td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
printf()  
Format String Examples

- Print a hexadecimal:

  \[ \text{printf}\left("0x806x\n", \ x\right); \]

  0  Any unused spaces will be filled with zeros

  6  Specifies that 6 characters must be output (including 0x prefix)

  \[ \begin{align*} \text{871587} & \rightarrow & \text{0x0d4ca3} \end{align*} \]
printf()  
Format String Examples

- 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

03.300
printf() Format String Examples

• Printing a double:

```c
printf("%1f88\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 ‘%’
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\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

**Example**

```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

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

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

Nothing!
`scanf()`

The input buffer

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

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

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

\[ a = 3140, \quad b = 56 \]
`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
\textbf{scanf()}  
\textit{Examples}

\begin{itemize}
  \item Read input line with date in the format:
    \begin{itemize}
      \item 25 Dec 2012
    \end{itemize}

\texttt{scanf("%d %s %d", \&day, month, \&year);}

\begin{itemize}
  \item \texttt{day} int, \&day is pointer to day
  \item \texttt{month} char[20], is a string for putting the month into, does not need \texttt{\&} because name of array is already a pointer
  \item \texttt{year} int, \&year is pointer to year
\end{itemize}

\end{itemize}
`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

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**

```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

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

- 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
How to Use an Array

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

```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

```
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

```
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**

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

**Example**

```
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}};
```

```
a[y][x]
```

Row, Column

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

Visualizing 3-Dimensional Arrays

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

Plane, Row, Column

```
a[0][0][0] = 0;
a[0][0][1] = 1;
a[0][1][0] = 2;
a[0][1][1] = 3;
a[1][0][0] = 4;
a[1][0][1] = 5;
a[1][1][0] = 6;
a[1][1][1] = 7;
```
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:

**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'};
```