Unit testing
Unit testing

• Testing portions of code in isolation

• Normally testing is per function

• Requires input and expected output to be known a priori
Unit testing

Rationale

- Find problems early
  - Before integration
- Simplify testing by only testing small, segmented portions of code
- Test functionality that may not be exposed otherwise
- Find documentation errors
Unit testing
Preparing

• The most important question:

"How am I going to test this?"

• Break code into clean functions with:
  – Clear input
  – Clear output
  – No/minimal side effects
Unit testing

Testing architecture

- Known input
- \( f() \)
- Actual output

- Expected output

- Output matches
  - no → Print failure
  - yes → Print success
Unit testing
Testing architecture

Example

```c
// Declare test constants
testInput ← some input
testExpOutput ← precalculated output

// Calculate result
testActOutput ← function result

// Output test results
if testActOutput equals testExpOutput
    output "Test passed"
else
    output "Test failed!"
```
Unit testing
Trivial example

ExampleLib.c

```c
int AddFive(int x)
{
    return x + 5;
}
```

main.c

```c
#include "ExampleLib.h"

int main(void)
{
    // Declare test constants
    int test1Input = 0;
    int test1ExpOutput = 5;

    // Calculate result
    int test1ActOutput;
    test1ActOutput = AddFive(test1Input);

    // Output test results
    if (test1ActOutput == test1ExpOutput) {
        printf("Test1 passed.\n");
    } else {
        printf("Test1 failed!\n");
    }
}
```
Unit testing

Writing tests

• Write multiple tests
  – At least 1 for every group of inputs
  – Each edge case should have their own test

• Each test should check one part of the total functionality
  – One function or logical block of code at a time

Try to break the code you're testing!
Unit testing
Testing framework

- Track how many tests passed/failed
  - Per function
- Track how many functions passed/failed
  - With all tests must pass for the function to pass
- Each test cleanly separated from other tests
  - Both in code and in logic
- Output results
  - Per function/per test results
MMEquals(m0+1, m0+2) \rightarrow

return TRUE;
Parameter passing

Pass by value
Pass by reference
Parameter Passing

By Value

- Parameters passed to a function are generally passed by value.
- Values passed to a function are copied into the local parameter variables.
- The original variable that is passed to a function cannot be modified by the function since the function has a duplicate of the variable, not the original.
Parameter Passing
By Value

Example

```c
int a, b, c;

int Foo(int x, int y)
{
    x = x + (y++);
    return x;
}

int main(void)
{
    a = 5;
    b = 10;
    c = Foo(a, b);
}
```

The value of `a` is copied into `x`.
The value of `b` is copied into `y`.
The function does not change the value of `a` or `b`. 
Parameter Passing

By Value

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Parameter Passing

By Reference

- Parameters can be passed to a function by reference
- Entails passing around memory address
- The original variable that is passed to a function can be modified by the function since the function knows where the data "lives" in memory
Parameter Passing

By Reference

Example function

```c
int Foo(int x[3])
{
    int z = x[2];
    x[1] = 0;
    return z;
}
```

Example main

```c
int main(void)
{
    int a[3] = {6, 19, -1};
    Foo(a);
    while (1);
}
```
Scope
Scope

Variables Declared Within a Function

- Variables declared within a code block are local to that block.

```
int x, y, z;

int Foo(int n)
{
    int a;
    ...
    a += n;
}
```

The `n` refers to the function parameter `n`.
The `a` refers to the `a` declared locally within the function body.
Scope
Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block.

Example

```c
int x;
int Foo(int n)
{
    int a;
    return (a += n);
}
int main(void)
{
    x = Foo(5);  // This will generate an error. 'a' may not be accessed outside of the scope where it was declared.
    x = a;
}
```
Scope
Variables Declared Within a Function

• Variables declared within a block are not accessible outside that block

Example

```c
int x;
int main(void)
{
    {
        int a = 6;
    }

    x = Foo(5);
    x = a;  // This will generate an error. a may not be accessed outside of the scope where it was declared.
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
**Scope**

Global versus Local Variables

**Example**

```c
int x = 5;

int Foo(int y)
{
    int z = 1;
    return (x + y + z);
}

int main(void)
{
    int a = 2;
    x = foo(a);
    a = foo(x);
}
```

- `x` can be seen by everybody
- `foo's local parameter is y`
  - `foo's local variable is z`
  - `foo cannot see main's a`
  - `foo can see x`
- `main's local variable is a`
  - `main cannot see foo's y or z`
  - `main can see x`
Scope
Parameters

"Overloading" variable names:

Declared Locally and Globally

```c
int n;

int Foo(int n)
{
    ... y += n;
    ... local n hides global n
}
```

Declared Globally Only

```c
int n;

int Foo(int x)
{
    ... y += n;
    ...
}
```

A locally defined identifier takes precedence
Scope
Parameters

Example

```c
int n;

int Foo(int n)
{
    y += n;
}

int Bar(int n)
{
    z *= n;
}
```

- Different functions may use the same parameter names.
- The function will only use its own parameter by that name.
Scope
Preprocessor and scoping

Example

```c
#define x 2

void Test(void)
{
    #define x 5
    printf("%d\n", x);
}

void main(void)
{
    printf("%d\n", x);
    Test();
}
```

Result:

```
5
5
```
Storage Class Specifiers
Scope and Lifetime of Variables

• Scope and lifetime of a variable depends on its storage class:
  – Automatic Variables
  – Static Variables
  – External Variables
  – Register Variables

• Scope refers to where in a program a variable may be accessed

• Lifetime refers to how long a variable will exist or retain its value
Storage Class Specifiers

Automatic Variables

- Local variables declared inside a function
  - Created when function called
  - Destroyed when exiting from function
- **auto** keyword *usually* not required – local variables are automatically **auto**
- Typically created on the stack

```c
int Foo(int x, int y)
{
    int a, b;
    ...
```

*Except when the compiler provides an option to make parameters and locals static by default.*
Storage Class Specifiers

auto Keyword with Variables

```c
int Foo(auto int x, auto int y)
{
    ...
}
```

- **auto** is almost never used
- Many books claim it has no use at all
- Some compilers still use **auto** to explicitly specify that a variable should be allocated on the stack when a different method of parameter passing is used by default
Storage Class Specifiers

Static Variables

- Given a permanent address in memory
- Exist for the entire life of the program
  - Created when program starts
  - Destroyed when program ends
- Global variables are always static (cannot be made automatic using `auto`)

```c
int x; // Global variable is always static
int main(void)
{
    ... x = 2;
}
```
Storage Class Specifiers

**static** Keyword with Variables

- A variable declared as **static** inside a function retains its value between function calls (not destroyed when exiting function)
- Function parameters cannot be **static** with some compilers (XC32)

```c
int Foo(int x)
{
    static int a = 0;
    ...
    a += x;
    return a;
}
```

*a will remember its value from the last time the function was called. If given an initial value, it is only initialized when first created – not during each function call.*
Storage Class Specifiers

External Variables

- Variables that are **defined** outside the scope where they are used
- Still need to be **declared** within the scope where they are used
- **extern** keyword used to tell compiler that a variable defined elsewhere will be used within the current scope

External Variable Declaration Syntax:

```c
extern type identifier;
```

External Variable Declaration Example:

```c
extern int x;
```
Storage Class Specifiers

External Variables

- A variable *declared* as `extern` within a function is analogous to a function prototype – the variable may be *defined* outside the function after it is used.

Example

```c
int Foo(int x)
{
    extern int a;
    ...
    return a;
}

int a;
```
A variable *declared* as `extern` outside of any function is used to indicate that the variable is *defined* in another source file – memory only allocated when it's *defined*.
Storage Class Specifiers

Register Variables

• **register** variables are placed in a processor's "hardware registers" for higher speed access than with external RAM
  
  – Common with loop counters

• Not as important when RAM is integrated into processor package (microcontrollers, ...)

• May be done with PIC®/dsPIC®, but it is architecture/compiler specific...
Storage Class Specifiers

Scope of Functions

• Scope of a function depends on its storage class:
  – Static Functions
  – External Functions

• Scope of a function is either local to the file where it is defined (static) or globally available to any file in a project (external)
Storage Class Specifiers

External Functions

- Functions by default have global scope within a project
- `extern` keyword not required, but function prototype is required in calling file

```c
Main.c
int foo(void);

int main(void)
{
    ...  
    x = foo();  
}

SomeFileInProject.c
int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

• If a function is declared as `static`, it will only be available within the file where it was declared (makes it a local function)

```c
Main.c
int foo(void);
int main(void)
{
    ...
    x = foo();
}

SomeFileInProject.c
static int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

- If a variable is declared as `static`, it will only be available within the file where it was declared.

```c
// Main.c
extern int myVar;

int main(void)
{
    ...  // myVar = 6;
}

// SomeFileInProject.c
static int myVar = 0;
```
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Literals & Constants

Literals
Constants
A Simple C Program

Literal Constants

Example

```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
  - **Literal**s 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{0X}
- May include digits 0-9 and A-F / a-f
- Valid Hexadecimal Integers:
  \begin{align*}
  &0x, \quad 0X1, \quad 0x0A2B, \quad 0xBEEF \\
  &\uparrow \quad \uparrow
  \end{align*}

- Invalid Hexadecimal Integers:
  \begin{align*}
  &0x5.3, \quad 0EA12, \quad 0xEG, \quad 53h
  \end{align*}
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 \texttt{0b} or \texttt{0B}
- May include digits 0 and 1
- Valid Binary Integers:
  \begin{verbatim}
  0b    0b1    0b01010011000001111
  \end{verbatim}
- Invalid Binary Integers:
  \begin{verbatim}
  0b1.0   01100   0b12    10b
  \end{verbatim}

\textbf{Warning:}\ ANSI C does \textbf{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

## Unqualified

- 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><strong>int</strong></td>
<td><strong>int</strong></td>
</tr>
<tr>
<td></td>
<td><strong>long int</strong></td>
<td><strong>unsigned int</strong></td>
</tr>
<tr>
<td></td>
<td><strong>long long int</strong></td>
<td><strong>long int</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>unsigned long int</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>long long int</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>unsigned long long int</strong></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 int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned 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>unsigned long long int</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: \(k \times 10^{\pm n}\)
- Valid Floating Point Literals:
  2.56e-5  10.4378  48e8  0.5  10f
- Invalid Floating Point Literals:
  0x5Ae-2  02.41  F2.33
Character Literals

- Specified within single quotes (')
- May include any single printable character
- May include any single non-printable character using escape sequences (e.g. '\0' = NUL) (also called digraphs)
- Valid Characters: 'a', 'T', '\n', '5', '@', ' ' (space)
- Invalid Characters: 'me', ' 23', ''
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:
    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:
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' 
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

\[
x = 28 - y;\\
\]
\[
z = 38 + a;\\
\]
Symbolic Constants
Text Substitution Labels Using \texttt{#define}

- Defines a text substitution label

**Syntax**

\texttt{#define label text}

- Each instance of \texttt{label} will be replaced with \texttt{text} by the \textit{preprocessor} unless \texttt{label} is inside a string
- Requires no memory

**Example**

\texttt{#define PI 3.14159}
\texttt{#define mol 6.02E23}
\texttt{#define MCU "PIC32MX320F128H"}
\texttt{#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

```c
#define MyConst 5;

c = MyConst + 3;

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

- Declaring constants can be done with `const`:
  ```
  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.
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...)

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

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```