Functions

Recursion

- A function can call itself repeatedly.
- Useful for iterative computations (each action stated in terms of previous result).
- Example: Factorials ($5! = 5 \times 4 \times 3 \times 2 \times 1$)

```
long int factorial(int n)
{
    if (n <= 1)
        return(1);
    else
        return(n * factorial(n - 1));
}
```

Evaluation of Recursive Functions

- Evaluation of $5!$ (based on code from previous slide)

<table>
<thead>
<tr>
<th>Recursive iterations of function</th>
<th>Partial results pushed on stack</th>
<th>Factorial term replaced with result of expression above</th>
<th>Result evaluated from TOS downward</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] $1!$</td>
<td>1</td>
<td>$1$</td>
<td>1</td>
</tr>
<tr>
<td>[1] $2!$</td>
<td>$2 \times 1!$</td>
<td>$2 \times 1 = 2$</td>
<td>2</td>
</tr>
<tr>
<td>[2] $3!$</td>
<td>$3 \times 2!$</td>
<td>$3 \times 2 = 6$</td>
<td>6</td>
</tr>
<tr>
<td>[3] $4!$</td>
<td>$4 \times 3!$</td>
<td>$4 \times 6 = 24$</td>
<td>24</td>
</tr>
<tr>
<td>[4] $5!$</td>
<td>$5 \times 4!$</td>
<td>$5 \times 24 = 120$</td>
<td>120</td>
</tr>
</tbody>
</table>

Conceptual evaluation of recursive function.
Functions and Scope

Parameters

- A function's parameters are local to the function – they have no meaning outside the function itself.

- Parameter names may have the same identifier as a variable declared outside the function – the parameter names will take precedence inside the function.

\[
\text{int } n; \\
\text{long int factorial(int n)\{...\}}
\]

These are not the same \( n \).

Functions and Scope

Variables Declared Within a Function

- Variables declared within a function block are local to the function.

Example

\[
\text{int } x, y, z; \\
\text{int foo(int n)\{ \\
\quad \text{int a;} \\
\quad \ldots \\
\quad a += n; \\
\}\}}
\]

The \( n \) refers to the function parameter \( n \).

The \( a \) refers to the \( a \) declared locally within the function body.
**Functions and Scope**

**Variables Declared Within a Function**

- Variables declared within a function block are not accessible outside the function

```c
int x;
int foo(int n)
{
    int a; // always initialize
    return (a += n);
}
int main(void)
{
    x = foo(5); // This will generate an error. a may not be accessed outside of the function where it was declared.
    x = a;
}
```

**Example**

**Global versus Local Variables**

```c
int x = 5;
int foo(int y)
{
    int z = 1;
    return (x + y + z);
}
int main(void)
{
    int a = 2;
    x = foo(a); // x can be seen by everybody
    a = foo(x); // 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
}
```
Functions and Scope

Parameters

• "Overloading" variable names:

<table>
<thead>
<tr>
<th>n Declared Locally and Globally</th>
<th>n Declared Globally Only</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int n;</code></td>
<td><code>int n;</code></td>
</tr>
<tr>
<td><code>int foo(int n)</code></td>
<td><code>int foo(int x)</code></td>
</tr>
<tr>
<td>`{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>y += n;</code></td>
<td><code>y += n;</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

A locally defined identifier takes precedence over a globally defined identifier.

Example

```
int n;  // GLOBAL
int foo(int n)
{
    y += n;
}
int bar(int n)
{
    z *= n;
}
```

Functions and Scope

Parameters

• Different functions may use the same parameter names

• The function will only use its own parameter by that name
Running this code will result in the following output in the Uart1 IO window:

```
5
5
```

Why?
Remember: `#define` is used by the preprocessor to do text substitution before the code is compiled.

Functions and Scope

#define Within a Function

Example

```c
#define x 2

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

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

Functions

Historical Note

- C originally defined functions like this:

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

- Do not use the old method – use the new one only:

```c
int maximum(int x, int y)
{
   return ((x >= y) ? x : y);
}
```
Lab Exercise 8
Functions

Exercise 08
Functions

• Open the lab Project:

On the class website
Examples -> Lab8.zip

1 Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
Open the Project listed above.

⚠️ If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”
Exercise 08
Functions

Solution: Step 1

/*############################################################################
# STEP 1: Write two function prototypes based on the following information:
#         + Function Name: multiply_function()
#           - Parameters: int x, int y
#           - Return type: int
#         + Function Name: divide_function()
#           - Parameters: float x, float y
#           - Return type: float
############################################################################*/

int multiply_function( int x, int y);
float divide_function( float x, float y );

Solution: Step 2

#ifndef XOR
# STEP 2: Call the multiply_function() and divide_function().
# (a) Pass the variables intVariable1 and intVariable2 to the
# multiply_function().
# (b) Store the result of multiply_function() in the variable "product".
# (c) Pass the variables floatVariable1 and floatVariable2 to the
# divide_function().
# (d) Store the result of divide_function() in the variable "quotient".
#endif

//Call multiply_function
product = multiply_function( intVariable1 , intVariable2 );

//Call divide_function
quotient = divide_function( floatVariable1 , floatVariable2 );

// intQuotient will be 0 since it is an integer
intQuotient = divide_function( floatVariable1 , floatVariable2 );
Exercise 08
Functions

Solution: Steps 3 and 4

/**
 * STEP 3: Write the function multiply_function(). Use the function prototype
 * you wrote in STEP 1 as the function header. In the body, all you
 * need to do is return the product of the two input parameters (x * y)
 */

//Function Header
int multiply_function( int x, int y)
{
    return (x * y);  //Function Body
}

/**
 * STEP 4: Write the function divide_function(). Use the function prototype
 * you wrote in STEP 1 as the function header. In the body, all you
 * need to do is return the quotient of the two input parameters (x / y)
 */

//Function Header
float divide_function( float x, float y)
{
    return (x / y);  //Function Body
}

Exercise 08
Conclusions

• Functions provide a way to modularize code
• Functions make code easier to maintain
• Functions promote code reuse
Questions?
CMPE-013/L

Modules and Scope

Gabriel Hugh Elkaim
Spring 2013

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 automatic*

- Typically created on the stack

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

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 main(void)
{
  ...
}
```

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 (MPLAB-C30)

```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:

extern type identifier;

External Variable Declaration Example:

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;
}
```

```c
int a;
```
Storage Class Specifiers

External Variables

• 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*

```
Main.c

extern int x;

int main(void)
{
    x = 5;
    ...
}

SomeFileInProject.c

int x;

int foo(void)
{
    ...
}
```

Storage Class Specifiers

Register Variables

• `register` variables are placed in a processor's "hardware registers" for higher speed access than with external RAM (mostly used for microprocessors)

• Doesn't *usually* make sense in embedded microcontroller system where RAM is integrated into processor package

• 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 (or .h)
**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
#include <LibFile.h>

int x;

int main(void)
{
    int myVar;

    x = foo();
    myVar = x;
}

int foo(void);

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

static int foo(void)
{
    ...
}
```

**Library Files and Header Files**

- `#include <LibFile.h>`: Include declarations
- `extern int myVar;`: Declare
- `int foo(void);`: Define
- `int x;`: Use
Exercise 09
Multi-file Projects

• Open the lab Project:

On the class website
/Examples/Lab09.zip -> Load “Lab09.X”

1 Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
Open the Project listed above.

⚠ If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”
Exercise 09
Multi-File Projects

Solution: Step 1a and 1b (File1_09.h)

```c
/*############################################################################
# STEP 1a: Add variable declarations to make the variables defined in 
#          File1_09.c available to any C source file that includes this 
#          header file. (intVariable1, intVariable2, product) 
# ############################################################################*/
extern int intVariable1;
//Reference to externally defined "intVariable1"
extern int intVariable2;
//Reference to externally defined "intVariable2"
extern int product;

/*############################################################################
# STEP 1b: Add a function prototype to make multiply_function() defined in 
#          File1_09.c available to any C source file that includes this header 
#          file. 
# ############################################################################*/
int multiply_function(int x, int y);
```

Solution: Step 2a and 2b (File2_09.h)

```c
/*############################################################################
# STEP 2a: Add variable declarations to make the variables defined in 
#          File2_09.c available to any C source file that includes this header 
#          file. (floatVariable1, floatVariable2, quotient, intQuotient) 
# ############################################################################*/
extern float floatVariable1;
//Reference to externally defined "floatVariable1"
extern float floatVariable2;
//Reference to externally defined "floatVariable2"
extern float quotient;
//Reference to externally defined "quotient"
extern int intQuotient;

/*############################################################################
# STEP 2b: Add a function prototype to make divide_function() defined in 
#          File2_09.c available to any C source file that includes this header 
#          file. 
# ############################################################################*/
float divide_function(float x, float y);
```
Exercise 09

Conclusions

• Multi-file projects take the concept of functions further, by providing an additional level of modularization
• Globally declared variables and all normal functions are externally available if extern declarations and function prototypes are available
• Static functions are not available externally

Questions?
Public API

- Add
- Subtract
- Divide
- Multiply

Global Variables

Private Implementation

```c
#include "FindAverage.h"

void GetAverage(int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    float average = (float)sum / size;
    printf("Average: %.2f\n", average);
}
```