Functions
Functions

- Simple Function Example
- Function Prototype and Declaration
- Math Library Functions
- Function Definition
- Header Files
- Random Number Generator
- Call by Value and Call by Reference
- Scope (global and local)
- Call by Value Example
- Static Variables
char isalive (int i)
{
    if (i > 0)
        return 'A';
    else
        return 'D';
}

int main()
{
    int Peter, Paul, Mary, Tom;

    Peter = 1; Paul = 2; Mary = -1; Tom = 0;

    printf("Peter is %c Paul is %c
Mary is %c Tom is %c
", 
        isalive(Peter), isalive(Paul), 
        isalive(Mary), isalive(Tom));
    return 0;
}

main
- C programs start execution at main.
- is simply another function.
All functions have a return value.

%./dora
Peter is A Paul is A
Mary is D Tom is D
char isalive ( int i);

int main ()
{
    int Peter, Paul, Mary, Tom;

    Peter = 1; Paul = 2; Mary = -1; Tom = 0;

    printf("Peter is %c Paul is %c\nMary is %c Tom is %c\n", 
        isalive(Peter), isalive(Paul), 
        isalive(Mary), isalive(Tom));
    return 0;
}

char isalive ( int i)
{
    if (i > 0)
        return 'A';
    else
        return 'D';
}
5.2 Program Modules in C

- Functions \{also referred to as routines or subroutines\}
  - Modules in C
  - Programs combine user-defined functions with library functions.
    - C standard library has a wide variety of functions.

- Function calls
  - Invoking functions
    - Provide function name and arguments (data).
    - Function performs operations or manipulations.
    - Function returns results.
5.3 Math Library Functions

- Math library functions
  - perform common mathematical calculations.
  - #include <math.h>

- Format for calling functions
  - FunctionName ( argument );
    - If multiple arguments, use comma-separated list.
  - printf( "%2f", sqrt( 900.0 ) );
    - Calls function sqrt, which returns the square root of its argument.
    - All math functions return data type double.
  - Arguments may be constants, variables, or expressions.
### Fig. 5.2 Commonly used math library functions. (Part 1)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sqrt{x}$</td>
<td>square root of $x$</td>
<td>$\sqrt{900.0}$ is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\sqrt{9.0}$ is 3.0</td>
</tr>
<tr>
<td>$\exp(x)$</td>
<td>exponential function $e^x$</td>
<td>$\exp(1.0)$ is 2.718282</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\exp(2.0)$ is 7.389056</td>
</tr>
<tr>
<td>$\log(x)$</td>
<td>natural logarithm of $x$ (base $e$)</td>
<td>$\log(2.718282)$ is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\log(7.389056)$ is 2.0</td>
</tr>
<tr>
<td>$\log10(x)$</td>
<td>logarithm of $x$ (base 10)</td>
<td>$\log10(1.0)$ is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\log10(10.0)$ is 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\log10(100.0)$ is 2.0</td>
</tr>
<tr>
<td>$\text{fabs}(x)$</td>
<td>absolute value of $x$</td>
<td>$\text{fabs}(5.0)$ is 5.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{fabs}(0.0)$ is 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{fabs}(-5.0)$ is 5.0</td>
</tr>
<tr>
<td>$\text{ceil}(x)$</td>
<td>rounds $x$ to the smallest integer</td>
<td>$\text{ceil}(9.2)$ is 10.0</td>
</tr>
<tr>
<td></td>
<td>not less than $x$</td>
<td>$\text{ceil}(-9.8)$ is -9.0</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>rounds <code>x</code> to the largest integer not greater than <code>x</code></td>
<td><code>floor(9.2)</code> is 9.0, <code>floor(-9.8)</code> is -10.0</td>
</tr>
<tr>
<td><code>pow(x, y)</code></td>
<td><code>x</code> raised to power <code>y</code> ($x^y$)</td>
<td><code>pow(2, 7)</code> is 128.0, <code>pow(9, .5)</code> is 3.0</td>
</tr>
<tr>
<td><code>fmod(x, y)</code></td>
<td>remainder of <code>x/y</code> as a floating-point number</td>
<td><code>fmod(13.657, 2.333)</code> is 1.992</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of <code>x</code> ($x$ in radians)</td>
<td><code>sin(0.0)</code> is 0.0</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of <code>x</code> ($x$ in radians)</td>
<td><code>cos(0.0)</code> is 1.0</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of <code>x</code> ($x$ in radians)</td>
<td><code>tan(0.0)</code> is 0.0</td>
</tr>
</tbody>
</table>
5.4 Functions

- Functions
  - Modularize a program.
  - All variables defined inside functions are local variables.
    - Known and accessed only in defined function.
  - Parameter list
    - Communicate information between functions.
    - Local variables of the function.

- Benefits of functions
  - Software reusability
    - Use existing functions as building blocks for new programs.
    - Abstraction - hide internal details (library functions).
  - Avoid code repetition
5.5 Function Definitions

Function definition format

```
return-value-type  function-name( parameter-list )
{
    declarations and statements
}
```

- Function-name: any valid identifier.
- Return-value-type: data type of the result (default int)
  - void - indicates that the function returns nothing.
- Parameter-list: comma separated list, declares parameters.
  - A type must be listed explicitly for each parameter unless, the parameter is of type int.
Function definition format (continued)

```
return-value-type function-name( parameter-list )
{
    declarations and statements
}
```

- Definitions and statements: function body (block)
  - Variables can be defined inside blocks (can be nested).
  - Functions cannot be defined inside other functions!
- Returning control
  - If nothing returned
    - return;
    - or, until reaches right brace
  - If something returned
    - return expression;

© 2007 Pearson Ed -All rights reserved.
5.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters – what the function takes in.
  - Return type – data type function returns.
    (default \texttt{int})
  - Used to validate functions.
  - Prototype only needed if function definition comes after use in program.

- Promotion rules and conversions
  Promotion ::= temporary conversion to the highest type in the expression.
  - Converting to lower types can lead to errors.
Fig. 5.5 Promotion Hierarchy

<table>
<thead>
<tr>
<th>Data type</th>
<th>printf conversion specification</th>
<th>scanf conversion specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long double</td>
<td>%Lf</td>
<td>%Lf</td>
</tr>
<tr>
<td>double</td>
<td>%f</td>
<td>%lf</td>
</tr>
<tr>
<td>float</td>
<td>%f</td>
<td>%f</td>
</tr>
<tr>
<td>Unsigned long int</td>
<td>%lu</td>
<td>%lu</td>
</tr>
<tr>
<td>long int</td>
<td>%ld</td>
<td>%ld</td>
</tr>
<tr>
<td>unsigned int</td>
<td>%u</td>
<td>%u</td>
</tr>
<tr>
<td>int</td>
<td>%d</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned short</td>
<td>%hu</td>
<td>%hu</td>
</tr>
<tr>
<td>short</td>
<td>%hd</td>
<td>%hd</td>
</tr>
<tr>
<td>char</td>
<td>%c</td>
<td>%c</td>
</tr>
</tbody>
</table>
5.7 Function Call Stack and Activation Records

Function call stack (or program execution stack)

- A stack is a last-in, first-out (LIFO) data structure.
  - Anything put into the stack is placed “on top”.
  - The only data that can be taken out is the data on top.
- C uses a program execution stack to keep track of which functions have been called.
  - When a function is called, it is placed on top of the stack (pushed onto the stack).
  - When a function ends, it is taken off the stack (popped off the stack) and control returns to the function immediately below it.
- Calling more functions than C can handle at once is known as a “stack overflow error”.

© 2007 Pearson Ed - All rights reserved.
5.8 Headers

- Header files
  - Contain function prototypes for library functions.
  - E.g., `<stdlib.h>`, `<math.h>`
  - Load with `#include <filename>`
    ```
    #include <math.h>
    ```

- Custom header files
  - Create file with functions.
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - This facilitates functions reuse.
<table>
<thead>
<tr>
<th>Standard library header</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;stdio.h&gt;</code></td>
<td>Contains function prototypes for the standard input/output library functions, and information used by them.</td>
</tr>
<tr>
<td><code>&lt;stdlib.h&gt;</code></td>
<td>Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.</td>
</tr>
<tr>
<td><code>&lt;string.h&gt;</code></td>
<td>Contains function prototypes for string-processing functions.</td>
</tr>
<tr>
<td><code>&lt;time.h&gt;</code></td>
<td>Contains function prototypes and types for manipulating the time and date.</td>
</tr>
</tbody>
</table>
5.10 Random Number Generation

- rand function
  - Load <stdlib.h>
  - Returns a "random" number between 0 and RAND_MAX (at least 32767).
    \[ i = \text{rand}(); \]
  - Pseudo random
    - Preset sequence of "random" numbers
    - Same sequence for every function call

- Scaling
  - To get a random number between 1 and n.
    \[ i = 1 + (\text{rand}() \mod n) \]
  - rand() \mod n returns a number between 0 and n – 1.
  - Add 1 to make random number between 1 and n.
    \[ i = 1 + (\text{rand}() \mod 6) \]
  - number between 1 and 6

© 2007 Pearson Ed - All rights reserved.
/* Fig. 5.7: fig05_07.c  
Shifted, scaled integers produced by 1 + rand() % 6 */

#include <stdio.h>
#include <stdlib.h>

/* function main begins program execution */
int main( void )
{
    int i; /* counter */

    /* loop 20 times */
    for ( i = 1; i <= 20; i++ ) {

        /* pick random number from 1 to 6 and output it */
        printf( "%10d", 1 + ( rand() % 6 ) );

        /* if counter is divisible by 5, begin new line of output */
        if ( i % 5 == 0 ) {
            printf( "\n" );
        } /* end if */
    } /* end for */

    return 0; /* indicates successful termination */
} /* end main */
Call by Value

- When arguments are passed by the calling routine to the called routine by value,
  - A copy of the argument’s value is passed to the called routine.
  - Hence, any changes made to the passed argument by the called routine DO NOT change the original argument in the calling routine.
  - This avoids accidental changes known as side-effecting.
Call by Reference

- When arguments are passed by the calling routine to the called routine by reference,
  - The original argument is passed to the called routine.
  - Hence, any changes made to the passed argument means that this changes remain in effect when control is returned to the calling routine.
In C, all arguments (by default) are passed by value.

* Call by reference is “simulated” in C by using the address operator (\&) and the indirection operator (*).

Array arguments are automatically passed by reference!

{Much more about all this when we introduce pointers.}
In C, the scope of a declared variable or type is defined within the range of the block of code in which the declaration is made.

Two simple examples:

1. declarations outside all functions are called **globals**. They can be referenced and modified by **ANY** function.

{Note - this violates good programming practice rules}.
2. **Local variables** – declarations made inside a function mean that variable name is defined only within the scope of that function.

- Variables with the same name outside the function are **different**.
- Every time the function is invoked the value of local variables need to be re-initialized upon entry to the function.
- Local variables have the automatic storage duration by default (implicit).

```plaintext
auto double x, y    /* explicit */
```
/* Example shows call-by-value and the scope of a global variable 'out' */

int out = 100;     /* out is global variable */

/* byval modifies local, global and variables passed by value. */

int byval ( int i, int j)
{
    int tmp;
    tmp = 51;
    i = tmp - 10*i - j;
    out = 2*out + i + j;
    j++;
    tmp++;
    printf("In byval: i = %2d, j = %2d, tmp = %2d, out = %3d\n",
           i, j, tmp, out);
    return i;
}
int main ()
{
    int i, j, tmp, s;

    tmp = 77;
    j = 1;

    for (i = 0; i < 2; i++)
    {
        s = byval(i,j);
        out = out + s - j;
        printf("In main : i = %2d, j = %2d, tmp = %2d, out = %3d, s = %d\n", i, j, tmp, out, s);
    }
    return 0;
}
int main ()
{
    int i, j, tmp, s;
    tmp = 77;
    j = 1;
    for (i = 0; i < 2; i++)
    {
        s = byval(i, j);
        out = out + s - j;
        printf("In main : i = %2d, j = %2d, tmp = %2d, out = %3d, s = %d \n", i, j, tmp, out, s);
    }
    return 0;
}
Static Variables

- Local variables declared with the keyword `static` are still only known in the function in which they are defined.
- However, unlike automatic variables, static local variables retain their value when the function is exited.

```
e.g.,

    static int count = 2;
```

- All numeric static variables are initialized to zero if not explicitly initialized.
/* An Example of a Static Variable */

float nonstat ( float x)
{
    int i = 1;
    i = 10*i;
    x = i - 5.0*x;
    return x;
}

float stat (float y)
{
    static int i = 1;
    i = 10*i;
    y = i - 5.0*y;
    return y;
}
```c
int main()
{
    int i;
    float var1, var2;
    var2 = var1 = 2.0;
    printf(" var1 = %9.2f, var2 = %9.2f\n", var1, var2);

    for ( i = 1; i <= 3; i++)
    {
        var1 = nonstat(var1);
        var2 = stat(var2);
        printf(" var1 = %9.2f, var2 = %9.2f\n", var1, var2);
    }
    return 0;
}
```

```
$../static
var1 = 2.00, var2 = 2.00
var1 = 0.00, var2 = 0.00
var1 = 10.00, var2 = 100.00
var1 = -40.00, var2 = 500.00
```
The important concepts introduced in this Powerpoint session are:

- Functions
- Libraries
- Header Files
- Call by Value
- Call by Reference
- Scope (global and local)
- Static Variables