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

```c
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`. 
5.5 Function Definitions

Function definition format (continued)

```c
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;`
5.6 Function Prototypes

- Function prototype
  - Function name
  - Parameters - what the function takes in.
  - Return type - data type function returns.
    (default `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>
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”**.
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.
    
    \[
    i = 1 + (\text{rand}() \% n)
    \]
    - `rand() \% n` returns a number between 0 and \( n - 1 \).
    - Add 1 to make random number between 1 and n.
      
      \[
      i = 1 + (\text{rand}() \% 6)
      \]
      - 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 */

Generates a random number between 1 and 6
Use of modulo operator is a standard trick to print output in column format.
Call by Value

When arguments are passed by the calling routine to the called routine by value,

- A copy of each 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.
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).

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

int out = 100; /* out is a 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.

```c
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;
}
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;
}
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