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

Simple Function Example

```
char isalive ( int i)
{
  if (i > 0)
    return 'A';
  else
    return 'D';
}
int main ()
{
  int Peter, Paul, Mary, Tom;

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

  printf("Peter is %c Paul is %c\nMary 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 D Paul is D
Mary is A Tom is A
```

Function Declarations

```c
char isalive ( int i);

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

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

  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';
}
```

function prototype

function placed after reference

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)

Function	Description	Example
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282` `exp( 2.0 )` is `7.389056`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 1.0 )` is `0.0`
		`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`fabs( x )`	absolute value of *x*	`fabs( 5.0 )` is `5.0`
		`fabs( 0.0 )` is `0.0`
		`fabs( -5.0 )` is `5.0`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`

Fig. 5.2 Commonly used math library functions. (Part 2)

Function	Description	Example
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128.0` `pow( 9, .5 )` is `3.0`
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

5.4 Functions

- Functions
  - Modularize a program.
  - All variables defined inside functions are local variables.
    - Known only in function defined.
  - Parameters
    - Communicate information between functions.
    - Local variables
- 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.

5.5 Function Definitions

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 can not 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
  – Converting to lower types can lead to errors.

Fig. 5.5 Promotion hierarchy

Data type	printf conversion specification	scanf conversion specification
`Long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
`Unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

5.7 Function Call Stack and Activation Records

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.
  - When a function ends, it is taken 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.

Fig. 5.6 Standard library headers (Part 3)

Standard library header	Explanation
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

5.10 Random Number Generation

- `rand` function
  - Load <stdlib.h>
  - Returns "random" number between `0` and `RAND_MAX` (at least `32767`).

    ```
    i = rand();
    ```
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call
- Scaling
  - To get a random number between `1` and `n`.

    ```
    i = 1 + ( rand() % n )
    ```
    - `rand() % n` returns a number between `0` and `n - 1`.
    - Add `1` to make random number between `1` and `n`.

      ```
      i = 1 + ( rand() % 6)
      ```
      - number between `1` and `6`

Random Number Example

```c
/* 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

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Call by Value

- When arguments are passed by the calling routine to the called routine by value,
  - A copy of the argument is passed to the called routing.
  - 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 routing.
  - Hence, any changes made to the passed argument means that this changes remain in effect when control is returned to the calling routine.

Scope (simple)

- 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}.

Scope (simple)

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 re-initialized upon entry to the function.
- Local variables have the automatic storage duration by default (implicit).

```
auto double x, y    /* explicit */
```

Call by Value Example

```
/* 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;
}
```

global is changed

Call by Value Example (cont)

```
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;
}
```

global is changed

Call by Value Example

```
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;
}
```

```
$./byval
In byval: i = 50, j =  2, tmp = 52, out = 251
In main : i =   0, j =  1, tmp = 77, out = 300, s = 50
In byval: i = 40, j =  2, tmp = 52, out = 641
In main : i =   1,  j =  1, tmp = 77, out = 680, s = 40
```

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.

Static Variables

```
/* 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;
}
```

Static Variables

```
int main()
{
  int i;
  float var1, var2;
  var2 = var1 = 2.0;
  printf(" var1 = %9.2f, var2 = %9.

  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
```

Summary

- 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