**Engr 101** **Sept. 13, 2019**
**Day 9**

# Top-Down Design and Structure Charts

- top-down design
  - a problem solving method
  - first, break a problem up into its major subproblems
  - solve the subproblems to derive the solution to the original problem

Example
Write a program which will use the * character to draw a circle and a triangle.

We could do this with just a single main program. It is easier to think about if we break the problem down into two separate and simpler functions: DrawCircle and DrawTriangle. Each of these functions can be written without arguments and without a return value.

DrawCircle might look like this:

```
void DrawCircle()
{
    printf("   * *  \n");
    printf(" *     * \n");
    printf(" *     * \n");
    printf("   * *  \n");
}
```

The DrawTriangle function can be broken down into two simpler pieces: DrawIntersectingLines and DrawBase. The complete program is shown below.

Notes:
1. Each function has two parts: prototype and function definition. The prototype appears before the main program and is typically identical to the first line of the function definition except that it must end in semicolon. The function definition has the body of the function.
2. The prototypes are processed first. This enables the compiler to make use of the functions in the main program. We could skip the prototypes altogether and write the function definitions first – before the main program where they are used. But this makes for awkward reading.

Trace through the program by hand.
Use the debugger to go through the program and watch its execution.

```c
#include<stdio.h>
void DrawCircle();           //function prototypes
void DrawTriangle();
void DrawIntersectingLines();
void DrawBase();

void main(void)
{
    DrawCircle();
    DrawTriangle();
}
void DrawCircle()
{
    printf("   * *  \n");
    printf(" *     * \n");
    printf(" *     * \n");
    printf("   * *  \n");
}
void DrawTriangle()
{
    DrawIntersectingLines();
    DrawBase();
}
void DrawIntersectingLines()
{
    printf("  / \\   \n");
    printf(" /   \\  \n");
    printf("/     \\ \n");
}
void DrawBase()
{
    printf("-------\n");
}
```

```
  * *
 *   *
 *   *
  * *
 /\
/  \
/   \
-------
```

Press any key to continue . . .

# Advantages of Using Function Subprograms

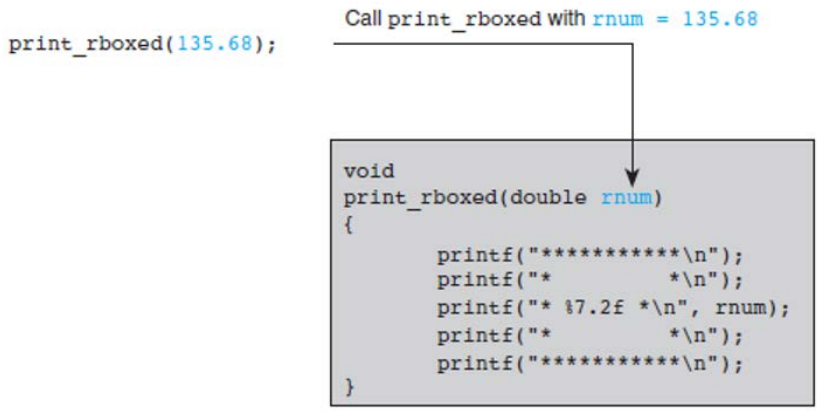
- procedural abstraction
  - a programming technique in which a main function consists of function calls and each function is implemented separately
- reuse of function subprograms
  - functions can be executed more than once in a program

## Functions with Input Arguments

- input argument
  - arguments used to pass information into a function subprogram
- output argument
  - arguments used to return results to the calling function

## void Functions with Input Arguments

- actual argument
  - an expression used inside the parentheses of a function call
- formal parameter
  - an identifier that represents a corresponding actual argument in a function definiiton




This function has a single input argument and returns nothing. Its return type is void.

When the function is called the actual argument is copied into the formal parameter. There are two instance of the argument in memory – one belongs to the calling program and the second belongs to the function. Changing the argument in the function does not change the value of the variable in the main program.

```
#include "stdio.h"
void Fun1(double x);  //Prototype – note semicolon at end.

int main()
  {double x;
   x = 22;
   //Function call
   Fun1(x);      //x here is an argument
   return 0;
  }
//Function body
//This function takes one double argument and returns nothing.
//double x is called a formal parameter.
void Fun1(double x)  //Function heading (no semicolon)
  {double y;           //Function body
   y = 3*x*x + 9.7;
   printf("if x = %f, y = %f \n", x, y);
  }
```

Note that we could call x by a different name in the function since it is indeed a different variable.

We can change this function to one that accepts a double argument and returns a double result:

```
#include "stdio.h"
double Fun1(double x);  //Prototype – note semicolon at end.

int main()
  {double x, y;
   x = 22;
   //Function call
   y = Fun1(x);    //x here is an argument
   printf("if x = %f, y = %f \n", x, y);

   return 0;
  }
//Function body
//This function takes one double argument and returns a double.
//double x is called a formal parameter.
double Fun1(double x)  //Function heading (no semicolon)
  {double y;           //Function body
   y = 3*x*x + 9.7;
   return y;
  }
```

Draw a memory map for this function

Talk about local variables.

## Functions with Multiple Arguments

### Argument List Correspondence

- The number of actual arguments used in a call to a function must be the same as the number of formal parameters listed in the function prototype.

- Each actual argument must be of a data type that can be assigned to the corresponding format parameter with no unexpected loss of information.

## Functions with Multiple Arguments

### Argument List Correspondence

- The order of arguments in the lists determines correspondence.
  - The first actual argument corresponds to the first formal parameter.
  - The second actual argument corresponds to the second form parameter.
  - And so on…

Example

## Function scale

```
1.  /*
2.   * Multiplies its first argument by the power of 10 specified
3.   * by its second argument.
4.   * Pre : x and n are defined and math.h is included.
5.   */
6.  double
7.  scale(double x, int n)
8.  {
9.       double scale_factor;     /* local variable */
10.      scale_factor = pow(10, n);
11.
12.      return (x * scale_factor);
13. }
```

```
1.  /*
2.   * Tests function scale.
3.   */
4.  #include <stdio.h>           /* printf, scanf definitions */
5.  #include <math.h>            /* pow definition */
6.
7.  /* Function prototype */
8.  double scale(double x, int n);
9.
10. int
11. main(void)
12. {
13.       double num_1;
14.       int num_2;
15.
16.       /* Get values for num_1 and num_2 */
17.       printf("Enter a real number> ");
18.       scanf("%lf", &num_1);
19.       printf("Enter an integer> ");
20.       scanf("%d", &num_2);
21.
22.       /* Call scale and display result. */
23.       printf("Result of call to function scale is %f\n",
24.              scale(num_1, num_2));          actual arguments
25.
26.       return (0);
27. }
28.                                            information flow
29.
30. double
31. scale(double x, int n)                     formal parameters
32. {
33.       double scale_factor;      /* local variable - 10 to power n */
34.
35.       scale_factor = pow(10, n);
36.
37.       return (x * scale_factor);
38. }
    Enter a real number> 2.5
    Enter an integer> -2
    Result of call to function scale is 0.025
```

Draw a memory map for this function

**Engr 101** **September 13, 2019**
**Polynomial method**

Write a console application that prompts the user for a value for x (a double). Evaluate and print the value of y where $y = x^4 - 3x^3 + 2x^2 + 1$. Put the function into a method called FindY which accepts an argument of type double and returns a double.

Turn in a printed copy of your source file.