C-6 Functions

Library Functions:

- They come with the system.
- We have already used some of them.

- Examples:
- x = **sqrt** (y);
- w = **pow**(x,3);
- Sine = **sin**(start);

Vocabulary:

Function Prototype A definition or outline of a function to be used.

Each function requires a Function Prototype

The Prototype can tell us:

- if a function will return a value
- what values will be sent in to the function
- what type those values have (int, double, etc)
- the name of the function.

Function Prototype

For functions that **we write**, the prototype is located:

- Above the line "int main(void)"
- Later we will learn to put them in a separate file.

For functions that come with the **system**:

- The prototype is located in the system include file.
- Ex: The prototype for *printf* is located in *stdio.h* and we are not required to re-enter it as long as our program starts with *#include <stdio.h>*

```c
/* An example */
/*-----------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
int reverse (int num);       /* function prototype */
/*-----------------------------------------------------------*/
int main (void)
{
    int num_in;

    printf ("\nEnter a 2-digit number:  ");
    scanf ("%i", &num_in);

    printf ("\n\nThe number %i reversed is %d \n\n",
             num_in, reverse(num_in));
    return EXIT_SUCCESS;
}
/*---End of main-------------------------------------------*/
```

```
/*-----------------------------------------------------*/
// This function will reverse a two digit number

int reverse (int num)
{
   int digit1, digit2;

   digit1 = num / 10;
   digit2 = num % 10;
   return (digit2 * 10) + digit1;
}
/*---End of reverse----------------------------------*/
```

***The Run*:**

The number 27 reversed is 72

```
#include <stdio.h>                    //Another example
#include <stdlib.h>
void function1 (void);           /* function prototypes */
void function2 (int n, double x);

/*----------------------------------------------------*/
int main (void)
{   int m;   double y;
    m = 15;
    y = 308.24;
    printf ("The value of m in main is m = %d \n\n", m);

    function1 ( );           /* has no argument */
    function2 (m, y);

    printf ("The value of m in main is still m = %d \n\n", m);
      return EXIT_SUCCESS;
}
/*----------------------------------------------------*/
```

Output after the 1st printf in main:

The value of m in main is m = 15

```
/*-----------------------------------------------------------*/
void function1 (void)
{
   printf("function 1 is a void function that does"
          " not receive values from main. \n\n");
   return;
}
/*-----------------------------------------------------------*/
```

Output after we finish with **function 1**:

The value of m in main is m = 15

function 1 is a void function that does not receive values from main.

```
/*-----------------------------------------------------------------------------*/
void function2 (int n, double x)
{
   int k, m;
   double z;
   k = 2 * n + 2;
   m = 5 * n + 37;
   z = 4.0 * x – 58.4;

   printf(“function2 is a void function that does receive \n”
          “values from main.  The values received from main are:  “
          “\t n = %d \n\t x = %lf \n\n”, n, x);

   printf(“function2 creates three new variables, k, m, and z \n”
          “These variables have the values : \n”
          “\t t = %d \n\t m = %d \n\t z = %lf \n\n”, k, m, z);

   return;
}
/*-----------------------------------------------------------------------------*/
```

Output – FINAL – after **function 2**:

The value of m in main is m = 15

function 1 is a void function that does not receive values from main.

function2 is a void function that does receive
values from main. The values received from main are:

n = 15

x = 308.240000

function2 creates three new variables, k, m, and z
These variables have the values:

k = 32

m = 112

z = 1174.560000

The value of m in main is still m = 15

General Form of Functions:

Functions that *don't* return a value:

```
void function_name (parameter declarations) {
   declarations;
   statements;
   return;
}
```

Function Example: **void function2 (int n, double x)**

The prototype for this sort of function:

```
void function_name (parameter declarations);
```

Prototype Example: **void function2 (int n, double x);**

General Form of Functions:

Functions that return *one* value & *only one value*.

```
return_type function_name (parameter declarations)
{
      declarations;
      statements;
      return expression;
}
```

Function Example: **int reverse (int num)**

The prototype for this sort of function:

return_type function_name (parameter declarations);

Prototype Example: **int reverse (int num);**

We will use a **<u>return</u>** in every function that we write;

It is good engineering practice.

Call-by-value

On passing a value to a function,
the actual value stored in memory is passed to the sub-function,
not its memory location.

In other words,
<u>main</u> keeps the original copy.

The <u>sub-function</u> gets a xerox copy.
If the sub-function alters its xerox copy,
it does not change the original copy/value in main.

```
/*----------------------------------------Functions2.c--------------*/
#include <stdio.h>
#include <stdlib.h>
int m = 12;          /* file scope variable (global) */

int function1 (int a, int b, int c, int d);   /*function prototype */
/*------------------------------------------------------------------------*/
int main (void)
{
   int n = 30;
   int e, f, g, h, i;
   e = 1;
   f = 2;
   g = 3;
   h = 4;
   printf("\n\nIn main (before call to function 1): \n"
      "  m = %d\n  n = %d\n  e = %d\n\n", m, n, e);
```

Let’s stop and see the output of that printf:

In main (before the call to function1):

m = 12

n = 30

e = 1

and now on with the code, repeating that printf to put us in context…

```
   printf(“\n\nIn main (before call to function 1): \n”
          “  m = %d\n  n = %d\n  e = %d\n\n”, m, n, e);

   i = function1(e, f, g, h);

   printf(“After returning to main:  \n”);
   printf(“  n = %d \n  m = %d \n  e = %d \n  I = %d\n\n”,
          n, m, e, i);

return EXIT_SUCCESS;
}
/*-----end of main -------------------------------------------------*/
```

```
/*-------------------------------------------------------------*/
int function1 (int a, int b, int c, int d)
{
   int n = 400;
   printf(“In function1: \n  n = %d \n  m = %d  intially \n”
          “  a = %d initially \n\n”, n, m, a);
   m = 999;

   if (a >= 1)   {
      a += b + m + n;
      printf(“   m = %d after being modified \n”
             “  a = %d after being modified \n\n”, m, a);
      return a;
   }
   else   {
      c += d + m + n;
      return c;
   }
}
/*------end of function 1-----------------------------------*/
```

Now to see the WHOLE output:

In main (before the call to function1):
m = 12
n = 30
e = 1

In function1:
n = 400
m = 12 initially
a = 1   initially

m = 999 after being modified
a = 1402 after being modified

After returning to main:
n = 30
m = 999
e = 1
i = 1402

Scope of Variables

Definition of SCOPE

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed.

There are three types of Scope.

Scope of Variables:

(1) local variables –

- located inside a function or a block
- an identifier declared within a block unit is only active within the block.
- blocks are surrounded by { }
- an identifier declared within a function is only active within the function.

Scope of Variables:

(2) global variables –

- located outside of all functions
- usually on top of the program.
- they hold their values throughout the lifetime of your program
- they can be accessed inside any of the functions defined for the program.

Scope of Variables:

(3) **prototype scope**

- an identifier used in a function prototype
- used only within the one line of the prototype
- is not declared outside that line
- are treated as local variables with-in a function
- they take precedence over global variables.

Where do you Declare Variables?

Outside any *function definition*

.e.g., Prior to the start of the **main()** function

Global/external variables

Within a *function*, after the opening **{**

Local to the function

Within a *block of code*, after the **{**

Local to the area surrounded by the **{}** braces

Random Numbers

RANDOM NUMBERS

In stdlib.h there is a function to generate random numbers:

int rand (void);

rand generates a random integer
between 0 and RAND_MAX

RAND_MAX is a system-defined integer in stdlib.h
in our system, it is:
/* The largest number rand will return (same as INT_MAX). */
#define RAND_MAX 2,147,483,647

Random numbers are generated using a seed value.

By default, the seed = 1

Examples:

printf (“%i %i %i\n”, rand( ), rand( ), rand( ));

41 18467 6334

```
for (k=1; k <= 10; k++)
{
   printf(“%i”, rand( ) );
}
```

41 18467 6334 26500 19169
15724 11478 29358 26962 24464

These two examples show that **rand ( )** is *pseudo random.*

Random numbers are generated using a seed value.

By default, the seed = 1

The same random number sequence will be generated given a certain seed.

This is called **pseudo-randomness**.

We can change the value of seed from the default by using another function ***srand*** .

Its prototype in stdlib.h is:

```
void srand (unsigned int);
```

A different seed will generate a different sequence of random numbers.

***Examples*:**

```
unsigned seed;
printf("Enter a seed: ');
scanf("%u", &seed);        /* %u for unsigned int*/
srand(seed);
printf("%i", rand( ) );
```

seed of 1 → 41
seed of 123 → 440
seed of 5 → 54

So using the different seeds produces a list of numbers that look random.

```c
/*-----------------------------------------------------------------*/
/*  This program generates and prints ten random      */
/*  integers between user-specified limits.                  */

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

int rand_int(int a, int b);      // function prototype
/*-----------------------------------------------------------------*/
int main(void)
{
  /*  Declare variables and function prototypes.  */
  unsigned int seed;
  int a;
  int b;
  int k;
  char go_on[3] = "y";
```

```
while (go_on[0] == 'y' || go_on[0] == 'Y')
{
        /*  Get seed value and interval limits.  */
        printf("\nEnter a positive integer seed value:  ");
        scanf("%u",&seed);
        srand(seed);
        printf("\nEnter integer limits a and b (a<b):  ");
        scanf("%i %i",&a,&b);

        /*  Generate and print ten random numbers  */
        printf("\nRandom Numbers: \n\n");
        for (k=1; k<=10; k++)
                printf("%i ",rand_int(a,b));
        printf("\n\n");
        printf("Enter \"y\" or \"Y\" for YES if you wish to continue:  ");
        scanf("%s", go_on);
  }
```

```
    return EXIT_SUCCESS:
}
/*------End of main------------------------------------------*/


/*  This function generates one random integer       */
/*  between specified limits a and b (a<b).          */

int rand_int(int a, int b)
{
  return (rand( ) % (b - a + 1) + a);
}

/*------End of rand_int-------------------------------------*/
```

Functions that “return” more than one value

First a look at a variable.

variable

name → counter

$\boxed{\quad 5 \quad}$ ← contents

address → 664136

New use of operators:

We will use the <u>address operator</u> (**&)** to *pass* the address of the variable to a sub-function.

Inside the sub-function, we will use the

<u>indirection operator</u> (*****) (the asterisk)

to refer to the contents of the variable rather than its address.

```
#include <stdio.h>                    //Functions3.c
#include <stdlib.h>
#include <math.h>

void find_area(double s, double *b, double *h, double *a);
          /* function prototype */
/*-----------------------------------------------------------------*/
int main(void)
{
   double side = 5.1875;  /* triangle parts*/
   double base = 3;
   double height = 5;
   double area = 0;

   printf ("\nIn main before function, the values are:  \n"
          "    Side   = %f \n"
          "    Base   = %f \n"
          "    Height = %f \n"
          "    Area   = %f \n", side, base, height, area);
```

Results of the Run:

In main before function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 0.000000

```
        /* after the printf statement */

    find_area(side, &base, &height, &area);

    printf ("\nIn main after function, the values are: \n"
            "    Side   = %f \n"
          "    Base   = %f \n"
          "    Height = %f \n"
          "    Area   = %f \n\n", side, base, height, area);

     return EXIT_SUCCESS;
}
/*----end of main----------------------------------------*/
```

```c
/*---------------------------------------------------------------------------*/
/*  This function will calculate the area of a triangle      */

void find_area(double s, double *b, double *h, double *a)
{
   *a = ( (*b) * (*h)) / 2.0;
   s = 400;    /* Just to see what will happen */

   printf ("\n   In find_area after computation, the values are: \n"
           "       Side   = %f \n"
           "       Base   = %f \n"
           "       Height = %f \n"
           "       Area   = %f \n", s, *b, *h, *a);
  return;
}
/*------end of find_area--------------------------------------*/
```

Results of the Run:

In main before function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 0.000000

In find_area after computation, the values are:

Side = 400.000000

Base = 3.000000

Height = 5.000000

Area = 7.500000

In main after function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 7.500000

Now to really understand why the “&” and “*” pair work (the address operator and the indirection operator).

We **re-run** the above program but this time we will **print** out the actual addresses in memory of the variables we used.

%p prints an address in a notation consistent with the addressing scheme of our computers. (for our computers, this is HEX)

%u prints an unsigned integer (in base ten)

```c
/*------------------------------------------------------------*/
#include <stdio.h>                    //functions4.c
#include <stdlib.h>
#include <math.h>

void find_area(double s, double *b, double *h, double *a);
         /* function prototype */
/*------------------------------------------------------------*/
int main(void)
{
   double side    = 5.1875;  /* triangle parts*/
   double base    = 3;
   double height = 5;
   double area    = 0;
```

```
printf ("\n In main before function, the values are:  \n"
        "   Side    = %f \n"
        "   Base    = %f \n"
        "   Height = %f \n"
        "   Area     = %f \n", side, base, height, area);

/* NEW Printf  follows*/

printf ("\n In main, the addresses are: \n"
        "   Side    = %p   %u\n"
        "   Base    = %p   %u\n"
        "   Height = %p   %u\n"
        "   Area     = %p   %u\n",
       &side,&side,
       &base,&base,
       &height,&height,
       &area,&area);
```

In main before function, the values are:

Side   = 5.187500

Base   = 3.000000

Height = 5.000000

Area   = 0.000000

In main, the addresses are:

Side   = 0012FF78   1245048

Base   = 0012FF70   1245040

Height = 0012FF68   1245032

Area   = 0012FF60   1245024

```
   /* rest of function main  */

   find_area(side, &base, &height, &area);

   printf ("\nIn main after function, the values are: \n"
           "    Side    = %f \n"
           "    Base   = %f \n"
           "    Height = %f \n"
           "    Area    = %f \n", side, base, height, area);

    return EXIT_SUCCESS;
}
/*-----end of main------------------------------------------*/
```

```
/*  This function will calculate the area of a triangle      */
void find_area(double s, double *b, double *h, double *a)
{
   *a = ( *b * *h) / 2.0;
    s = 400;    /* Just to see what will happen */
    printf ("\n   In find_area after computation, the values are: \n"
           "       Side    = %f \n"
           "       Base   = %f \n"
           "       Height = %f \n"
           "       Area    = %f \n", s, *b, *h, *a);
     printf ("\n   In find_area, the addresses are: \n"
            "       Side    = %p   %u\n"
            "       Base   = %p   %u\n"
            "       Height = %p   %u\n"
            "       Area    = %p   %u\n", &s,&s,  b,b,  h,h,  a,a);
  return;
}               /*----end of find_area---*/
```

In find_area after computation, the values are:

Side = 400.000000
Base = 3.000000
Height = 5.000000
Area = 7.500000

In find_area, the addresses are:

Side = 0012FF00 1244928
Base = 0012FF70 1245040
Height = 0012FF68 1245032
Area = 0012FF60 1245024

In main after function, the values are:

Side = 5.187500

Base = 3.000000

Height = 5.000000

Area = 7.500000

More on Scope of Variables

Storage Classes Specifiers

automatic – the default. We may use the word “auto”.

Ex. auto int x;

external – for global variables initially declared in other files

Ex. extern int count;

static - specifies that memory for this local variable should be kept or saved even after control has returned to the calling function

Ex. static int function_count;

register – specifies use of computer registers, rather than memory IF POSSIBLE.

Ex. register int speed;

typedef – to be covered in class later this semester.

<u>Example of Storage Class</u>**:**

```
#include …
int count=0;
…
int main (void) {
    int x, y, z;
    …
}
/*--------------------------*/
extern int count;
int calc(int a, int b) {
    int x;
     …
}
/*--------------------------*/
extern int count;
int check(int sum) {
     …
}
/*--------------------------*/
```

count is a global variable referenced by ***calc*** and ***check.***

x, y. & z are local variables referenced only by ***main.***

a, b, x are local variables only referenced by ***calc.***

sum is a local variable that can only be referenced by ***check.***

There are TWO local variables ***x*** but they are different variables with different scopes.

Static Variables

Scope Rules for “Static” variables

- Static Variables: use static prefix on functions and variable declarations to **limit scope**

  - static prefix on external variables will limit scope to the rest of the source file (not accessible in other files)

  - static prefix on functions will make them invisible to other files

  - static prefix on internal variables will create permanent private storage; retained even upon function exit

New operator: sizeof()

The **sizeof** operator may be used to determine the size of a data object or type.

```c
/* Demonstrate static variables (4_UNIX)        */

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

char name[100];                 /* variable accessible from all files */
static int i;                   /* variable accessible only from this
                                   file */
static int max_so_far(int);   /* function prototype accessible
                                   only from this file */

int main (void) {
   int val[] = {14, 25, 10, 100, 20};
   int i;
   for (i=0; i<sizeof(val)/sizeof(int); i++)
      printf("max = %d \n", max_so_far(val[i]));
   return (EXIT_SUCCESS);
}
/*-----------------------------------------*/
```

```
/*-----------------------------------------------*/
/* code for the function                   */

int max_so_far(int curr) {
   static int biggest=0;      /*Variable whose value is retained
                                 between each function call */
   if(curr > biggest)
      biggest=curr;
   return biggest;
}
/*-----------------------------------------------*/
```

The run with the static variable:

```
[bielr@athena ClassExamples]> a.out
max = 14
max = 25
max = 25
max = 100
max = 100
[bielr@athena ClassExamples]>
```

C-6 Functions

The End