ECE 131 – Programming Fundamentals

Dr. Daryl Lee

University of New Mexico

UNM | Electrical & Computer Engineering

Parameters

- What is the purpose of functions in programming? They allow you to package some useful functionality in a way that can be reused.
- The motivation for including parameter lists in functions is that they allow generic procedures to be developed. I.e., the data that a function will work on does not have to be specified beforehand, but instead can be varied from one function call to the next.
  Ex: `bubblesort()`
- Thus, a function is written so that it operates on the "dummy variables" supplied in the parameter list. These dummy variables are referred to as formal parameters.
- The data supplied when the procedure is actually called are referred to as actual parameters.

Parameters

Consider:

```
void func(int);

main() {
   int x=1;
   func(x);
}

void func(int a) {
   int y;
   y = a * 2;
}
```

The variable `x` is the actual parameter, and it is supplied to `func()`, where it is assigned to the formal parameter a.

THE UNIVERSITY of NEW MEXICO

Parameter Passing

There are two mechanisms that can be used to pass the actual parameters to a procedure:

- Pass-by-value – a copy of the actual parameters (i.e., their values) is supplied to the formal parameters. Thus, any modification of these parameters is not reflected outside of the function. That is, a change in value of one of the parameters inside the function is local to that function.
- Pass-by-reference – the formal parameters are bound to the reference values of the actual parameters. This means that a change in the value of a parameter inside a function will actually change the value contained in the variable to which that parameter references. Thus, a change in the value of a parameter inside the function is *not* local to that function.

Important: C only supports pass-by-value.

Parameter Passing

What does the following program print:

```
main() {
   int x=1;
   increment(x);
   printf("1+1 = %d", x);
}

void increment(int a) {
   a = a+1;
}
```

It prints `1+1 = 1`. When the value of `x` is passed to `increment()`, it's assigned to the formal parameter a in increment, and incremented locally, without changing `x`.

THE UNIVERSITY of NEW MEXICO

Parameter Passing

Here's how to fix it:

```
main() {
   int x=1;
   x = increment(x);
   printf("1+1 = %d", x);
}

int increment(int a) {
   a = a+1;
   return a;
}
```

It now prints `1+1 = 2`.

Parameter Passing – Arrays

- In C, you do not pass an entire array as a parameter – rather, you pass a “pointer” to the array. I.e., we pass the value of a pointer. (Pointers will be explained fully in Chapter 11)
- As part of the formal parameter, you can provide the dimension of the array.
  Ex: (http://www.youtube.com/watch?v=lyZQPjUT5B4)

```
main() {
 int i, num[10] =
   {3,0,1,8,7,2,5,4,6,9};
 bubblesort(num);
 for (i = 0; i<10; i++)
   printf(" %d ", num[i]);
}
```

```
const int N=10;
void bubblesort(int A[N]) {
  int i, j, temp;
  for (i=0; i<N-1; i++)
    for (j=0; j<N-i; j++)
      if (A[j] > A[j+1]) {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }
}
```

Parameter Passing – Arrays

- The problem with the previous implementation is that it will only sort a list of ten numbers.
- In order to make `bubblesort()` more useful, we should pass in the size of the list we wish to sort.
  Ex:

```
main()
{
 int i, num[10] =
   {3,0,1,8,7,2,5,4,6,9};
 int length =
   sizeof(num)/sizeof(num[0]);
 bubblesort(length, num);
 for (i = 0; i<length; i++)
   printf(" %d ", num[i]);
}
```

```
void bubblesort(int sz, int A[sz])
{
  int i, j, temp;
  for (i=0; i<sz-1; i++)
    for (j=0; j<sz-i; j++)
      if (A[j] > A[j+1]) {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }
}
```

Parameter Passing – Arrays

- Once again, remember that we can index beyond the bounds of the array, and the C compiler will not complain at compile time – this will either yield erroneous results, or a run-time error.

  Ex: If I change the 3rd line in `bubblesort()` to:

  ```
  for (i=0; i<sz; i++)
  ```

  the code compiles on my machine without warnings/errors, but it produces the wrong results.
  However, if I change it to:

  ```
  for (i=0; i<sz+100; i++)
  ```

  it again compiles without any problems, but I get a "segmentation fault" at run-time.

Parameter Passing – Multidimensional Arrays

- Because C always allocates array elements in contiguous memory locations, once we have the pointer to a one dimensional array, we can just index into it.
- The situation is different for multi-dimensional arrays. In order to index into a multidimensional array, we need know every dimension except for the first.
  E.g., if we wish to pass the array:

  ```
  int ary[5][8]
  ```

  to a function, that function needs to know that every row contains eight integers. I.e., to determine where in memory ary[2][3] is, the following pointer arithmetic is performed: `ary + (2 * 8) + 3`.
  The function knows about the 2 and the 3, they were used when indexing the array, but how does it know about the 8?
  In the function declaration, we need to supply this information:

  ```
  void func(int A[][8]);
  ```

Parameter Passing – Multidimensional Arrays

- The function definition also needs to contain this information:

```
void func(int A[][8]) {
   ...
}
```

- What if we wanted to pass the three-dimensional array

```
float ary[4][6][10];
```

  to a function, what would the function prototype need to look like?

```
void func(float myary[][6][10]);
```

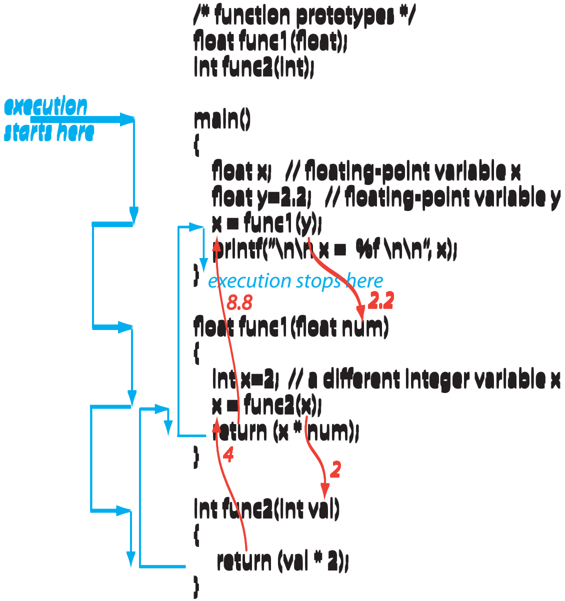
- This is a little ugly, since we hard-coded two of the three array dimensions, which tends to make the code less reusable. One alternative is to use variable sizing, as illustrated earlier.

Parameter Passing – How Does It Actually Work?

- Recall our example of how functions work from Module #2 (next slide).
- Notice that `main()` calls `func1()`, and `func1()` calls `func2()`. Furthermore, the value returned by `func2()` is used in `func1()`, and then value returned by `func1()` is used in `main()`.
- How does the program keep track of all of these values across multiple function calls while program is running?
- Every function call in C creates an activation record. Each time a function is called, the state of the function that made the call is recorded in its activation record, and then stored on the run-time stack for later retrieval.
- Think of a "stack" as being organized like a stack of cafeteria trays. The next one to be removed is the one on top, which must also be the tray that was most recently added.

THE UNIVERSITY of NEW MEXICO

Functions in C



```
/* function prototypes */
float func1(float);
int func2(int);

main()
{
   float x;   // floating-point variable x
   float y=2.2;  // floating-point variable y
   x = func1(y);
   printf("\n\n x =  %f \n\n", x);
}

float func1(float num)
{
   int x=2;  // a different integer variable x
   x = func2(x);
   return (x * num);
}

int func2(int val)
{
    return (val * 2);
}
```

The Run-time Stack

- An activation record contains storage for all local variables declared in the function, as well as either a copy of, or a reference to, the parameters that are being passed to the procedure.
- An activation record must also contain some information that specifies where program execution will resume when the function is completed.
- At the completion of the function, the associated activation record will be removed from the run-time stack, and program control will return to the point specified in the activation record.
- Activation records are managed using a stack (a LIFO data structure). Why? Upon completing execution of a function, control should return to the function that called it — which will be the one most recently stored on the run-time stack.

THE UNIVERSITY of NEW MEXICO

The Run-time Stack

Management of the run-time stack in the previous example:

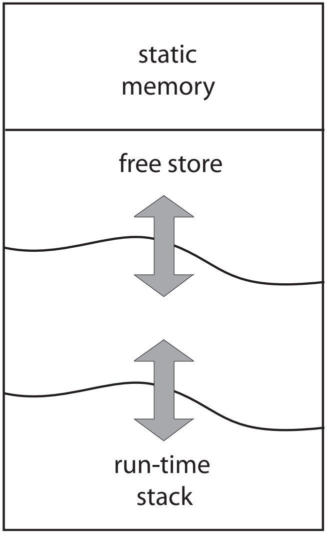
```
activation
  record
    1          main()
    2              func1()
    3                  func2()
    3                      return(4)
    2                  x=4
    2                  return 8.8
    1              x = 8.8
```

Activation record 1 is placed on the stack when `func1()` is called. Then activation record 2 is placed on top of it, when `func2()` is called. When `func2()` completes execution, activation record 2 is popped from the stack, and when it finishes, activation record 1 is popped from the stack. Why a stack? We always want to remove the most recently added activation record.

THE UNIVERSITY of NEW MEXICO

A More Complete C Memory Model

The memory model you should have when programming in C:



- Static Memory – Variables and constants that will persist in memory throughout the execution of the program.
- Free Store – Space set aside for dynamically allocated memory.
- Run-time Stack – Each time a function is called in a program, an activation record is created and stored in computer memory on the run-time stack.