EECS 22: Advanced C Programming

Lecture 6

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 6: Overview

- Functions
  - Terms and concepts
  - Scope
- Variable Lifetimes
- Memory Organization
  - Memory segmentation
  - Memory errors
- Storage Classes
- Program Example **`Fibonacci2`**

Functions

- Review: Terms and Concepts
  - Function declaration
    - function prototype with name, parameters, and return type
  - Function parameters
    - formal parameters holding the data supplied to a function
  - Function definition
    - extended declaration, defines the behavior in function body
  - Local variables
    - variables defined locally in a function body (compound statement)
  - Function call
    - expression invoking a function with supplied arguments
  - Function arguments
    - arguments passed to a function call (initial values for parameters)
  - Return value
    - result computed by a function call, passed to the caller

Functions

- Review: Terms and Concepts (continued)
  - Pass by value
    - A copy of the value in the argument is passed to the parameter
      - ➢ Changes to the parameter do not affect the argument
    - In C, basic types (and structures) are passed by value
  - Pass by reference
    - A reference to the argument is passed to the parameter
      - ➢ Changes to the parameter do affect the argument
    - In C, array types (and data via pointers) are passed by reference
  - Function call graph
    - Graphical representation of functions (nodes) and calls (edges)
  - Function call trace
    - Sequence of function calls logged during the program run-time
  - Function call stack
    - Stack of frames keeping track of active function calls

Scope

- *Scope* of an identifier
  - Portion of the program where the identifier can be referenced
  - aka. *accessability*, *visibility*
- Variable scope examples
  - Global variables: *file scope*
    - Declaration outside any function (at global level)
    - Scope in entire translation unit after declaration
  - Function parameters: *function scope*
    - Declaration in function parameter list
    - Scope limited to this function body (entirely)
  - Local variables: *block scope*
    - Declaration inside a compound statement (i.e. function body)
    - Scope limited to this compound statement block (entirely)

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Header file inclusion

Function declarations

Global variables

Function definition
Local variable

Function definition
Local variable

Function definition
Local variable

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of global functions **`printf()`**, **`scanf()`**, etc.

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of global function **`square()`**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of global function **add_y()**

EECS22: Advanced C Programming, Lecture 6 (c) 2014 R. Doemer 9

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of global variable **x**

EECS22: Advanced C Programming, Lecture 6 (c) 2014 R. Doemer 10

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of global variable **y**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of parameter **a**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of local variable **s**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Local variables are independent!
(unless their scopes are nested)

Scope of local variable **s**

Scope of local variable **s**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Local variables are independent!
(unless their scopes are nested)

Scope of local variable **s**

Scope of local variable **s**

Scope of local variable **z**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Scope of parameter **x**

Scope Example

```
#include <stdio.h>
int square(int a);
int add_y(int x);
int x = 5,
    y = 7;
int square(int a)
{  int s;
   s = a * a;
   return s;
}
int add_y(int x)
{  int s;
   s = x + y;
   return s;
}
int main(void)
{  int z;
   z = square(x);
   z = add_y(z);
   printf("%d\n", z);
   return 0;
}
```

Shadowing!
In nested scopes,
inner scope takes precedence!

Scope of global variable
x

Scope of parameter
x

Variable Lifetimes

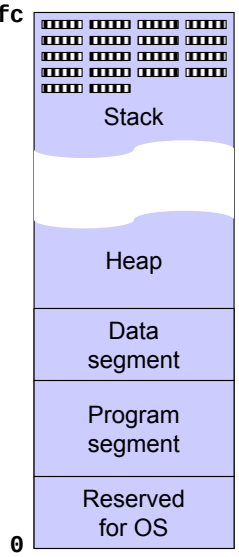
- Lifetime of Variables
  - Begins with *allocation*
    - Assignment of a (new) address in memory
  - Ends with *deallocation*
    - Memory is freed, address is marked as unused
  - ➢ Initialization: first access must be a write-access
    - ➢ Otherwise, variable value is undefined!
  - ➢ Access to a variable before or after its lifetime results in undefined behavior!
- ➢ Don't confuse *Variable Lifetime* with *Variable Scope*!
  - ➢ Variable Scope is determined at compile time
  - ➢ Variable Lifetime is determined at run time

Variable Lifetimes

- Lifetime of Variables
  - Global variables (storage class **`static`**, **`extern`**)
    - From program start to end
  - Local variables (storage class **`register`**, **`auto`**)
    - From beginning of execution of their compound statement
      - Stack frame entry
    - To leaving their compound statement
      - Stack frame exit
  - Function parameters (storage class **`register`**, **`auto`**)
    - From beginning of function call
    - To returning from the function call
  - Dynamically allocated objects (more details in Lecture 12)
    - From successful return of **`malloc()`**
    - To call of **`free()`**

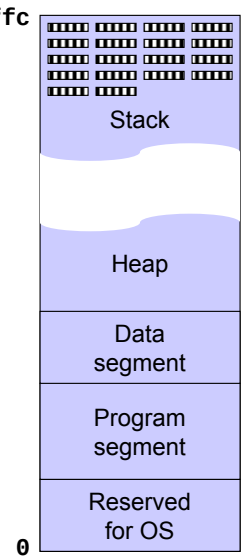
Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 4 GB of memory
  - Stack
    - grows and shrinks dynamically (from top)
    - contains function call hierarchy
    - stores stack frames with local variables
  - Heap
    - "free" storage
    - dynamic allocation by the program
  - Data segment
    - global (and **`static`**) variables
  - Program segment (aka. text segment)
    - program instructions (binary code)
  - Reserved area for operating system



Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 4 GB of memory
- Memory errors
  - *Out of memory*
    - Stack and heap collide
  - *Segmentation fault*
    - access outside allocated segments
    - e.g. access to segment reserved for OS
  - *Bus error*
    - mis-aligned word access
    - e.g. word access to an address that is not divisible by 4



Storage Classes

- C Language distinguishes 2 Storage Classes
  - (but uses 5 keywords and a default, depending on scope)
  - Automatic (i.e. on the stack)
    - **`auto`** local variable, on stack (default)
    - **`register`** local variable, in register (preferred) or on stack
  - Static (i.e. in the data segment)
    - **`static`** static variable in data segment
    - **`extern`** declaration of global variable in data segment
  - At compile-time, a 3rd "storage class" exists
    - **`typedef`** definition of an alias for a type at compile time (no storage)

Storage Classes

Keyword	Global Scope	Local Scope
(none)	Global variable/function in data segment (ext. linkage)	Local variable on stack
`auto`	n/a	Local variable on stack
`register`	n/a	Local variable on stack or in register (preferred)
`static`	Global variable/function in data segment (int. linkage)	Local variable in data segment
`extern`	Decl. of global variable/function in data segment (ext. linkage)	Decl. of global variable/function in data segment (ext. linkage)
`typedef`	Alias for a type at compile time (no storage in memory)	Alias for a type at compile time (no storage in memory)

Storage Classes

- Program example: **`StorageClasses.c`** (part 1/3)

```
/* StorageClasses.c: example for storage classes and linkage */
/* author: Rainer Doemer                                     */
/*                                                           */
/* modifications:                                            */
/* 10/13/13 RD  initial version                              */

/*** global scope ***/

       void f(int); /* global function (defined below) */
extern void g(int); /* global function (defined somewhere else)*/
static void h(int); /* internal function (defined below) */

        double x;   /* global variable (defined here) */
extern  double y;   /* global variable (defined somewhere else)*/
static  double z;   /* internal global variable (defined here) */

typedef double t;   /* type definition */

...
```

Storage Classes

- Program example: **`StorageClasses.c`** (part 2/3)

```
...
void f(int p)
{
    /*** local scope ***/

            int i; /* local variable (on stack) */
    auto    int j; /* local variable (on stack) */
    register int r; /* local variable, preferably in register */
    static  int n = 0; /* static local variable */

    n++; /* count executions of this function */
    for(i=0; i<n; i++)
    { for(j=0; j<p; j++)
      { g(i*j);
      }
    }
    for(r=0; r<1000000; r++)
    { h(r);
    }
}
...
```


Storage Classes

- Program example: **`StorageClasses.c`** (part 3/3)

```
...

static void h(int p)
{
    g(p + (x*y*z));
}

/* EOF */
```

Program Example

- Example Revisited: Fibonacci series
  - Recursive definition:
    - Base case: *fibonacci(0) = 0, fibonacci(1) = 1*
    - Recursion step: *fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)*
  - Mathematical properties:
    - The first two numbers are 0 and 1
    - Every subsequent number is the sum of the *previous* two
  - ➢ Problem:
    - Recursive calculation time grows exponentially!
  - ➢ Idea:
    - If we remember previously calculated numbers, we can calculate the next number immediately!
    - ➢ Whenever a new number is calculated, keep it stored in a **`static`** array in memory
    - ➢ When a number is present in the memory, just look it up

Program Example

- Program example: **`Fibonacci2.c`** (part 1/3)

```
/* Fibonacci2.c: example demonstrating recursion */
/* author: Rainer Doemer                         */
/* modifications:                                */
/* 11/09/11 RD  version with 'static' memory     */
/* 11/14/04 RD  initial version                  */

#include <stdio.h>

#define MEM_SIZE 100

/* function definition */

...
```

Program Example

- Program example: **`Fibonacci2.c`** (part 2/3)

```
long fibonacci(long n)
{  static long fib[MEM_SIZE] = {0,1}; /* memory */
   if (n <= 1)          /* base case */
      { return n;
       } /* fi */
   else                 /* previouly calculated results */
      { if (n < MEM_SIZE && fib[n])
           { return fib[n];
            } /* fi */
        else            /* recursion step */
           { long f;
             f = fibonacci(n-1) + fibonacci(n-2);
             if (n < MEM_SIZE)
                { fib[n] = f;   /* remember this */
                 } /* fi */
             return f;
            } /* esle */
       } /* esle */
} /* end of fibonacci */
...
```


Program Example

- Program example: **`Fibonacci2.c`** (part 3/3)

```
...
int main(void)
{
   /* variable definitions */
   long int n, f;

   /* input section */
   printf("Please enter value n: ");
   scanf("%ld", &n);

   /* computation section */
   f = fibonacci(n);

   /* output section */
   printf("The %ld-th Fibonacci number is %ld.\n", n, f);

   /* exit */
   return 0;
} /* end of main */

/* EOF */
```

Program Example

- Example session: **`Fibonacci2.c`**

```
% cp Fibonacci.c Fibonacci2.c
% vi Fibonacci2.c
% gcc Fibonacci2.c -o Fibonacci2 -Wall -ansi
% Fibonacci2
Please enter value n: 20
The 20-th Fibonacci number is 6765.
% Fibonacci2
Please enter value n: 30
The 30-th Fibonacci number is 832040.
% Fibonacci2
Please enter value n: 40
The 40-th Fibonacci number is 102334155.
% Fibonacci2
Please enter value n: 50
The 50-th Fibonacci number is 12586269025.
% Fibonacci2
Please enter value n: 60
The 60-th Fibonacci number is 1548008755920.
%
```