EECS 22: Advanced C Programming

Lecture 6 (TuTh – Tentative)

Rainer Dömer

doemer@uci.edu

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

Overview

- Functions
  - Terms and concepts
- Recursion
  - Concept of recursion
  - Example program **`Fibonacci.c`**
- Scope
- Variable Lifetimes
- Memory Organization
  - Memory segmentation
  - Memory errors
- Storage Classes

Functions

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

Recursion

- Introduction
  - *Recursion* is often an alternative to *Iteration*
  - Recursion is a very simple concept, yet very powerful
  - Recursion is present in nature
    - Trees have branches, which have branches, which have branches, ... which have leaves.
  - Recursion is traversal of hierarchy
    - *Traverse* (climb) a tree to the top:
      - start at the root
      - at a leaf, stop
      - at a branch, *traverse* one branch
    - *Traverse* a file system on a computer
      - start at the current directory
      - at a file, process the file
      - at a directory, *traverse* the directory

Recursion

- Recursive Function
  - Function that calls itself ...
    - ... directly, or
    - ... indirectly
  - ➢ *Cycle* in function call graph!

```
int f(...)
  { ...
    f(...);
    ...
   }
```

```
int a(...)
  { ...
    b(...);
    ...
   }
int b(...)
  { ...
    a(...);
    ...
   }
```

- Concept of Recursion
  - Trivial *base case*
    - Return value defined for simple case
    - Example: **`if (arg == 0) {return 1; }`**
  - *Recursion step*
    - Reduce the problem towards the base case
    - Make a recursive function call
    - Example: **`if (arg > 0) { return ...fct(arg-1); }`**
  - *Termination* of recursion
    - Converging of recursive calls to the base case
    - Recursive call must be “simpler” than current call

Recursion

- Example: Fibonacci series
  - Sequence of integers
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...
  - Mathematical properties
    - The first two numbers are 0 and 1
    - Every subsequent Fibonacci number is the sum of the previous two Fibonacci numbers
  - Ratio of successive Fibonacci numbers is ...
    - ... converging to constant value 1.618...
    - ... called *Golden Ratio* or *Golden Mean*
  - Recursive definition:
    - Base case: $fibonacci(0) = 0$
      $fibonacci(1) = 1$
    - Recursion step: $fibonacci(n) = fibonacci(n\text{-}1) + fibonacci(n\text{-}2)$

Recursion

- Program example: **`Fibonacci.c`** (part 1/2)

```
/* Fibonacci.c: example demonstrating recursion */
/* author: Rainer Doemer                          */
/* modifications:                                 */
/* 11/14/04 RD  initial version                   */

#include <stdio.h>

/* function definition */

long fibonacci(long n)
{
   if (n <= 1)  /* base case */
      { return n;
       } /* fi */
   else         /* recursion step */
      { return fibonacci(n-1) + fibonacci(n-2);
       } /* esle */
} /* end of fibonacci */

/* main function */
...
```

Recursion

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

```
...
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 */
```

Recursion

- Example session: **Fibonacci.c**

```
% cp Factorial.c Fibonacci.c
% vi Fibonacci.c
% gcc Fibonacci.c -o Fibonacci -Wall –ansi –std=c99
% Fibonacci
Please enter value n: 1
The 1-th Fibonacci number is 1.
% Fibonacci
Please enter value n: 10
The 10-th Fibonacci number is 55.
% Fibonacci
Please enter value n: 20
The 20-th Fibonacci number is 6765.
% Fibonacci
Please enter value n: 30
The 30-th Fibonacci number is 832040.
% Fibonacci
Please enter value n: 40
The 40-th Fibonacci number is 102334155.
%
```

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>`	Header file inclusion
`int square(int a);` `int add_y(int x);`	Function declarations
`int x = 5, y = 7;`	Global variables
`int square(int a)`	Function definition
`{ int s;`	Local variable
`int add_y(int x)`	Function definition
`{ int s;`	Local variable
`int main(void)`	Function definition
`{ int z;`	Local variable

```
#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 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

```c
#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()**

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**

Scope Example

```c
#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

```c
#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

```c
#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

```c
#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
  - ➢ Access to a variable before or after its lifetime results in undefined behavior!
  - ➢ Initialization: first access must be a write-access
    - ➢ Otherwise, variable value is undefined!

➢ Don't confuse *Variable Lifetime* with *Variable Scope*!

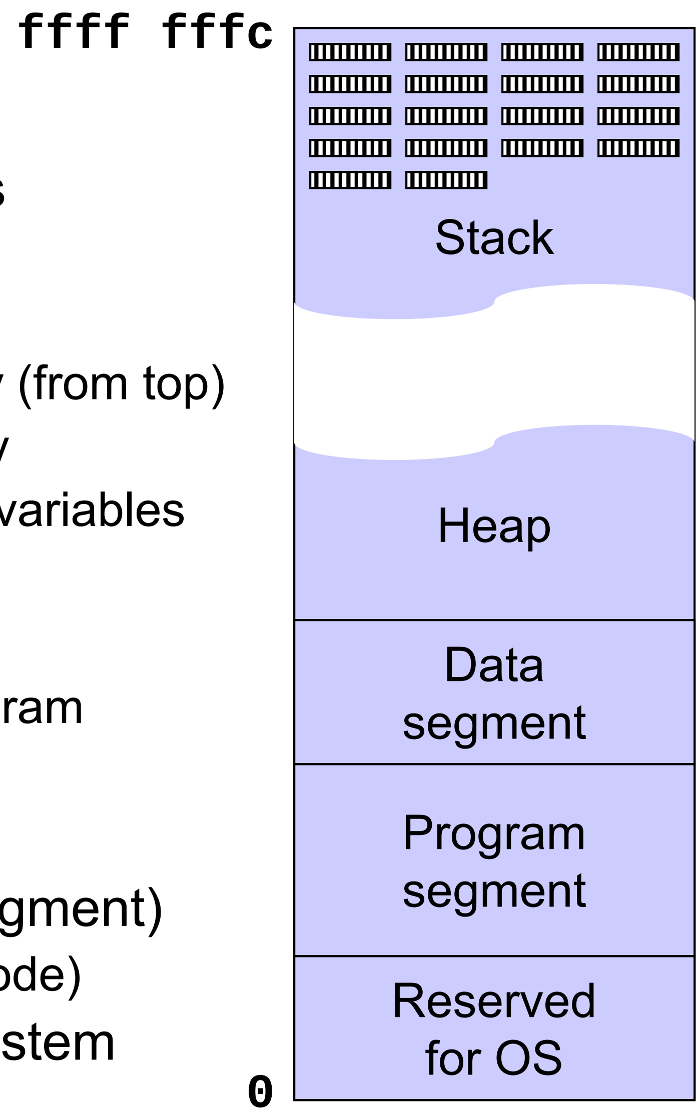
- ➢ Variable Scope is determined at compile time
- ➢ Variable Lifetime is determined at run time

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 17)
    - From successful return of **`malloc()`**
    - To the corresponding call of **`free()`**

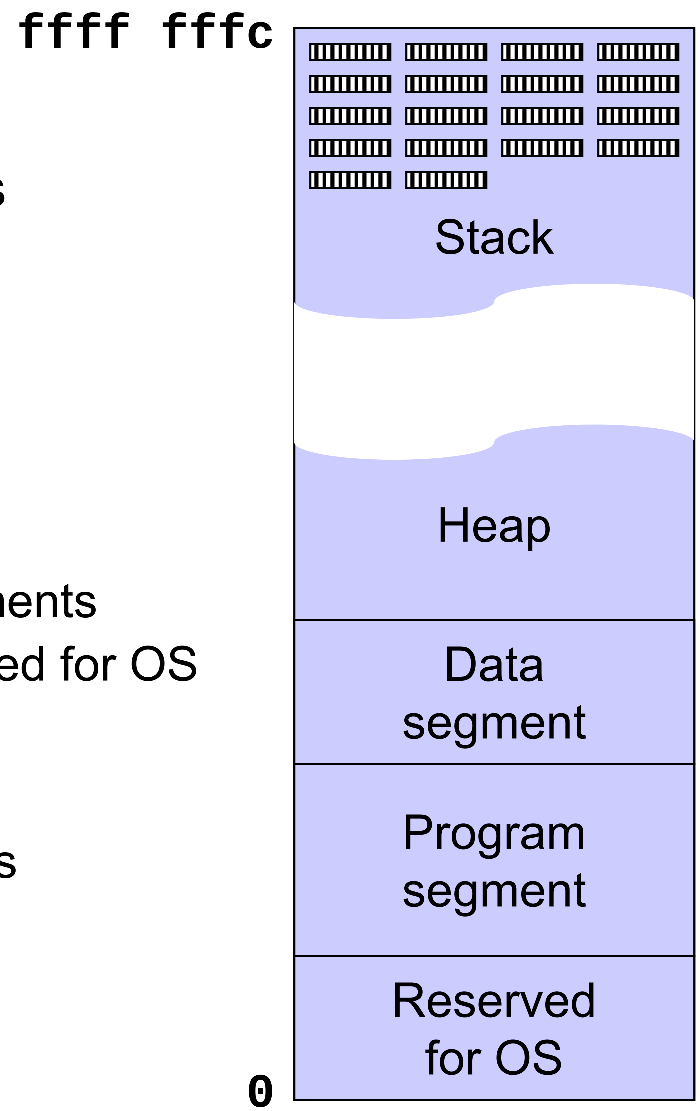
Memory Organization

- 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 */
```