


CS 135: Computer Architecture I


Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs135/



Concept of Scope of Variable

- In assembly, who has access to a memory location/variable ?
- In high level programs, who has access to a variable ?
 - Concept of Scope of a variable


CS 135



Scope: Global and Local

- Where is the variable accessible?
 - **Global:** accessed anywhere in program
 - **Local:** only accessible in a particular region
- **Compiler infers scope from where variable is declared**
 - programmer doesn't have to explicitly state
- **Variable is local to the block in which it is declared**
 - block defined by open and closed braces { }
 - can access variable declared in any "containing" block
- **Global variable is declared outside all blocks**


CS 135



Allocation of Variables

- Simply assigning a memory location for each variable may not be enough to enforce scope
- Need to look at a better scheme to allocate high level program variables to memory in the processor

CS 135



Example

```

#include <stdio.h>
int itsGlobal = 0;

main()
{
  int itsLocal = 1; /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  {
    int itsLocal = 2; /* local to this block */
    itsGlobal = 4; /* change global variable */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
  }
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}

```


Output

```

Global 0 Local 1
Global 4 Local 2
Global 4 Local 1


```

CS 135



Where can you put variables?


CS 135



Where do you want to go today?

- **Local variable inside a function**
 - Lasts only while the function is running
- **Local variable inside a function**
 - Value persists throughout the life of the program
- **Global variable visible to all functions within a file**
- **Global variable visible in more than one file**


CS 135



Automatic Variables


- **Local variable inside a function (lasts only while the function is running)**
- **auto keyword (never used!)**
- **Located on stack**
- **Storage Class: AUTO**

CS 135

 **Static Variables (I)**


- Local variable inside a function (value persists throughout the life of the program)
- `static` keyword declared inside a function
- Located in static area
- Storage Class: **STATIC**

CS 135

 **Static Variables (II)**


- Global variable visible to all functions within a file
- `static` keyword declared outside of any function
- Located in static area
- Storage Class: **STATIC**

CS 135

 **Static Variables (III)**

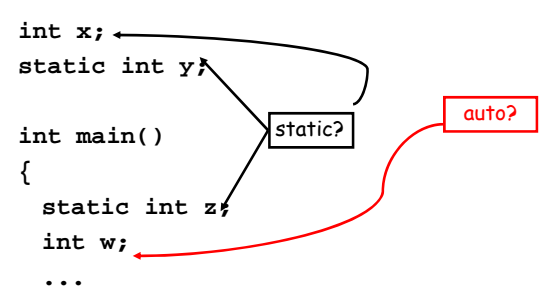
- Global variable visible in more than one file
- Declared outside of any function in one file
- Declared `extern` in other files (or functions/blocks)
- Located in static area
- Storage Class: **STATIC**

CS 135

 **Confused?**

```
int x;
static int y;

int main()
{
    static int z;
    int w;
    ...
}
```



CS 135

auto

```

int foo(int z)
{
  int x;
  ...
  if(x == z)
  {
    int y;
    y = ...
  }
}

```

auto variables
(implicitly)

CS 135

static

- **static variables exist in an area of memory set aside for alterable (or readable/writeable) data**
- **static can have different meanings depending on where used.**

CS 135

static

- **Variables**
 - > A global variable is stored in the static area of memory
 - > A variable that is global just to the functions in one file must be designated using static
 - > A variable that is local to a function AND retains its value (i.e. persistence) must be labeled static
- **Functions**
 - > A function labeled as static is visible only within the file where it is defined

CS 135

static

```

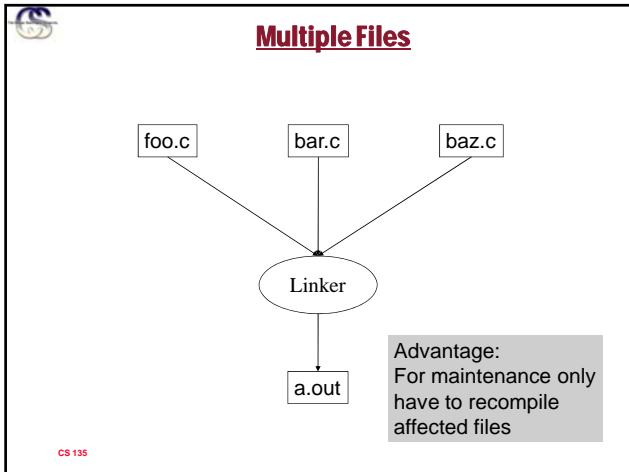
int i; /* Global variable...stored in
       * static area (storage class is
       * static
       */

int foo(...)
{
  ...
}

int main()
{
  ...
}

```

CS 135



- ### Multiple File Scope
- Scope can be global within a file
 - Scope can be global across all files
 - Scope can be defined illegally!
- Actually Linkage
 - > None
 - > Internal
 - > External
- Note: global here means visible in more than one function
- CS 135

Scope can be global within a file

These are different variables with Global scope within their respective files

<pre> static int i; int foo(...) { ... } int bar(...) { ... } </pre>	<pre> static int i; int main() { ... } int baz(...) { ... } </pre>
--	--

CS 135

Scope can be global across all files

<pre>int i;</pre>	<pre>extern int i;</pre>	<pre>void f() { extern int i; ... }</pre>
-------------------	--------------------------	---

These are the same variable!

CS 135

According to the C89 Standard...

- There are 4 types of scope
 - > **Prototype Scope**
 - > Just applies to prototypes
 - > **File Scope**
 - > Declared outside any function
 - > **Function Scope**
 - > only applies to labels!!!
 - > **Block Scope**
 - > includes function parameters

CS 135

Scope vs. Lifetime

Scope	Lifetime
• File Scope	• Life of program
• Block Scope	• Life of block

CS 135

Scope vs. Lifetime

Scope	Lifetime
• File Scope	• Life of program
• Block Scope	• Life of block

Can be overridden with "static"

CS 135

Memory

- Many languages have fixed mappings between scopes and lifetimes
- In C, we have the option to decide...


```
void foo(void) {
    int x;
    static int a;
    x = 42;
}
```
- When the function goes away, x and its value go away since they were on stack.
- Setting a value into a static variable (e.g. a) means that the value is stored in a static area and will persist throughout the entire execution of the program

CS 135



Static Initialization

```
void foo(void)
{
    int x = 10;
    static int y = 20;
```

- Both variables are initialized as allocated
- x is initialized to 10 every time function foo runs
- y is initialized when program starts

CS 135



Static Initialization?

```
void foo(void)
{
    int x = 10;
    static int y;
    y = 20;
```

- This would defeat purpose of static variable having persistence!

CS 135



Static Initialization

```
void foo(void)
{
    int x = 10;
    static int y = 20;
    printf("x = %d y = %d\n", x, y);
    y += 30;
}
```

- What prints the first time foo is called?
- What prints the second time?


CS 135



Other goodies

- **register**
 - > Suggests to compiler that this variable should be kept in a register
 - > Cannot get address of variable declared register
 - > Not as important as it once was with modern compilers
- **volatile**
 - > Type qualifier
 - > Tells compiler that value in this variable may change on its own!
 - > Used in
 - > shared memory applications
 - > Memory mapped I/O
- Will return to these at end of course...
 - > Read on your own for now.

CS 135




Storage Class Specifiers

register
auto
static
extern

const
volatile

Type Qualifiers


CS 135



Allocating variables in Memory

- How to allocate memory locations to variables
 - scope


CS 135



Allocation of Variables

- Simply assigning a memory location for each variable may not be enough to enforce scope
- Need to look at a better scheme to allocate high level program variables to memory in the processor

CS 135



Allocating variables in Memory

- How to allocate memory locations to variables such that usage rules are satisfied:
 - Scope
 - Type
 - Storage class

CS 135

Memory Representation

- We typically draw diagrams representing the memory of the computer, the memory of our particular program or both as rectangles.
- Our convention will be that "high-memory" will be on the bottom and "low-memory" on top.
- drawings are not to scale

x0000
Low Memory
High Memory
xFFFF

CS 135

Typical Arrangement

- Normally the actual program code (executable instructions) is placed in low memory

x0000
Code
xFFFF

CS 135

Typical Arrangement

- Next we have an area for storage of constant data
 - Also data such as "hello" in the printf("hello world") would be stored

x0000
Code
Constant Data
xFFFF

CS 135

Typical Arrangement

- Data that may be changed follows

x0000
Code
Constant Data
Alterable Data
xFFFF

CS 135

Typical Arrangement

- These three items comprise what is considered the static area of memory. The static area details (size, what is where, etc.) are known at translation or compile time.

x0000

Static

Code

Constant Data

Alterable Data

xFFFF

CS 135

Typical Arrangement

- Immediately above the static area the heap is located.
- The heap can expand upward as the program dynamically requests additional storage space
- In most cases, the runtime environment manages the heap for the user

x0000

Static

Code

Constant Data

Alterable Data

Heap

xFFFF

CS 135

Typical Arrangement

- Finally, the **run-time stack** starts in high memory and can grow down as space is needed.
- Items maintained in the stack include
 - > Local variables
 - > Function parameters
 - > Return values

x0000

Static

Code

Constant Data

Alterable Data

Heap

Stack

xFFFF

CS 135

auto

- auto, short for automatic variables are those that exist on the stack.
- Auto means that space is allocated and deallocated on the stack automatically without the programmer having to do any special operations.

x0000

Static

Code

Constant Data

Alterable Data

Heap

Stack

xFFFF

CS 135

Typical Arrangement

- These items in the upper portion of the diagram change* during execution of the program.
- Thus they are called dynamic

*Not just their value

Offset?

- Assembly code written by a compiler usually looks a little different from assembly code written by hand
- Registers are dedicated to point to key areas of memory
- R4 is the Global Pointer

CS 135

Keeping Track of auto variables

- Stack pointer is obvious but the compiler writer needs more info...
- Where is the activation stack frame?
 - i.e., where is the start of the block (i.e., scope)

CS 135

Why do we care?

- We would like to know where a variable is throughout the execution of a function
- Why not just reference the variable from the stack pointer?

CS 135

The Frame Pointer

```

int f(int a, int b) {
    int c;
    c = a + b;
    return c;
}

int main() {
    int x;
    int y = 4;
    x = f(7, y);
    printf("%d\n", x);
    return 0;
}

```

- What do we need to keep track of?

CS 135

Frame Pointer

- The Frame Pointer designates a fixed spot in the activation stack which can be used as a reference throughout execution of the function.

x0000
xFFFF

CS 135

LC3: Local Variable Storage

- Local variables are stored in an **activation record**, for each code block also known as a **stack frame**.
 - Cannot afford to forget about the Stack ☹
- Symbol table “offset” gives the distance from the base of the frame.
 - R5 is the **frame pointer** – holds address of the base of the current frame.
 - A new frame is pushed on the **run-time stack** each time a block is entered.
 - Because stack grows downward, base is the highest address of the frame, and variable offsets are ≤ 0 .

seconds
minutes
hours
time
rate
amount

R5 →

CS 135

LC3: Allocating Space for Variables

- **Global data section**
 - All global variables stored here (actually all static variables)
 - R4 points to beginning
- **Run-time stack**
 - Used for local variables
 - R6 points to top of stack
 - R5 points to top frame on stack
 - New frame for each block (goes away when block exited)
- **Offset = distance from beginning of storage area**
 - Global: `LDR R1, R4, #4`
 - Local: `LDR R2, R5, #-3`

0x0000
0xFFFF

CS 135



Compiler Magic

- The compiler has the job of converting your C program into assembly code
- Thus it must convert the symbolic variable names into addresses
- How does it keep track of what is where?

CS 135



Symbol Table

- For each variable keeps track of
 - > Type
 - > Location (as an offset)
 - > Scope
 - > Other info (const, etc.)

CS 135



Variables and Memory Locations

- In our examples, a variable is always stored in memory.
- When assigning to a variable, must store to memory location.
- A real compiler would perform code optimizations that try to keep variables allocated in registers.
 - > Why?

CS 135



Example: Compiling to LC-3

```

#include <stdio.h>
int inGlobal;

main()
{
  int inLocal; /* local to main */
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n",
        outLocalA, outLocalB);
}

```

CS 135

Example: Symbol Table

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

CS 135

Example: Code Generation

- ; main
- ; initialize variables
- ```

 AND R0, R0, #0
 ADD R0, R0, #5 ; inLocal = 5
 STR R0, R5, #0 ; (offset = 0)

 AND R0, R0, #0
 ADD R0, R0, #3 ; inGlobal = 3
 STR R0, R4, #0 ; (offset = 0)

```

CS 135

### Example (continued)

- ; first statement:
- ; outLocalA = inLocal++ & ~inGlobal;
- ```

      LDR R0, R5, #0 ; get inLocal
      ADD R1, R0, #1 ; increment
      STR R1, R5, #0 ; store

      LDR R1, R4, #0 ; get inGlobal
      NOT R1, R1 ; ~inGlobal
      AND R2, R0, R1 ; inLocal & ~inGlobal
      STR R2, R5, #-1 ; store in outLocalA
                       ; (offset = -1)
      
```

CS 135

Example (continued)

- ; next statement:
- ; outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
- ```

 LDR R0, R5, #0 ; inLocal
 LDR R1, R4, #0 ; inGlobal
 ADD R0, R0, R1 ; R0 is sum
 LDR R2, R5, #0 ; inLocal
 LDR R3, R5, #0 ; inGlobal
 NOT R3, R3
 ADD R3, R3, #1
 ADD R2, R2, R3 ; R2 is difference
 NOT R2, R2 ; negate
 ADD R2, R2, #1
 ADD R0, R0, R2 ; R0 = R0 - R2
 STR R0, R5, #-2 ; outLocalB (offset = -2)

```

CS 135

**Next.**

- How to handle function calls ?
  - Caller save and callee save concepts again!
  - Where to store the data?
- Example C code translated to LC3
- what can go wrong ?

CS 135

**Course announcements..**

- Next class: Implementing Pointers, Arrays, Dynamic data structures
  - Allocating space during run-time
- HW5 posted
  - Code generation
- HW 6 posted
  - Implement project 2 in C
- Project 3 will be posted next week
  - Work individually – no collaboration of any kind
  - This is just a C programming assignment
    - Play with pointers, dynamic allocation, and file I/O
- Exam 2: Thursday Nov.18th

CS 135

**Functions in C**

CS 135

**LC3: Allocating Space for Variables**

- **Global data section**
  - All global variables stored here (actually all static variables)
  - R4 points to beginning
- **Run-time stack**
  - Used for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - New frame for each block (goes away when block exited)
- **Offset = distance from beginning of storage area**
  - Global: `LDR R1, R4, #4`
  - Local: `LDR R2, R5, #-3`

CS 135

## Implementing Functions

- Functions in C
- Implementing Functions in C
  - Run-time Stack, Getting It All to Work, Tying it All Together
- Problem Solving Using Functions
- Recursion

CS 135

## Functions in C

- **Declaration** (also called prototype)
  - `int Factorial(int n);`
    - type of return value
    - name of function
    - types of all arguments
- **Function call** -- used in expression
  - `a = x + Factorial(f + g);`
    - 1. evaluate arguments
    - 2. execute function
    - 3. use return value in expression

CS 135

## Function Definition

- **State type, name, types of arguments**
  - must match function declaration
  - give name to each argument (doesn't have to match declaration)
- `int Factorial(int n)`
  - {
    - int i;
    - int result = 1;
    - for (i = 1; i <= n; i++)
    - result \*= i;
    - return result;
  - }


gives control back to calling function and returns value

CS 135

## Function calls.. What needs to be done?

- Caller can pass parameters to the function
- Function returns a value
- Function needs to return to caller
  - PC needs to be stored
  - "pointer" to variables used by caller needs to be restored
- Function uses local variables, so allocate space for these variables
- Function can be called from another function...
- model this behaviour and capture all this information in an **Activation Record**


CS 135



### Activation Record/Stack Frame

- Allows for recursion
- Place to keep
  - Parameters
  - Local (auto) variables
  - Register spillage
  - Return address
  - Return value
  - Old frame pointer


CS 135



### Caller and Callee: Who does what?

- Caller
  - Puts arguments onto stack (R→L)
  - Does a JSR (or JSRR) to function
- Callee
  - Makes space for Return Value and Return Address (and saves Return address)
  - makes space for and saves old FP (Frame Pointer)
    - Why ?
  - Makes FP point to next space
  - Moves SP enough for all local variables
  - Starts execution of "work" of function


CS 135



### Who does what?

- Callee (continued)
  - As registers are needed their current contents can be spilled onto stack
  - When computation done...
  - Bring SP back to base
  - Restore FP (adjust SP)
  - Restore RA (adjust SP)
  - Leave SP pointing at return value
  - RET

CS 135



### Who does what?

- Caller
  - Grabs return value and does the right thing with it!

CS 135

### Implementing Functions: Overview

- **Activation record**
  - > information about each function, including arguments and local variables
  - > stored on run-time stack

Calling function

push new activation record  
copy values into arguments  
call function

Called function

execute code  
put result in activation record  
pop activation record from stack  
return

get result from stack

CS 135

### Recall

- **R7: Return Address**
- **R6: Stack Pointer**
- **R5: Frame Pointer**
- **R4: Global Pointer**

CS 135

### Run-Time Stack

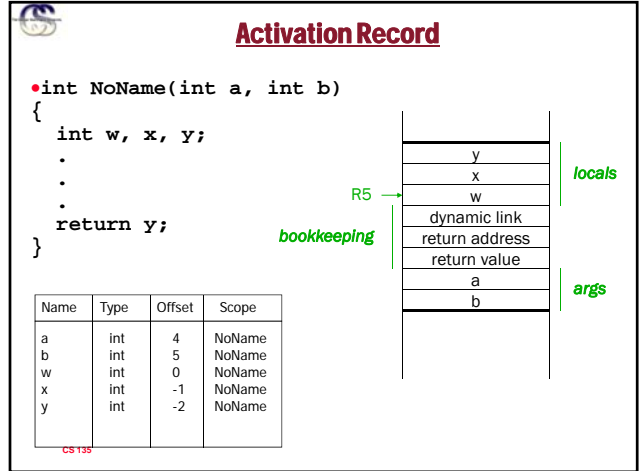
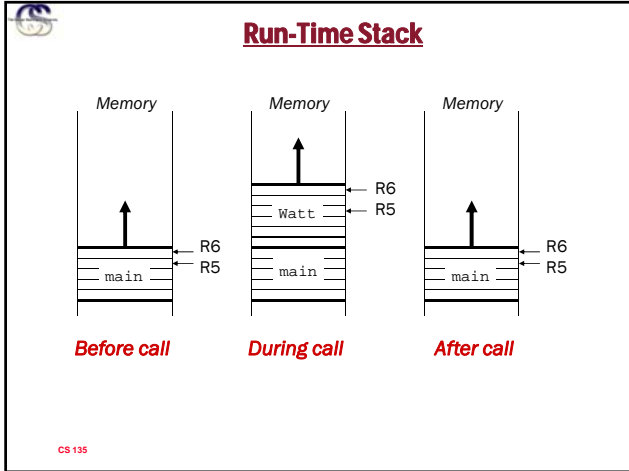
- Recall that local variables are stored on the run-time stack in an **activation record**
- **Frame pointer (R5)** points to the beginning of a region of activation record that stores local variables for the current function
- When a new function is **called**, its activation record is **pushed** on the stack; when it **returns**, its activation record is **popped** off of the stack.

CS 135

### Activation Record Bookkeeping

- **Return value**
  - > space for value returned by function
  - > allocated even if function does not return a value
- **Return address**
  - > save pointer to next instruction in calling function
  - > convenient location to store R7 in case another function (JSR) is called
- **Dynamic link**
  - > caller's frame pointer
  - > used to pop this activation record from stack

CS 135



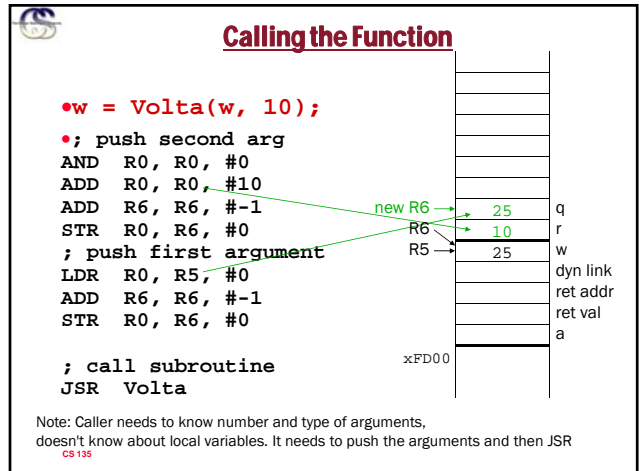
### Example Function Call

```

•int main
{int a,b;
...
b=Watt(a);
b=Volta(a,b);
}
•int Volta(int q, int r)
{
 int k;
 int m;
 ...
 return k;
}
int Watt(int a)
{
 int w;
 w = Volta(w,10);
 ...
 return w;
}

```

CS 135



CS 135

- Value of R5 ?
- a is at address: FCFF
- Ret.val.:??
- Ret.addr: ??
- Link: ??
- Frame pointer for Watt is: ??

CS 135

- Value of R5 ?
- a is at address: FCFF
- Ret.val.: FCFE
- Ret.addr: FCFD
- Link: FCFC
- Frame pointer for Watt is: FCFB

CS 135

### Next steps..

- Create space for return value
- Store return address
- Store frame pointer
- Set new frame pointer
- Set space for local variables

CS 135

### Starting the Callee Function

```

•; leave space for return value
ADD R6, R6, #-1
; push return address
ADD R6, R6, #-1
STR R7, R6, #0
; push dyn link (caller's frame ptr)
ADD R6, R6, #-1
STR R5, R6, #0
; set new frame pointer
ADD R5, R6, #-1
; allocate space for locals
ADD R6, R6, #-2

int Volta(int q, int r)
{
 int k;
 int m;
 ...
 return k;
}

```

### Returning from function...steps

- Write return value
- return address into R7
- Restore old frame pointer
- Pop local variables
- Where should top of stack point to after RET?

CS 135

### Ending the Callee Function

```

•return k;
•; copy k into return value
LDR R0, R5, #0
STR R0, R5, #3
; pop local variables
ADD R6, R5, #1
; pop dynamic link (into R5)
LDR R5, R6, #0
ADD R6, R6, #1
; pop return addr (into R7)
LDR R7, R6, #0
ADD R6, R6, #1
; return control to caller
RET

```

CS 135

### Back to caller...steps

- What should caller do now?
- Get return value
- Clear arguments

CS 135


### Resuming the Caller Function

```

•w = Volta(w,10);
•JSR Volta
; load return value (top of stack)
LDR R0, R6, #0
; perform assignment
STR R0, R5, #0
; pop return value
ADD R6, R6, #1
; pop arguments
ADD R6, R6, #2

```


CS 135



### Summary of LC-3 Function Call Implementation

1. **Caller** pushes arguments (last to first).
2. **Caller** invokes subroutine (JSR).
3. **Callee** allocates return value, pushes R7 and R5.
4. **Callee** allocates space for local variables.
5. **Callee** executes function code.
6. **Callee** stores result into return value slot.
7. **Callee** pops local vars, pops R5, pops R7.
8. **Callee** returns (JMP R7).
9. **Caller** loads return value and pops arguments.
10. **Caller** resumes computation...


CS 135



### For details ..

- Read Chapter 14 – section 14.3, Figure 14.8 for full implementation of the function call process
- Check out the lcc cross compiler


CS 135



### Question...What can go wrong?

- What if the return address was overwritten
  - Where does program return to ?
- Recall 'privilege' level – what if program was operating as root ?
  - Will level change ?
- Buffer overflow attack/stack smashing attack

CS 135



### How about recursion ?

CS 135

## What is Recursion?

- A recursive function is one that solves its task by **calling itself** on smaller pieces of data.
  - > Similar to recurrence function in mathematics.
    - > Question: How do you prove correctness of recurrence function ?? cs123!
  - > Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.
- Example: Running sum ( $\sum_1^n i$ )

|                                                                                                                                  |                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Mathematical Definition:</b><br/> <math>RunningSum(1) = 1</math><br/> <math>RunningSum(n) = n + RunningSum(n-1)</math></p> | <p><b>Recursive Function:</b></p> <pre>int RunningSum(int n) {     if (n == 1)         return 1;     else         return n + RunningSum(n-1); }</pre> |
|----------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|

CS 135

## Executing RunningSum

The diagram illustrates the execution of the RunningSum function for n=4. It shows a sequence of recursive calls and return values:

- Initial call: `res = RunningSum(4);`
- Call 1: `RunningSum(4)` calls `RunningSum(3)`. Return value = 10.
- Call 2: `RunningSum(3)` calls `RunningSum(2)`. Return value = 6.
- Call 3: `RunningSum(2)` calls `RunningSum(1)`. Return value = 3.
- Call 4: `RunningSum(1)` returns 1.

The return values are propagated back up the call stack, and the final result is 10.

CS 135

## Example: Fibonacci Numbers

- **Mathematical Definition:**

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = 1$$

$$f(0) = 1$$
- In other words, the n-th Fibonacci number is the sum of the previous two Fibonacci numbers.
  - > To compute value of  $F(n)$  we need values of  $F(n-1)$  and  $F(n-2)$

CS 135

## Fibonacci: C Code

```
int Fibonacci(int n)
{
 if ((n == 0) || (n == 1))
 return 1;
 else
 return Fibonacci(n-1) + Fibonacci(n-2);
}
```

CS 135

### How is recursion implemented ?

- Do we need to do anything different from how we handled function calls ?
- No!
  - Activation record for each instance/call of Fibonacci !

CS 135

### Activation Records

- Whenever Fibonacci is invoked, a new activation record is pushed onto the stack.

The diagram illustrates the stack state at three points:
 

- main calls Fibonacci(3):** The stack contains 'main' at the bottom and 'Fib(3)' above it. The register R6 points to the top of the 'Fib(3)' record.
- Fibonacci(3) calls Fibonacci(2):** The stack now contains 'main', 'Fib(3)', and 'Fib(2)'. R6 points to the top of the 'Fib(2)' record.
- Fibonacci(2) calls Fibonacci(1):** The stack now contains 'main', 'Fib(3)', 'Fib(2)', and 'Fib(1)'. R6 points to the top of the 'Fib(1)' record.

CS 135

### Activation Records (cont.)

The diagram illustrates the stack state during return operations:
 

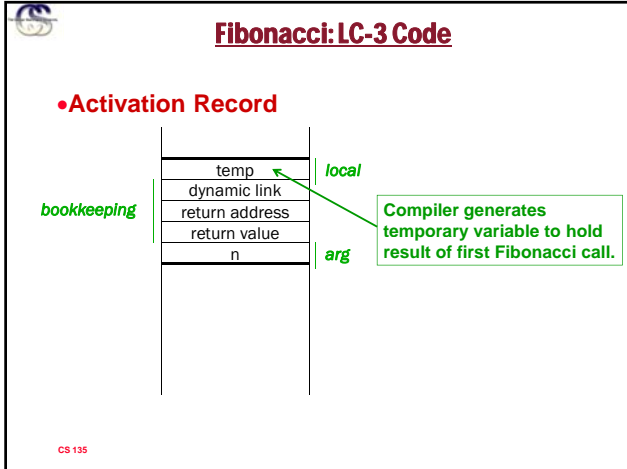
- Fibonacci(1) returns, Fibonacci(2) calls Fibonacci(0):** The stack contains 'main', 'Fib(3)', 'Fib(2)', and 'Fib(0)'. R6 points to the top of the 'Fib(0)' record.
- Fibonacci(2) returns, Fibonacci(3) calls Fibonacci(1):** The stack contains 'main', 'Fib(3)', and 'Fib(1)'. R6 points to the top of the 'Fib(1)' record.
- Fibonacci(3) returns:** The stack contains only 'main'. R6 points to the top of the 'main' record.

CS 135

### Tracing the Function Calls

- If we are debugging this program, we might want to trace all the calls of Fibonacci.
  - Note: A trace will also contain the arguments passed into the function.
- For Fibonacci(3), a trace looks like:
  - Fibonacci(3)
  - Fibonacci(2)
  - Fibonacci(1)
  - Fibonacci(0)
  - Fibonacci(1)
- What would trace of Fibonacci(4) look like?

CS 135



### LC-3 Code (part 1 of 3)

```

•Fibonacci ADD R6, R6, #-2 ; skip ret val, push ret addr
 STR R7, R6, #0
 ADD R6, R6, #-1 ; push dynamic link
 STR R5, R6, #0
 ADD R5, R6, #-1 ; set frame pointer
 ADD R6, R6, #-2 ; space for locals and temps

 LDR R0, R5, #4 ; load n
 BRz FIB_BASE ; check for terminal cases
 ADD R0, R0, #-1
 BRz FIB_BASE

```

CS 135

### LC-3 Code (part 2 of 3)

```

•
 LDR R0, R5, #4 ; read parameter n
 ADD R0, R0, #-1 ; calculate n-1
 ADD R6, R6, #-1 ; push n-1
 STR R0, R6, #0
 JSR Fibonacci ; call self

•
 LDR R0, R6, #0 ; pop return value
 ADD R6, R6, #1
 STR R0, R5, #-1 ; store in temp
 LDR R0, R5, #4 ; read parameter n
 ADD R0, R0, #-2 ; calculate n-2
 ADD R6, R6, #-1 ; push n-2
 STR R0, R6, #0
 JSR Fibonacci ; call self

```

CS 135

### LC-3 Code (part 3 of 3)

```

•
 LDR R0, R6, #0 ; pop return value
 ADD R6, R6, #1
 LDR R1, R5, #-1 ; read temp
 ADD R0, R0, R1 ; Fibonacci(n-1) + Fibonacci(n-2)
 BRnzip FIB_END ; all done

FIB_BASE AND R0, R0, #0 ; base case – return 1
 ADD R0, R0, #1

FIB_END STR R0, R5, #3 ; write return value (R0)
 ADD R6, R5, #1 ; pop local variables
 LDR R5, R6, #0 ; pop dynamic link
 ADD R6, R6, #1
 LDR R7, R6, #0 ; pop return address
 ADD R6, R6, #1
 RET

```

CS 135



### **Function Calls -- Summary**

- **Activation records keep track of caller and callee variables**
  - > Stack structure
- **What happens if we “accidentally” overwrite the return address ?**

CS 135



### **Next...**

- **Pointers and Arrays**
- **Dynamic data structures**
  - > Allocating space during run-time
- **C to LC3 cross compiler**
- **Exam 2: Thursday Nov.18th**

CS 135