Compiling C to Assembly – the Run-time Stack

(Chapters 11-13)

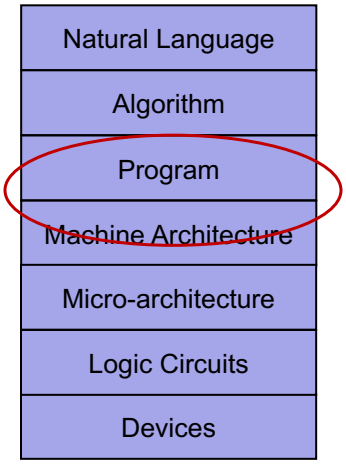
1

Problem Transformation - levels of abstraction

How do user programs get executed?

Translation from C to Assembly

- **Scope of variables**
- **Function calls**
- **Recursion**
- **Pointers & Arrays**
- **Data Structures**
- **Memory Allocation**



2

2

Programming Languages

•Assembly language is low-level.

- It exposes machine instructions and details of the ISA to the programmer.
- It is ISA-specific.
- It is useful when the programmer needs fine-grained control of instruction flow and memory usage.

•A high-level language provides a computational abstraction that is machine-independent.

- Symbolic names (variables) instead of registers and memory locations.
- High-level operators: multiply, divide, shift, ...

3

3

Why use a High-Level Language? …our choice= C

•Expressiveness: say more with less effort, closer to human-level thinking

```
a = b * c;
```

C statement

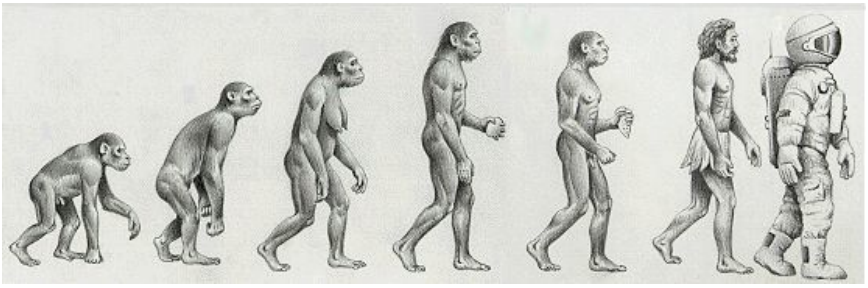
Equivalent LC-3 code

```
        AND R2,R2,#0
        AND R3,R3,#0
        LDR R0,R5,#-1  ; b
        BRz L3
        BRp L1
        NOT R3,R3
        NOT R0,R0
        ADD R0,R0,#1
L1      LDR R1,R5,#-2  ; c
        BRz L5
        BRp L2
        NOT R3,R3
        NOT R1,R1
        ADD R1,R1,#1
L2      ADD R2,R2,R0   ; b * c
        ADD R1,R1,#-1
        BRp L2
        ADD R3,R3,#0
        BRp L3
        NOT R2,R2
        ADD R2,R2,#1
L3      STR R2,R5,#0   ; store to a
```

•Both multiply two values together –
•which is easier to understand?

4

The Evolution of Programming



Machine Language (binary) | Assembly Language | C | Java | Python | Haskell | What's Next??

Simula/C++/etc

5

Quick Note: Compilation vs. Interpretation

•Different ways of translating high-level language

•*Interpretation (LISP, Java, Python, Matlab…)*

- interpreter = program that executes program statements (generally one line/command at a time)
  - Called a Virtual Machine
- easy to debug, make changes, view intermediate results

•*Compilation (C, C++, Pascal,…)*

- translates statements into machine language
  - does not execute, but creates executable program
- performs optimization over multiple statements

Example:

X = W+W

Y = X + X

Z= Y + Y

Interpretted language: 3 instructions

Compiler optimized code: 1 instruction Z = 8*W

6

6

Compiling a C Program

•Entire mechanism is usually called the “compiler”

•Preprocessor

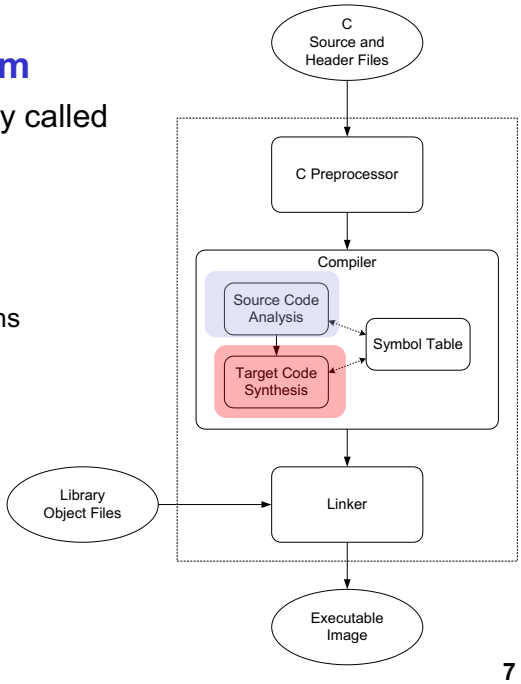
- macro substitution
- conditional compilation
- “source-level” transformations
  - output is still C

•Compiler

- generates object file
  - machine instructions

•Linker

- combine object files (including libraries) into executable image



7

7

Linking, Loading,…Makefiles

•Linking: produce final executable program by combining other code (libraries) used by program

- Dynamic Linking: instead of linking at compile time the linking is done at run-time

• Multi-file development….Makefiles assist in this

- Read tutorials on makefiles

• Loader: load executable image generated by linker & execute

- Part of Operating system
- Memory management system does the actual mapping from user space to physical addresses
  - .ORIG x3000 refers to user space x3000

8

8

Compiler

- Source Code Analysis….you will learn the theory/foundation behind this in Foundations of Computing course
  - "front end": parses programs to identify its pieces
    - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)
- Code Generation… we will cover it implicitly in this course
  - "back end": generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine
  - *We will play the role of the code generation component as we discuss how different C concepts are implemented in LC3*
    - This is what the compiler backend does
- Symbol Table
  - map between symbolic names and items
  - like assembler, but more kinds of information

9

9

A Simple C Program

```
#include <stdio.h>
#define STOP 0
int scale=10;
int square(){     /* function square */{
      int x,y; /* two local variables */
      x = scale*scale;
      return(x);}
/* Function: main  */
/* Description: gets value and squares it */
main()
{  /* variable declarations */
    int number, value, temp; /* three integer local
variables */
    /* prompt user for input */
    printf("Enter a positive number: ");
    scanf("%d", &value);  /* read into value */
    /* scale input and print */
    number = value*square();
    printf("number is %d\n", number);
}
```

10

10

Symbol Table

•Like assembler, compiler needs to know information associated with identifiers

- in assembler, all identifiers were labels and information is address
- Symbol table kept track of the addresses of the labels

•Compiler keeps more information

- Name (identifier)
- Type
- Location in memory
- Scope

Name	Type	Offset	Scope
scale	int	0	global
number	int	0	main
value	int	-1	main
temp	int	-2	main
x	int	0	square
y	int	-1	square

11

11

Implementing C: Translation from C to Assembly

•Translate C to assembly

- Translating operations is the easy part
- Y = A*B is: MULT R3, R1, R2

•Implement concept of scope

•Concept of storage class

- Static, Auto, Register….

•Pointers

•Function calls

- Passing arguments to functions

•…HOW ?

•Starting point (Today): How are C variables allocated to memory ?....*Run-time Stack*

12

12

Translation: Allocation of Variables

•What is compiler's task?

1. Allocate memory for variables in a systematic way.
   Will build a **symbol table** to keep track of variable type, size, location.
2. Generate instruction sequences that carry out the computations specified by operators and statements.

•**For this class, we will compile "by hand" to get a sense of how these translations occur.**

13

13

Start point for translation: 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

•So how do we make this happen in assembly ???

14

14

Lessons from Example 1 (inclassC-1.c)

- 1,2,3. **Concept of scope**
  - Global and local
    - Code block (enclosed in { }) defines a scope block (Useful for placing debugging statements)
  - Prints:
    - Question1: Global= 0 Local = 1 x= 4
    - Question2: Global= 4 Local = 2 x=10
    - Question 3: Global= 4 Local = 1 x=10
- 4. Compound conditional statements
  - False – odd behavior….because C does not support this
  - Evaluate (7 > x)=? first and then ? > 2 (boolean with constant)
- 5. for loop iteration variable
  - initialized to 4 so for loop is not executed – Question5 never printed
- 6. False: Post and pre increment operators
  - Pre x++: Access value and then increment
    - Therefore y = x = 4 and x= 4+1=5.
  - Post ++x: Increment and then read value
    - Therefore x= 5+1 and z= 6

15

15

Lessons from Example 1

- 7. Constants – computed at start and stays as a constant
  - CC=8 and x=7
- 8. concept of local variables and passing arguments by value to function
  - 8a: x=7,y=0, i= something; 8b: x=7, y=30
  - Argument x passed to function is not changed in main
  - Local variable y in function foo1 different from y in main; y in main is return value
  - Local variable i not initialized, so prints some random number
- 9. foo2 first call prints x=10, y=20 (function then adds 30 to y)
- 10 foo2 second time prints x=10, y=50
  - Concept of static variable but local in scope – value of static persists
- 11. swap does not swap arguments in main!
  - Only swaps values in local variables
  - When functions are called, the values of the arguments are passed to the function and not 'access' to the variables/arguments themselves.
- 12. loop terminates...overflow leads to negative number...T_min

16

16

Defining a variable in C

- Identifier: references to this translate to locations
  - Name of the variable
  - Example: itslocal
- Type: gives us information on data representation and space needed
  - Type of variable such as int, float, char…
  - Example: int itslocal
- Scope
  - Where can it be accessed
  - Example: global variable itsglobal
- Storage Class
  - How does C compiler allocate the storage
    - Does value persist or not
  - Two main classes in C: *Automatic* and *Static*

17

17

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
  - **Symbol Table constructs this information**
- 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

18

18

Where can you put variables….?

- 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
    - persistance
- Global variable visible to all functions within a file
- Global variable visible in more than one file
- Block scope – inside the block (defined by { })

19

19

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

20

20

Scope vs. Lifetime

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

21

Scope vs. Lifetime

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

Can be overridden with "`static`"

22

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 *auto storage class* (and placed 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*

23

Storage Class: Automatic Variables

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

24

Static Variables (II)

- **`static`**
  - `Initialized to zero (like global)`
- Located in static area
- Value persists !
  - Example: foo2 being called second time
- Storage Class: STATIC

25

25

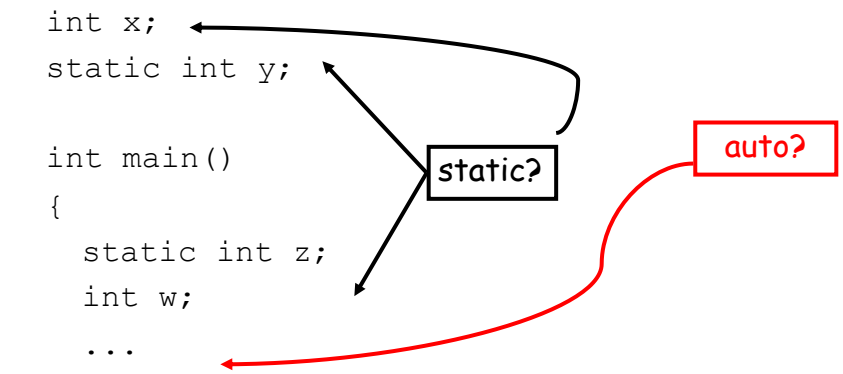
Static Variables (III)

- 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

26

26

Confused?



27

27

auto

```
int foo(int z)
{
  int x;
  ...
  if(x == z)
  {

     int y;
     y = ...
```

auto variables (implicitly)

28

28

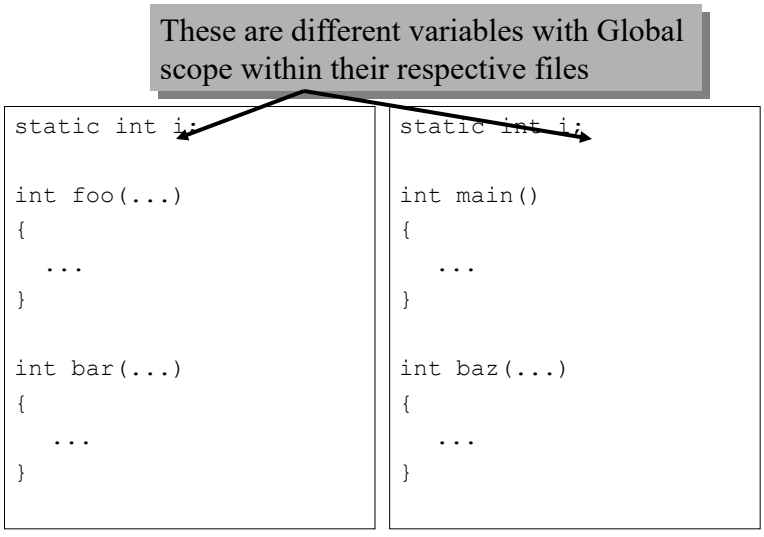
Static Initialization

```
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?

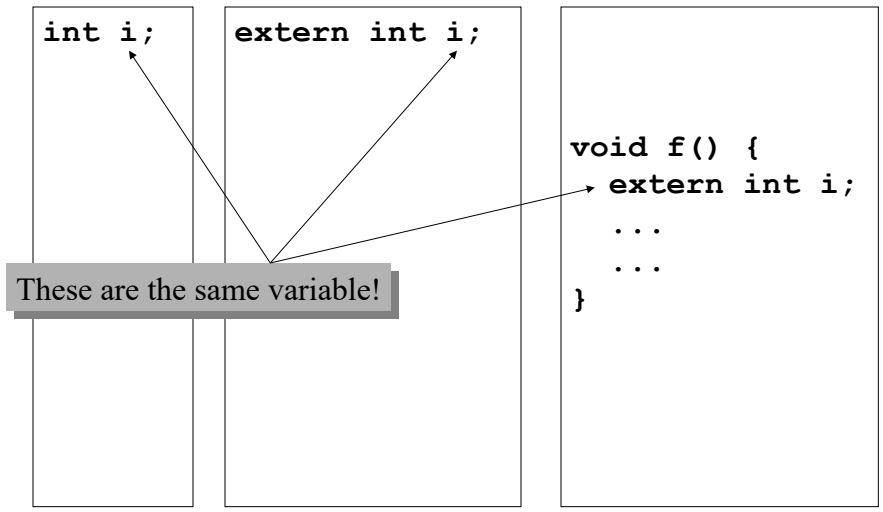
29

Scope can be global within a file



30

Scope can be global across all files



31

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
- …
  - Read on your own for now.

32

32

Storage Class Specifiers

register

auto

static

extern

const

volatile

Type Qualifiers

33

33

Allocation of Variables in Memory and enforcing Scoping rules

- Simply assigning a memory location for each variable may not be enough to enforce scope

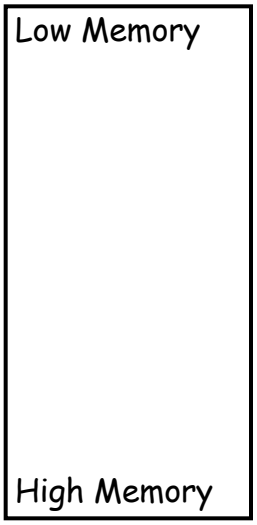
- Need to look at a better scheme to allocate high level program variables to memory in the processor
  - Scope
  - Storage class
  - Allocate space to a variable

34

34

Mapping C variables to Memory: Allocating variables in Memory

- How to allocate memory locations to variables
  - *Enforce scope*



35

35

Compiling & Executing C programs: The Run-time Stack

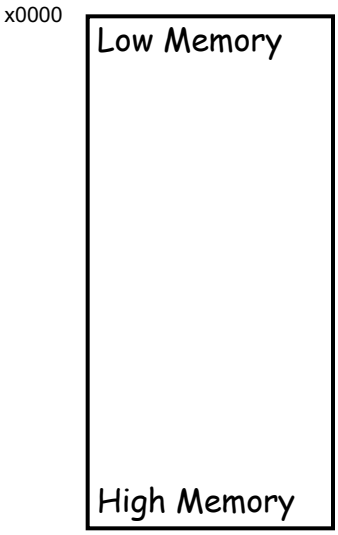
- **THIS** is the key….

36

36

Memory Representation

- Our convention will be that "high-memory" will be on the bottom and "low-memory" on top.
  - drawings are not to scale

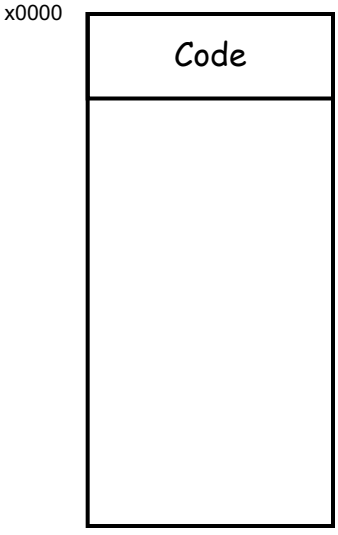


37

37

Typical Arrangement

- Normally the actual program code (executable instructions) is placed in low memory
  - Operating System and boot code usually in lowest mem area

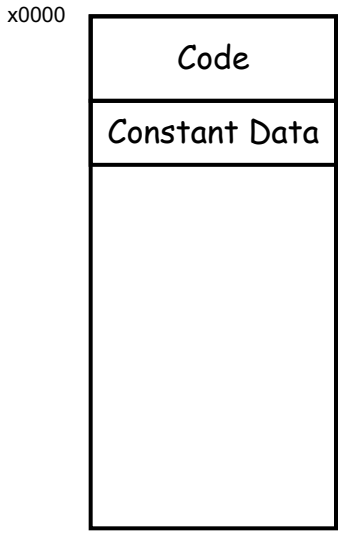


38

38

Typical Arrangement

- Next we have an area for storage of constant data

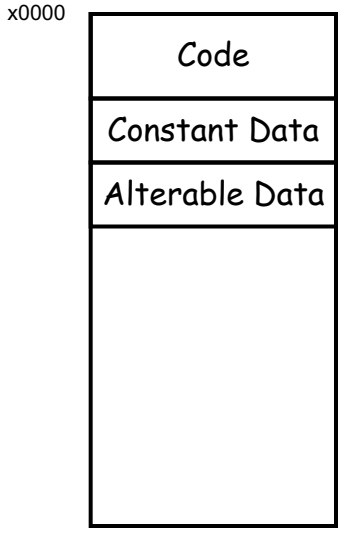


39

39

Typical Arrangement

- Data that may be changed follows

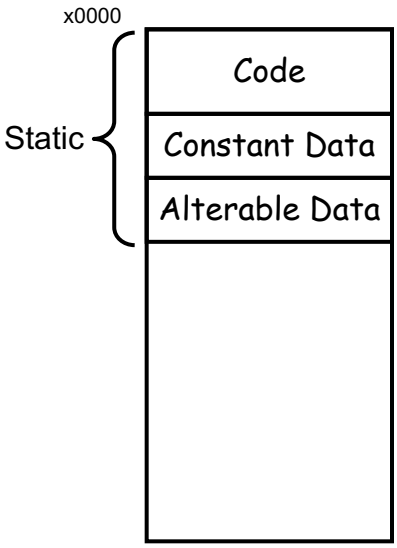


40

40

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.

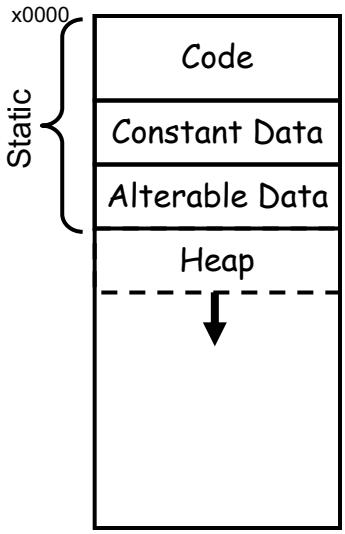


41

41

Typical Arrangement: Heap

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

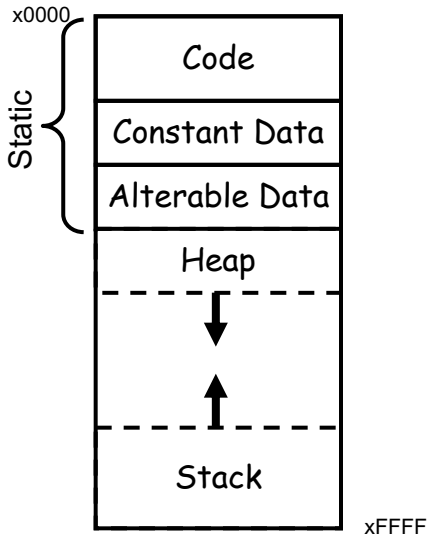


42

42

Typical Arrangement: stack for local variables

- Finally, the activation 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

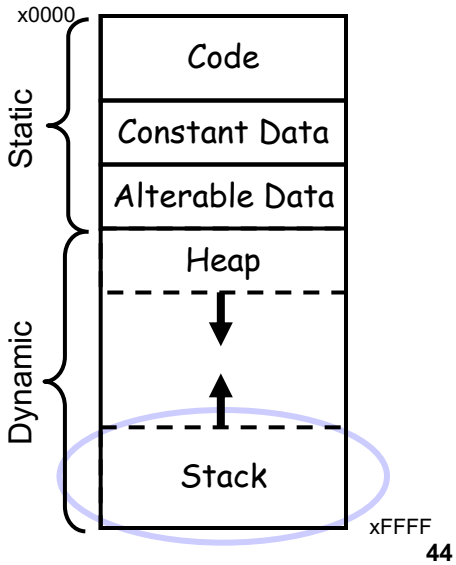


43

43

auto variables

- auto, short for automatic variables are those that exist on the stack. The auto keyword is not normally used.
- Automatic means that space is allocated and deallocated on the stack automatically without the programmer having to do any special operations.



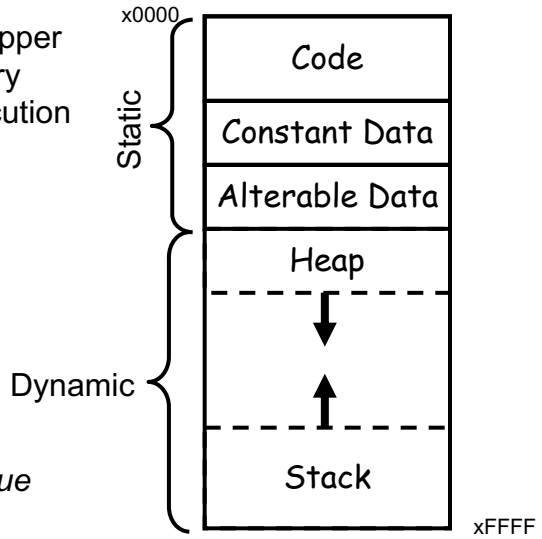
44

44

Summary of Typical Arrangement

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

*Not just their value



45

45

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?
  - Keep track of scope
  - Storage class
- Symbol table provides much of this information

46

46

Activation Records!!!

- Two main areas (for now) in memory:
  - Global data section
  - Run-time stack
- Local variables exist only during lifetime of function
  - De-allocated after function completes

- How to define area of memory for a code block/function ?
- .....Activation Record
  - Also called Stack Frame
  - Local variables allocated in the activation record
  - Activation record is portion of run-time stack
    - Function can only access a valid portion of the stack
    - Should not access another functions activation records!
  - When function returns…POP the record
    - The local variables can no longer be accessed!

47

47

Symbol Table

- For each variable keeps track of
  - Type
  - Scope
  - Location (as an offset)
    - Either in global area or local area
    - If in local area, then based on activation record
  - Other info (const, etc.)

48

48

Symbol Table

•Like assembler, compiler needs to know information associated with identifiers

- in assembler, all identifiers were labels and information is address
- Symbol table kept track of the addresses of the labels

•Compiler keeps more information

- Name (identifier)
- Type
- Location in memory
- Scope

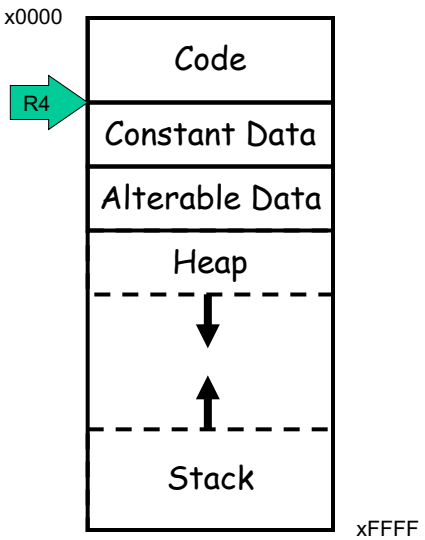
Name	Type	Offset	Scope
scale	int	0	global
number	int	0	main
value	int	-1	main
temp	int	-2	main
x	int	0	square
y	int	-1	square

49

49

Offset?

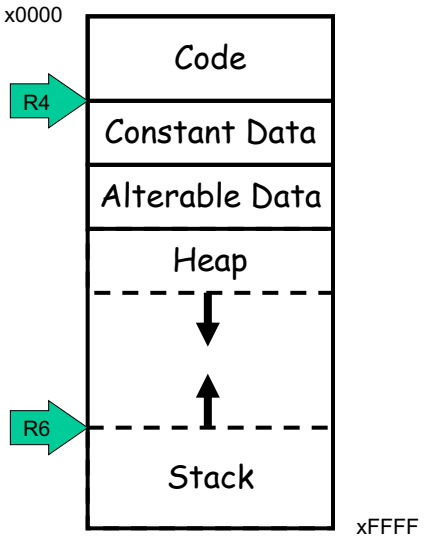
- Assembly code written by a compiler usually looks a little different from assembly code written by hand
- Registers are dedicated to point to key areas of memory
- R4 is the Global Pointer
  - Points to start of global static area



50

Keeping Track of `auto` variables

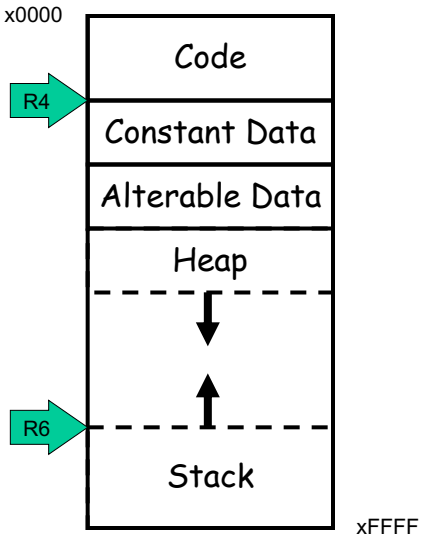
- Stack pointer is obvious but the compiler writer needs more info...
- Where is the activation stack frame?
- R6 is Top of Stack (TOS) pointer



51

Why do we care?

- We would like to know where a variable is throughout the execution of a function
- But, wait you say, I can just reference the variable from the stack pointer
- HA!



52

Can we use TOS

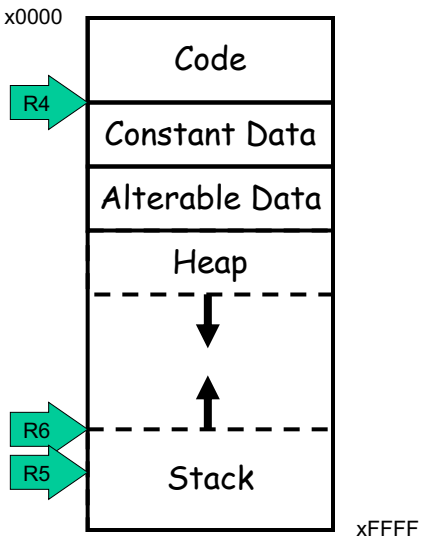
```
int f(int a, int b) {
   int c;
   c = a + b;
   return c;
}

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?**

53

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.
- Note: Frame pointer also called dynamic link
- Store Frame pointer in register….R5
  - Points to 'start' of set of local variables



54

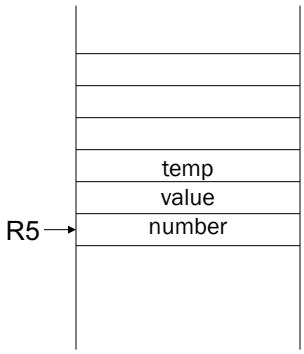
LC3: Local Variable Storage

•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 fram*e.

- 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.



55

55

Summary: LC3 Allocation for Variables

•Global data section

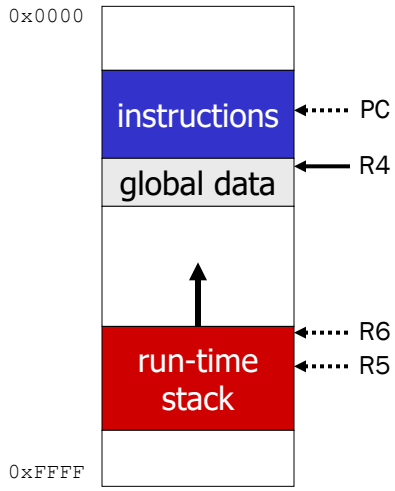
- All global variables stored here (actually all static variables)
- R4 points to beginning

•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`

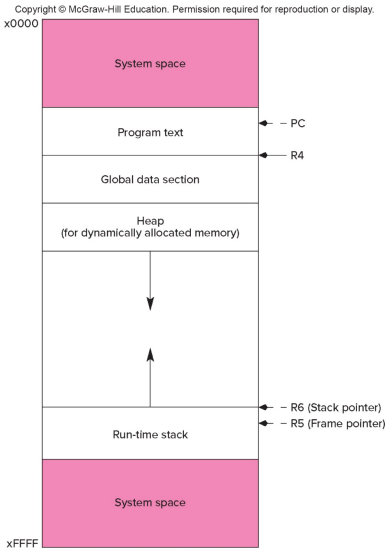


56

56

LC-3 Memory Map – The Complete Picture

- The LC-3 operating system reserves some of the memory address space for trap vectors, service routine code, and memory-mapped I/O.
- Program instructions will be placed in the "program text" section.
- Global variables are allocated next. R4 is set to the first allocated address for globals.
- Local variables are stored on the run-time stack. R5 points to the local variables of the currently-executing function.



57

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?

58

58

Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;   /* local to main */
  int LocalA, LocalB;

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

  /* perform calculations */
  LocalA = inLocal++;
  LocalB = inLocal & ~inGlobal ;

  /* print results */
  printf("The results are: LocalA = %d, LocalB =
%d\n",
         LocalA, LocalB);
}
```

59

59

Example: Symbol Table

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
LocalA	int	-1	main
LocalB	int	-2	main

60

60

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)
```

61

61

Example (continued)

```
; first statement:
; outLocalA = inLocal++;
;address of inLocal = R5 + #0
Address of localA = R5 + # -1
      LDR R0, R5, #0   ; get inLocal
      ADD R1, R0, #1   ; increment
      STR R1, R5, # -1   ; store localA
```

62

62

Example (continued)

```
•; second statement:
•; LocalB = inLocal & ~inGlobal;
• address of inGlobal= R4+ #0
        ; previous code segment left
        ; inLocal value in R0
       LDR R1, R4, #0  ; get inGlobal
       NOT R1, R1      ; Not inGlobal
       AND R2, R0, R1  ; inLocal & ~inGlobal
       STR R2, R5, #-2 ; store in LocalB
                       ; (offset = -2)
```

63

63

Example: C to LC3 Translation

•What is the C code corresponding to these LC3 code segments

Y= A+X;

•First identify accesses/addresses for variables A, X, Y:

- R4, #0 R5, #0 R5, # -1

•Symbol Table:

Identifier	Type	Offset	Scope
A	int	0	Global
B	int	2	Global
X	int	0	main
Y	int	-1	main
Z	int	-2	main

64

64