Section 3.2
Dynamic Memory Allocation

1. Description
2. Heap vs. function call stack
3. Double pointers

3.2.1 Description

- What is memory allocation?
  - compiler translates a C program (source code) to machine readable equivalent (object code)
  - compiler "allocates" (i.e. reserves) bytes in memory for each program variable
- Problems:
  - once allocated, memory **cannot** be resized !
  - we may not know at compile time how much memory we will need for some variables
- Two types of memory allocation
  - static = at compile time
  - dynamic = at runtime

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Static Memory Allocation

- If we know how much memory we need, we can just declare variables of the proper data type
- At compile time, the compiler will reserve the corresponding amount of memory
- Examples:

```
char ca[3], *cp;
int  i;
float f;
struct personName myName;
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Dynamic Memory Allocation

- If we don't know how much memory we need, we can allocate the memory at *runtime*
- We use the **`malloc()`** or **`calloc()`** functions to do this
  - **`malloc()`** returns a pointer to **`void`**
  - **`void`** is an empty data type
  - pointers to **`void`** are usually *typecast* to another data type
- What is typecasting?
  - changing from one data type to another
- Limitations of typecasting
  - can we typecast *any* data type to any other?
    - think about the size issues
  - we typecast pointers **only**

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

`malloc()` Function

- Function prototype:

```
void *malloc(size_t size)
```

- Description:
  - reserves in memory the number of bytes specified in *size*
  - returns the start address of the new block of reserved memory
  - do **not** lose this!

- Important points:
  - once allocated, memory is reserved until:
    - it is freed explicitly, using `free()` function
    - the program terminates
  - if you lose (i.e. clobber or move out of scope) a pointer to dynamically allocated memory, it stays reserved anyway
    - this is a **memory leak**

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Examples

- Example:

```
double *a;
a = (double *) malloc(sizeof(double));
*a = 3.14;
printf("%lf\n", *a);
free(a);
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

- Example of memory leak:

```
double *a;
a = (double *) malloc(sizeof(double));
a = (double *) malloc(sizeof(double));
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

`calloc()` Function

- Function prototype:

```
void *calloc(size_t nitems, size_t size)
```

- Description:
  - reserves in memory *nitems* number of items, each the number of bytes specified in *size*
  - returns the start address of the new block of reserved memory

- Example:

```
double *a;
a = (double *) calloc(70, sizeof(double));
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Accessing Dynamically Allocated Memory

- Dynamically allocated memory can be accessed:
  - through pointers
  - using array notation
- Example:

```
int *a;
a = (int *) calloc(5, sizeof(int));
a[0] = 8;
*(a+2) = 3;
a[3] = 9;
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Deallocating Dynamically Allocated Memory

- You must *deallocate* (release) all memory that you allocate
- Why?
  - you can't reuse memory that's allocated
  - no one else will clean up after you!
- Function prototype:

```
void free(void *ptr)
```

- Description:
  - releases the memory pointed to by **`ptr`**

© 2010 Christine Laurendeau

Memory Leaks

- What's a memory leak?
  - when you lose a pointer to dynamically allocated memory
  - when you forget to deallocate dynamically allocated memory
- What's so bad about this?
  - you lose access to the data in that memory
  - amount of memory available is not infinite
    - you are giving up potentially available memory for the rest of the program's lifetime
- Some languages perform *garbage collection*
  - mechanism that automatically frees all unreferenced memory

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

3.2.2 Heap vs. Function Call Stack

- On program startup, O/S allocates 4 different areas of memory:
  - storage for program instructions
  - global memory area
  - function call stack
  - heap
- Global memory:
  - global variables
  - static variables

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Function Call Stack

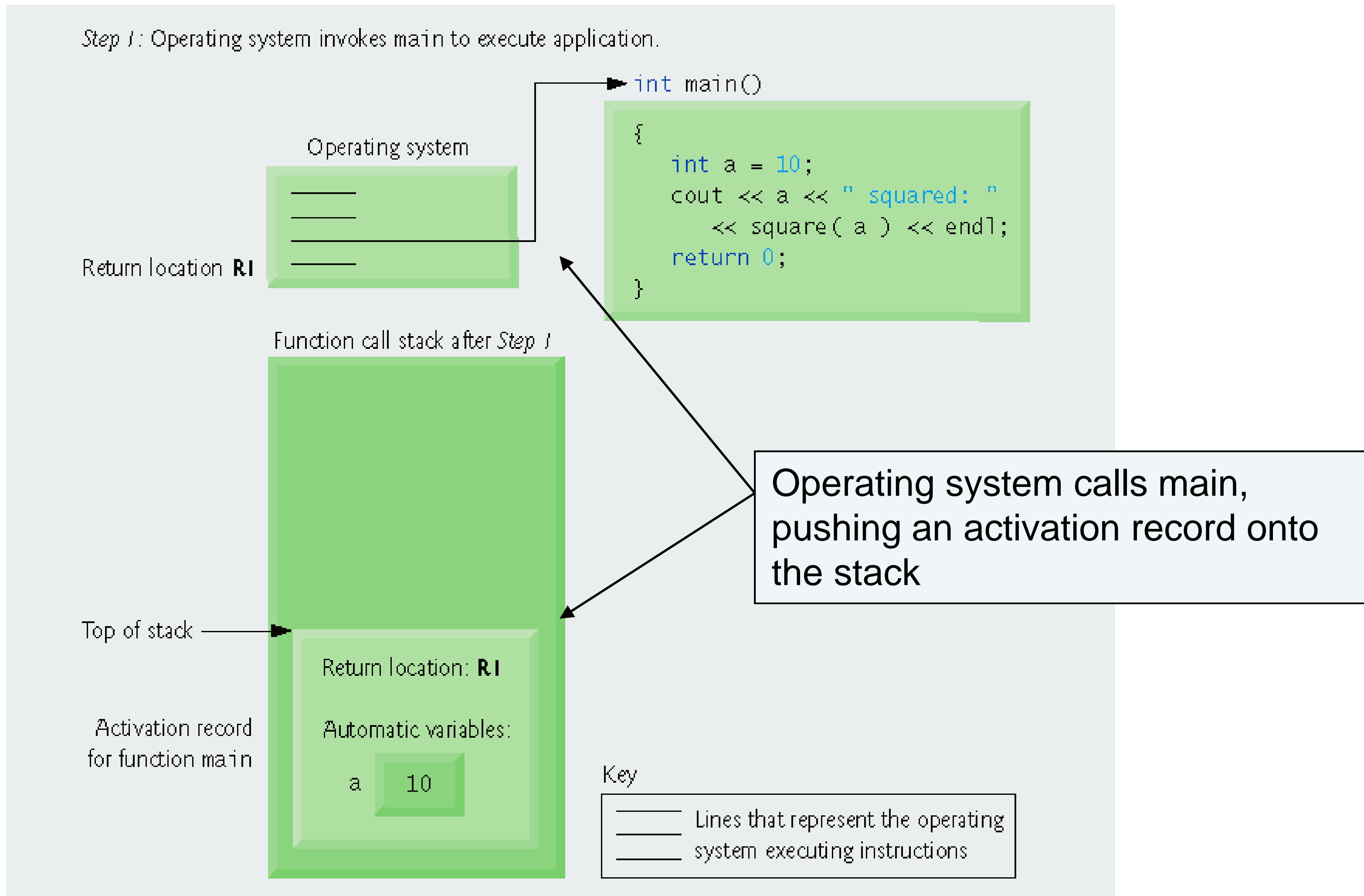
- Data structure: collection of related data items
- Stack data structure
  - Analogous to a pile of dishes
  - A last-in, first-out (LIFO) data structure
    - The last item pushed (inserted) on the stack is the first item popped (removed) from the stack
- Function call stack contains:
  - local variables
  - function parameters

© 2008 Pearson Education, Inc. All rights reserved.

Function Call Stack (Cont.)

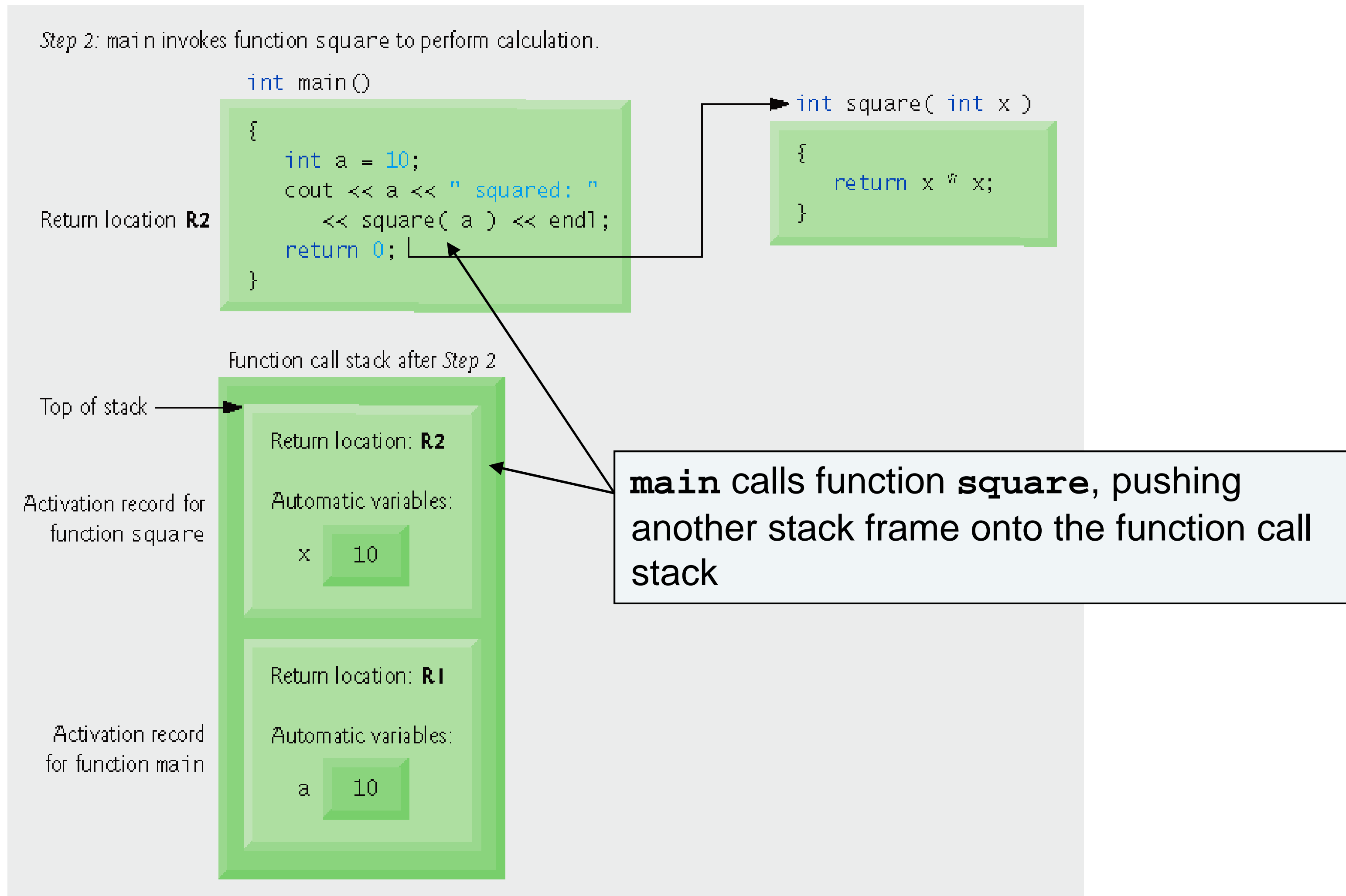
- Sometimes called the program execution stack
- Supports the function call/return mechanism
  - Each time a function calls another function, a stack frame (also known as an activation record) is *pushed* onto the stack
  - Activation record contains:
    - the return address that the called function needs to return to the calling function
    - automatic variables—parameters and any local variables the function declares
  - When the called function returns
    - Stack frame for the function call is *popped*
    - Control transfers to the return address in the popped stack frame

© 2008 Pearson Education, Inc. All rights reserved.



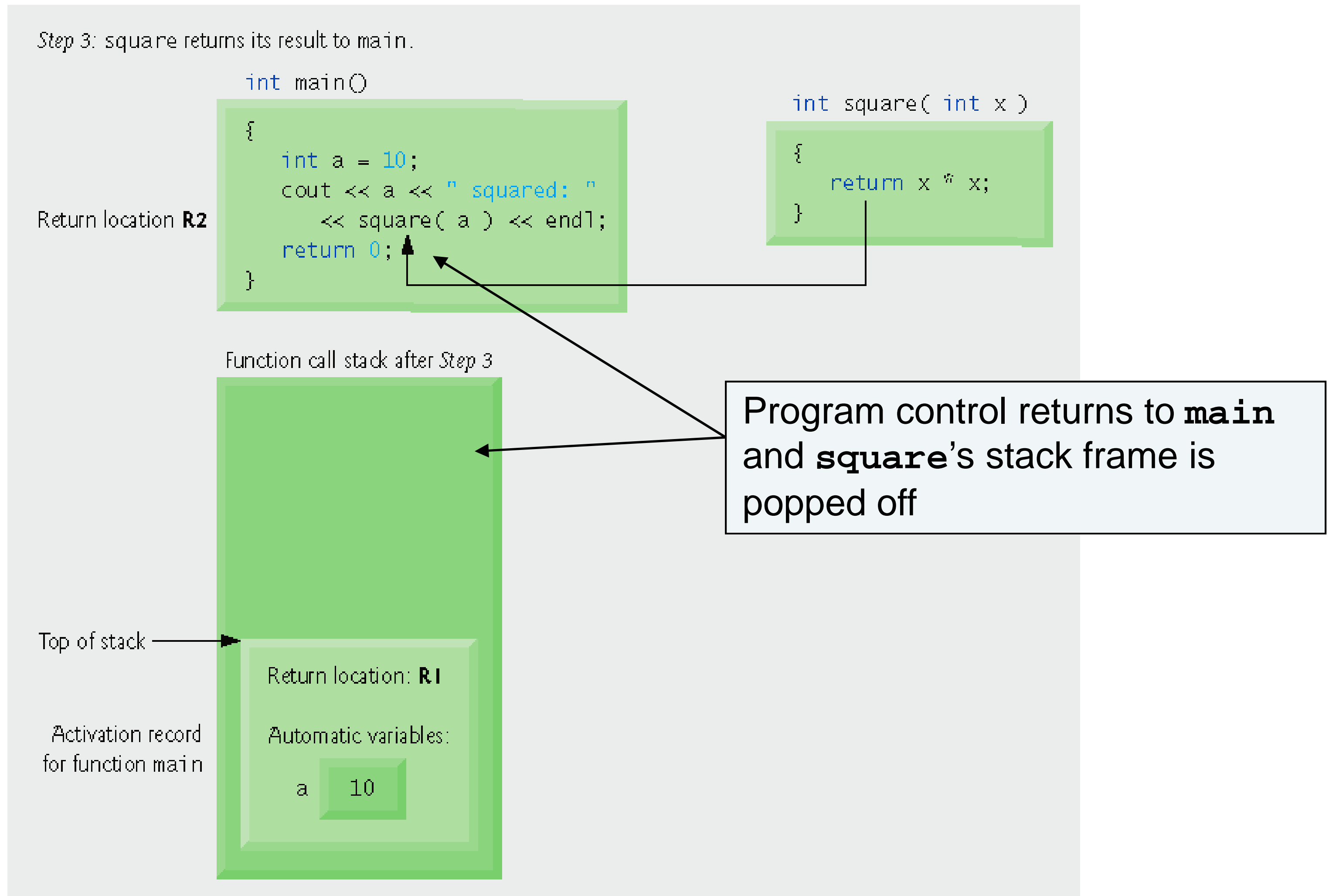
Function call stack after the operating system invokes main to execute the application

© 2008 Pearson Education, Inc. All rights reserved.



Function call stack after main invokes function square to perform the calculation

© 2008 Pearson Education, Inc. All rights reserved.



Function call stack after function square returns to main

© 2008 Pearson Education, Inc. All rights reserved.

Heap

- Heap contains:
  - chunk of memory for dynamic allocation
- Memory leaks:
  - dynamically allocated memory must be deallocated **explicitly**
    - neither O/S nor compiler will do it for you
    - use **`free()`** function
  - don't overwrite pointers into heap
  - if pointers into heap are stored in function call stack, deallocate before pointer moves out of scope
- If you run out of heap space:
  - **`malloc()`** returns null pointer (zero)
  - if not handled, program terminates abnormally (i.e. crashes)

© 2008 Pearson Education, Inc. All rights reserved.

3.2.3 Double Pointers

- Double pointer: pointer to a pointer
- Double pointers can be used:
  - through pointers
  - using array notation
- Example of declaration:
  - the declaration **`double **ptr;`** means that **`ptr`** is a pointer variable referencing a pointer variable referencing a double
  - **`ptr`** takes up 4 bytes, it points to another variable that is also 4 bytes, which points to a double that is 8 bytes

© 2010 Christine Laurendeau, based on Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Double Pointers (cont.)

- Example of accessing double pointer with array notation:

```
double **m;
m = (double **) calloc(2, sizeof(double *));
m[0] = (double *) calloc(3, sizeof(double));
m[1] = (double *) calloc(2, sizeof(double));
m[0][1] = 6.3;
m[1][0] = 2.8;
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010

Double Pointers (cont.)

- One possible use:
  - when a function modifies a pointer
  - to pass a pointer by reference, it must be a double pointer

```
int integers(int listsize, int **list)
{
  int    i;
  *list=(int *) malloc(listsize*sizeof(int));
  if (*list == NULL)
    return(0);
  for (i=0; i<listsize; i++)
    (*list)[i]=10+i;  /* mixed array/pointer reference - what is pure ptr? */
  return(1);
}

int main(int argc, char *argv[])
{
  int    *numbers;
  int    i;

  i=integers(7, &numbers);
  for (i=0; i<7; i++)
    printf("%d\n", numbers[i]);
}
```

© Hoover, System Programming Using C and Unix, Addison-Wesley, 2010