CS 240: Programming in C

Lecture 16: Pointers to Functions Recursion

Prof. Jeff Turkstra

Announcements

- Remember, no feasting with faculty Thursday
  - Will resume week after spring break
- Staff and I will be offline over break for the most part
  - Don’t expect responses after Friday until classes resume
  - Complete the homework before break to avoid scrambling when you get back!

Midterm 1 Stats

- Mean: 68.00
- Median: 71.00
- Standard Deviation: 17.37
- Maximum: 97.50
- Minimum: 5.50

- Regrade requests are open through next Wednesday
  - Do not expect responses after Friday
    - TAs will respond Monday when classes resume

Reading

- Read about recursion
  - Section 4.10 in K&R

Strings

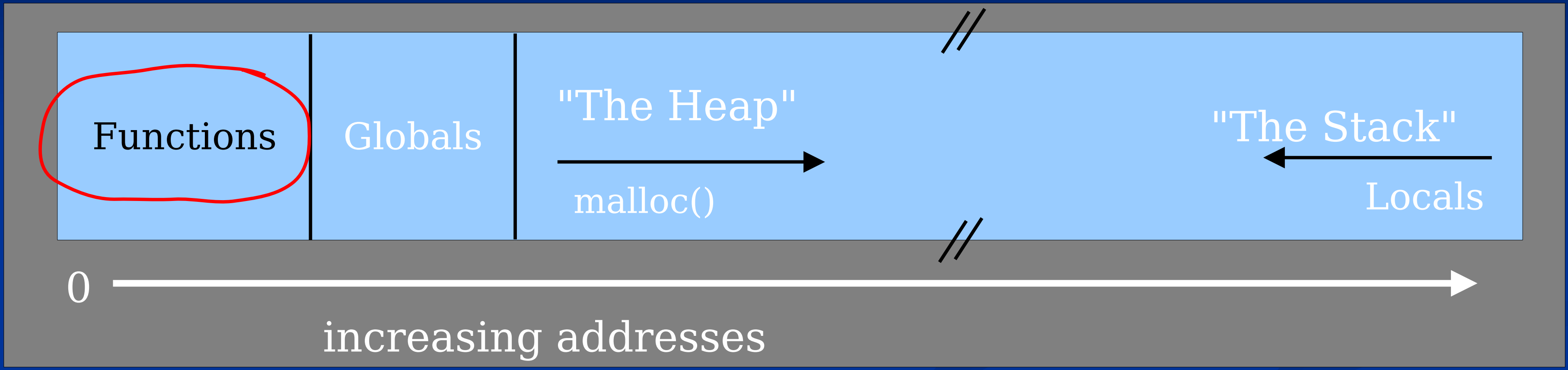
- When you use a string literal in C, that value is stored in “read-only” memory (the text/code segment)
  - It cannot be changed
- What’s the difference between this:
  `char *str = “Hello!”;`
  …and this:
  `char str[] = “Hello”;`

Strings

- `char *str = “Hello!”;`
  - Allocates a pointer on the stack
  - Points to an array in the read-only “code/text segment”
- `char str[] = “Hello”;`
  - Allocates an array on the stack and initializes it by copying values from the array in the read-only “code/text segment”

Function pointers

- Recall the dreaded memory layout map...



- Functions reside in memory. Therefore we can refer to their addresses
- We can call functions via their addresses!

Defining a function pointer

- The difficult part of using function pointers is figuring out how to declare a pointer to a function
- Here is a pointer to a function that accepts two integers and returns an integer:
  `int (*ptr_to_func)(int x, int y);`
- We could also initialize this pointer to NULL:
  `int (*ptr_to_func)(int x, int y) = NULL;`
- We don’t need argument names:
  `int (*ptr_to_func)(int, int) = NULL;`

Using a function pointer

```
int sum(int addend, int augend) {
  return addend + augend;
}

int main() {
  int result = 0;
  int (*ptr_to_func)(int, int) = NULL;

  ptr_to_func = sum;
  result = (*ptr_to_func)(3, 5);
  printf(“result = %d\n”, result);
  return 0;
}
```

Or like this…

```
int sum(int addend, int augend) {
  return addend + augend;
}

int main() {
  int result = 0;
  int (*ptr_to_func)(int, int) = NULL;

  ptr_to_func = sum;
  result = ptr_to_func(3, 5);
  printf(“result = %d\n”, result);
  return 0;
}
```

Passing a pointer to function

```
int do_operation(int (*pf)(int, int),
                 int value1,
                 int value2) {
  return pf(value1, value2);
}

int main() {
  int (*ptr_to_func)(int, int) = NULL;
  ptr_to_func = sum;
  printf(“%d\n”,
      do_operation(ptr_to_func, 3, 5));
  return 0;
}
```

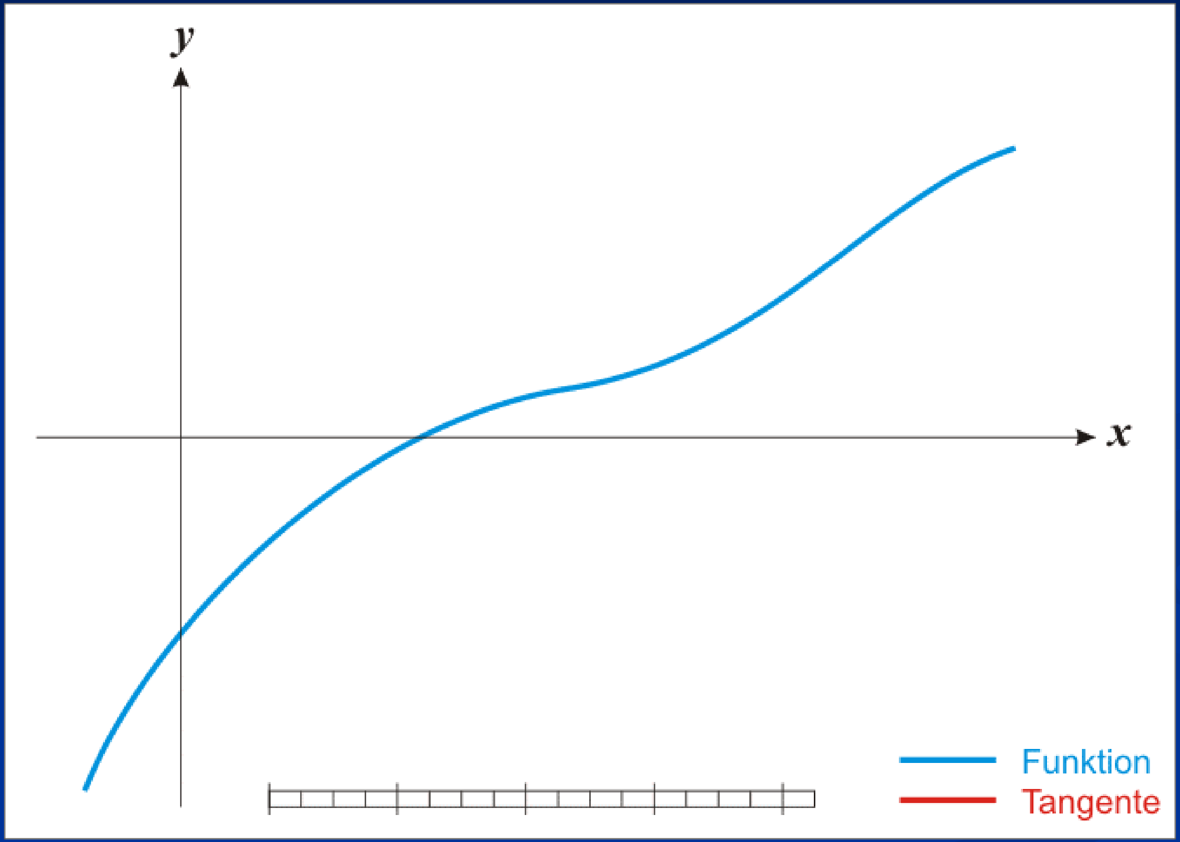
What’s this good for?

- Suppose we have a subroutine that uses Newton’s Method to locate a root of a polynomial function:

```
float newton(float (*ptr_fn)(float x),
             float start);
```

- We might want to call the subroutine for different mathematical functions…

```
root1 = newton(func1, 5.3);
root2 = newton(func2, 2.9);
```



y
x
Funktion
Tangente

Newton's Method

```
float newton(float (*function)(float),
             float start) {
  float x1, x2, y1, y2, tmp;
  x1 = start;
  x2 = x1 + 1.0;
  do {
    y1 = function(x1);
    y2 = function(x2);
    tmp = x1 – y1 / ((y1 – y2) / (x1 – x2));
    x2 = x1;
    x1 = tmp;
  } while (fabs(y1 – y2) > 0.001);
  return x1;
}
```

Example: find the square root of 23

```
/* The positive root of this function
 * is the square root of 23.
 */
float func(float x) {
  return pow(x, 2) – 23.0;
}

int main() {
  float root = 0;
  root = newton(func, 1);
  printf("root of x^2 – 23 = %f\n", root);
  return 0;
}
```

Pointers to functions and linked-lists

- Linked list manipulation routines we've looked at so far have assumed that one of the elements of the node was the key of the search/sort
- What if we had multiple items in the node structure and we wanted to be able to search by any one of them?

```
struct node {
  char *name;
  char *title;
  char *company;
  char *location;
  struct node *next;
};
```

Fields that we might search by…

New list_search

```
struct node *list_search(
  int (*compare)(struct node *, char *),
  struct node *head_ptr, char *item) {

  while (head_ptr != NULL) {
    if (compare(head_ptr, item) == 0) {
      return head_ptr;
    }
    head_ptr = head_ptr->next;
  }
  return NULL;
} /* list_search */
```

Example comparison function

```
/* Definition of comparison:
 * zero:     equal
 * negative: structure value ‘less than’ item
 * positive: structure value ‘greater than’ item
 */
int compare_name(struct node *ptr, char *item) {
  return strcmp(ptr->name, item);
}

/*
 * Example of calling list_search…
 */
ptr = list_search(compare_name, head_ptr, “Jeff”);
```

Purdue Trivia

- We are on the seventh iteration of the Boilermaker Special, Purdue's official mascot.
  - World's largest, fastest, heaviest, and loudest collegiate mascot
  - Dedicated 9/3/2011
- Boilermaker Xtra Special is a smaller version designed for use indoors
  - Eighth iteration
- Both are entrusted to the Purdue Reamer Club


7
PURDUE

Logical expressions revisited

- How are logical expressions truly evaluated?
- For example, what values of x exist such that the value of x would be displayed on the screen?
  - What does this mean?
    `if (x == 0) printf(“x = %d\n”, x);`
  - What about this?
    `if (x = 0) printf(“x = %d\n”, x);`
  - And this?
    `if (x) printf(“x = %d\n”, x);`

Logical expression defined

- The ‘if’ statement evaluates to TRUE if the expression has a non-zero value

Using logical operators

- Compound expressions also check whether the quantities are zero or non-zero. E.g.:
```
if (x && y)
  printf(“y = %d\n”, y);
```
- Really means…
  if (((x != 0) && (y != 0)) != 0)
    printf(“y = %d\n”, y);
- …and the result of && is either 1 or 0
- Use logical operators to make a yes/no decision

Obvious properties of global variables

- Global variables are accessible from any function
- Every function sees the same global variables
- If any two functions read the value of a global variable, both functions will get the same value
- When any function returns, all global variables remain the same…
- Boring stuff… Why am I talking about this?

Not-so-obvious properties of local variables

- Local variables are visible only within the function they're defined in
- If you define a variable x in func1() and func1() invokes func2(), x is not visible in func2()
- If you invoke a function, set a local variable, and then return: the local variable is gone
  - When you invoke the function again, what's the value of the variable?
- If a function invokes itself, it gets a new copy of all of its local variables

A function invoking itself

```
void countdown(int n) {
  if (n >= 0) {
    printf(“%d...\n”, n);
    countdown(n-1);
  }
  return;
}

int main() {
  countdown(10);
  return 0;
}
```

Recursion

- When you write a function that invokes itself, the practice is called recursion. (The function recurs)
- For many computations, there is a way to write it recursively and a way to write it iteratively
  - The iterative version is often more efficient
  - The recursive way is often more convenient
- How does this work?

Representation of countdown

```
countdown(n=2):

If n >= 0
 print n
 countdown(n-1)
return
```

```
countdown(n=1):

If n >= 0
 print n
 countdown(n-1)
return
```

```
countdown(n=0):

If n >= 0
 print n
 countdown(n-1)
return
```

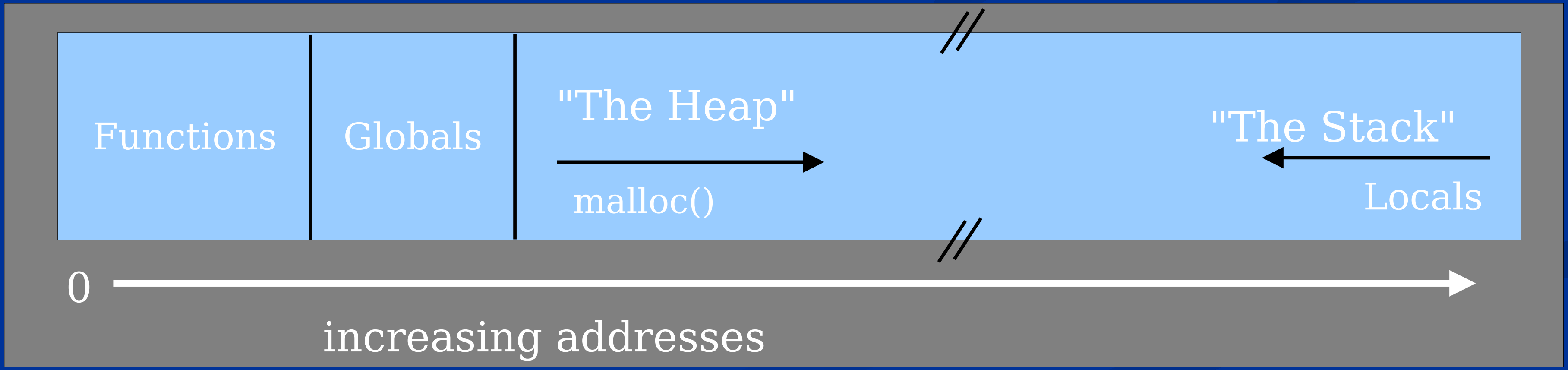
```
countdown(n=-1):

return
```

2...
1...
0...

The stack

- Memory layout again
- Every time you invoke a function, the invocation of the function takes some space on the stack for parameters, local variables and an indication about where to return to...
- When the function returns, the stack shrinks



Never return pointers to local variables

- When a function returns, its reservation of the stack for local variables goes away.
  - But, they won't be overwritten until another function is invoked
- Bad example:

```
char *create_string() {
  char str[100];
  strcpy(str, “Hello, world\n”);
  return str;
}
```

Returning pointer to something on the stack!!

Other recursion examples

```
void countup(int n) {
  if (n >= 0) {
    countup(n-1);
    printf(“%d...\n”, n);
  }
  return;
}

int main() {
  countup(10);
  return 0;
}
```

Factorial

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  return n * factorial(n – 1);
}
```

Fibonacci sequence

```
/*
 * Compute one number in the
 * Fibonacci sequence:
 * 1 1 2 3 5 8 13 21 34 55 89 144…
 */
int fibonacci(int n) {
  if (n == 0)
    return 1;
  if (n == 1)
    return 1;

  return (fibonacci(n – 1) +
          fibonacci(n – 2));
}
```

When using recursion…

- You always need to tell the function when to stop invoking itself
- Don’t return a pointer to something on the stack
  - I.e. don’t return an address of a local variable
- Don’t recurse too deeply…
  - You will run out of stack space

For next lecture

- Study the examples in this lecture at home
- Still struggling with pointers?
  - Chapter 5 in K&R (5.4, 5.6, and 5.11)
  - Sections 12.14 and 12.15 in Beej’s
- Read about recursion
  - Section 4.10 in K&R

Boiler Up!