CS 240: Programming in C

Lecture 16: Function Pointers, Recursion

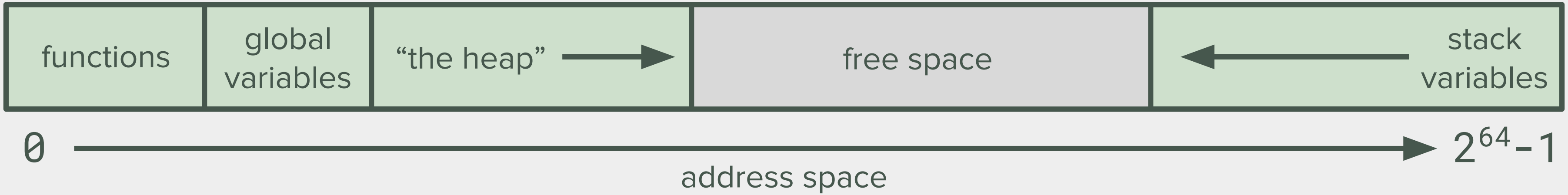
PURDUE UNIVERSITY

Announcements

- Midterm 2 next Tuesday!
  - Practice midterm and questions on the website
  - Look for the seating chart in a few days
- Cancelled lecture moved to 11/25
  - Enjoy your extended Thanksgiving break

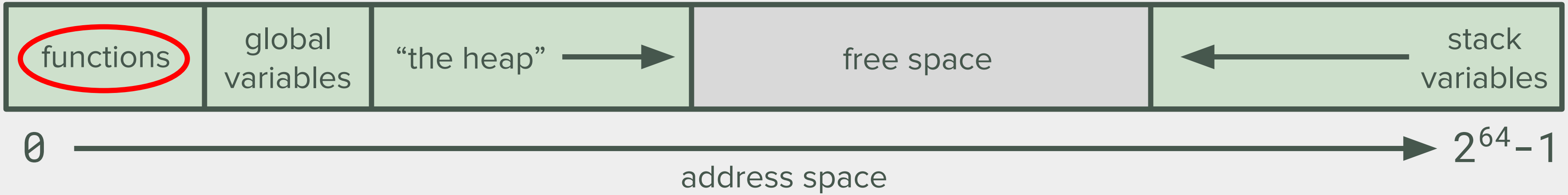
Function pointers

- Recall the memory layout map…



Function pointers

- Recall the memory layout map...



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

Declaring 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;
}
```

PURDUE UNIVERSITY

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?

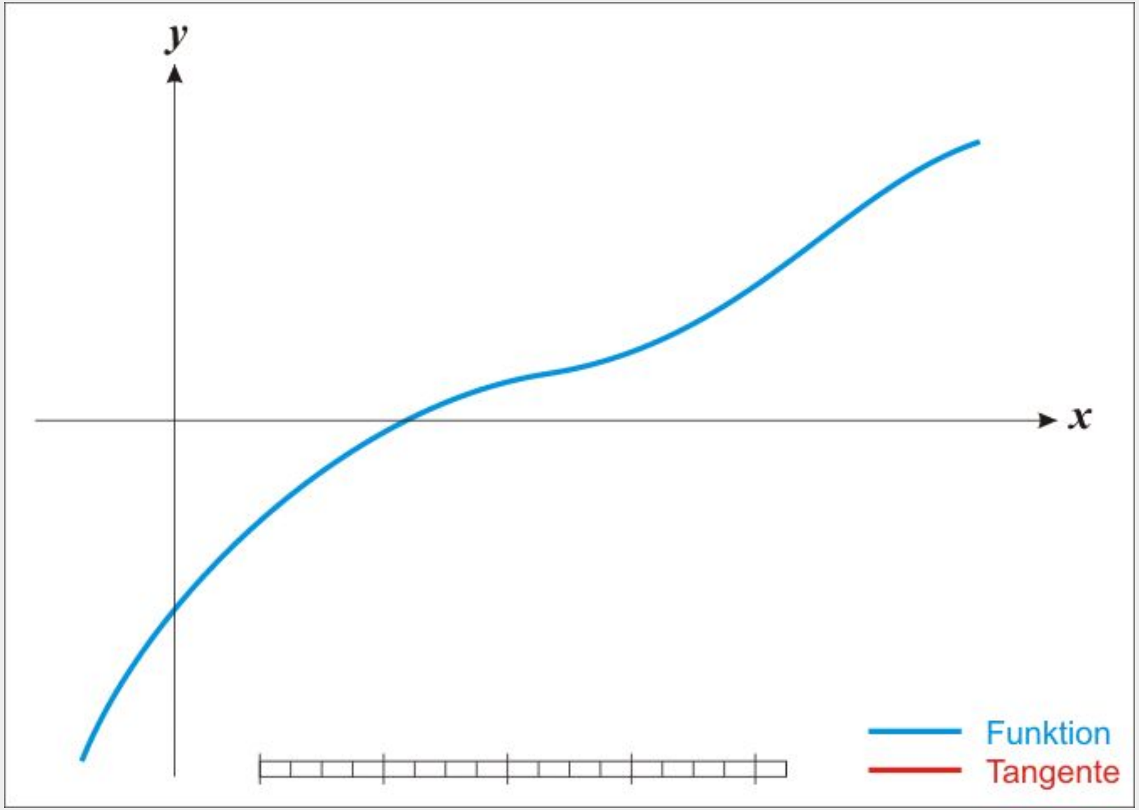
- Suppose we have a subroutine that uses Newton’s Method to locate a root of a polynomial function:

```
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);
```

Newton's Method



Source: [Wikipedia]

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 sqrt(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;
}
```

Another example: searching a list

- Suppose you have a list with many fields per node:

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

- What if we wanted to be able to search the list by any one of them?

List search

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

  while (ptr != NULL) {
    if (compare(ptr, item) == 0) {
      return ptr;
    }
    ptr = ptr->next;
  }
  return NULL;
}
```

Example comparison functions

```
int compare_name(struct node *ptr, char *item) {
  return strcmp(ptr->name, item);
}
```

```
int compare_title(struct node *ptr, char *item) {
  return strcmp(ptr->title, item);
}
```

```
ptr = list_search(compare_name, head, “Chris”);
```

```
ptr = list_search(compare_title, head, “Professor”);
```

What if we had different types?

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

- Could we compare salaries with our `list_search` function?

What if we had different types?

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

- Could we compare salaries with our `list_search` function?
  - No! (we'll revisit this problem later)

```
struct node *list_search(
  int (*compare)(struct node *, char *),
  struct node *ptr, char *item) { /* ... */ }
```

Function pointer syntax

- Can get quite cumbersome
- What does this line declare?

```
float foo(float (*bar)(int, long), float (*baz)(int, long));
```

Function pointer syntax

- Can get quite cumbersome
- What does this line declare?

```
float foo(float (*bar)(int, long), float (*baz)(int, long));
```

- We can simplify things using typedef

```
typedef float (*func_ptr_t)(int, long);
float foo(func_ptr_t bar, func_ptr_t baz);
```

- `foo` is a function prototype that returns a float and takes two function pointers `bar` and `baz` as its arguments.

PURDUE UNIVERSITY

We can also return function pointers

- But the syntax is very confusing

```
float (*foo(float (*bar)(int, long)))(int, long);
```

We can also return function pointers

- But the syntax is very confusing

```
float (*foo(float (*bar)(int, long)))(int, long);
```

Return type

We can also return function pointers

- But the syntax is very confusing

```
float (*foo(float (*bar)(int, long)))(int, long);
```

Argument list

Return type

We can also return function pointers

- But the syntax is very confusing

```
float (*foo(float (*bar)(int, long)))(int, long);
```

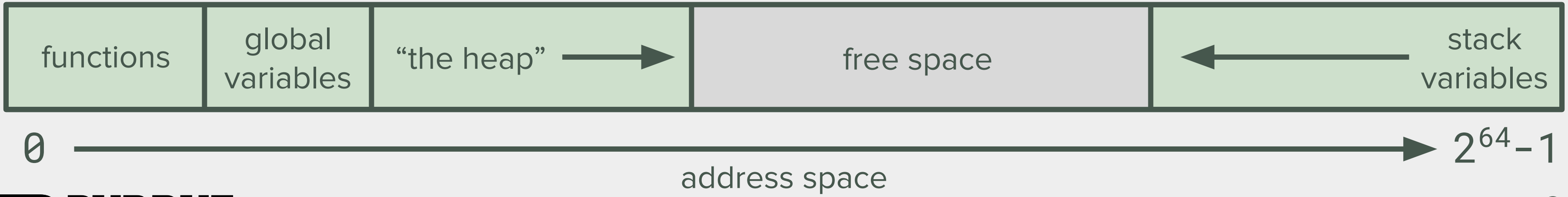
- Using typedef...

```
typedef float (*func_ptr_t)(int, long);
func_ptr_t foo(func_ptr_t bar);
```

PURDUE UNIVERSITY

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



Local variables

- Local variables are visible only within the function they're defined in
- If you define a variable *x* in func1() and func1() calls 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 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 it work?

Representation of countdown

```
countdown(n=2):
  if (n >= 0) {
    printf("%d...\n", n);
    countdown(n-1);
  }
  return;
```

```
countdown(n=1):
  if (n >= 0) {
    printf("%d...\n", n);
    countdown(n-1);
  }
  return;
```

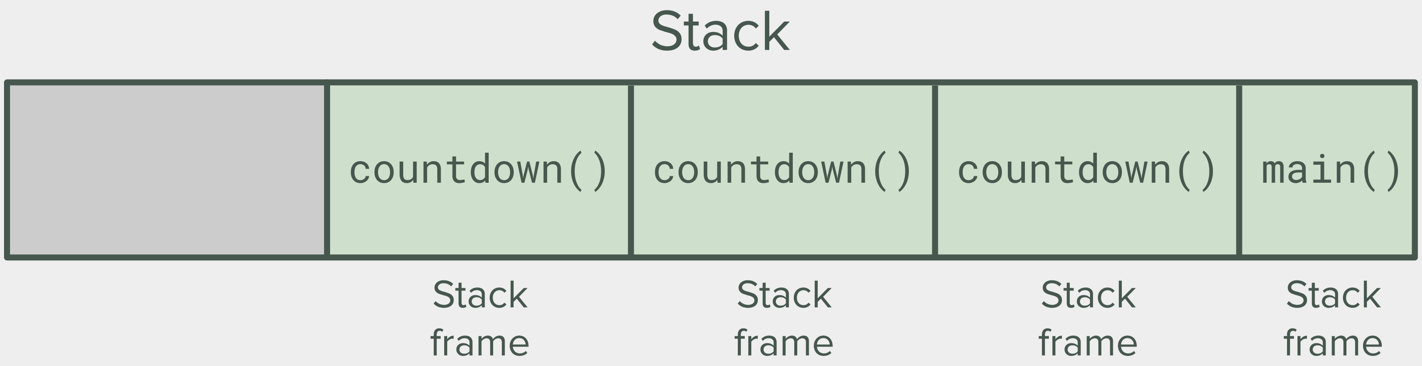
```
countdown(n=0):
  if (n >= 0) {
    printf("%d...\n", n);
    countdown(n-1);
  }
  return;
```

```
countdown(n=-1):
  if (n >= 0) {
    printf("%d...\n", n);
    countdown(n-1);
  }
  return;
```

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

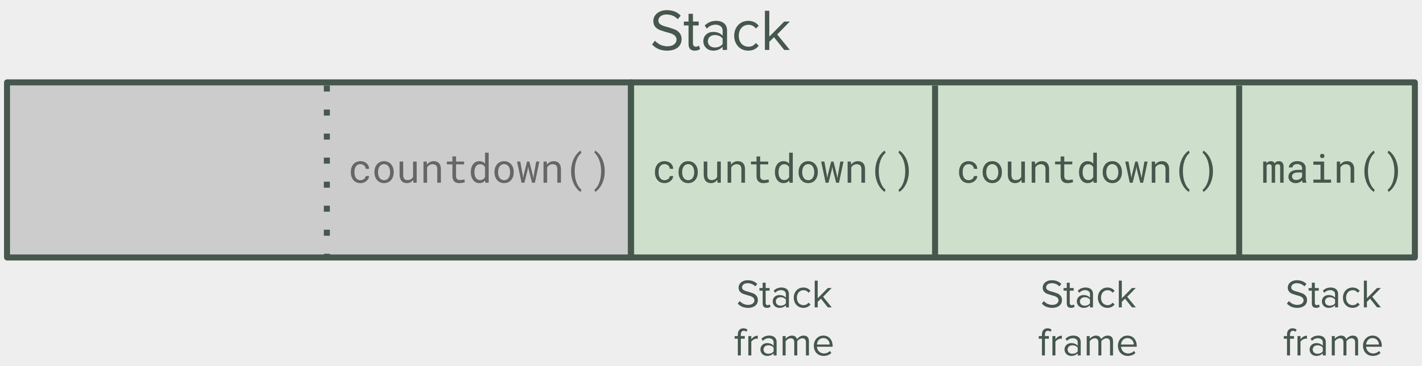
The stack

- Each 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
- This is called a stack frame



Never return pointers to local vars

- When a function returns, its reservation of the stack for local variables goes away
    - The “frame” is “popped” from the stack
    - But, variables won’t be overwritten until another function is invoked



Other recursion examples

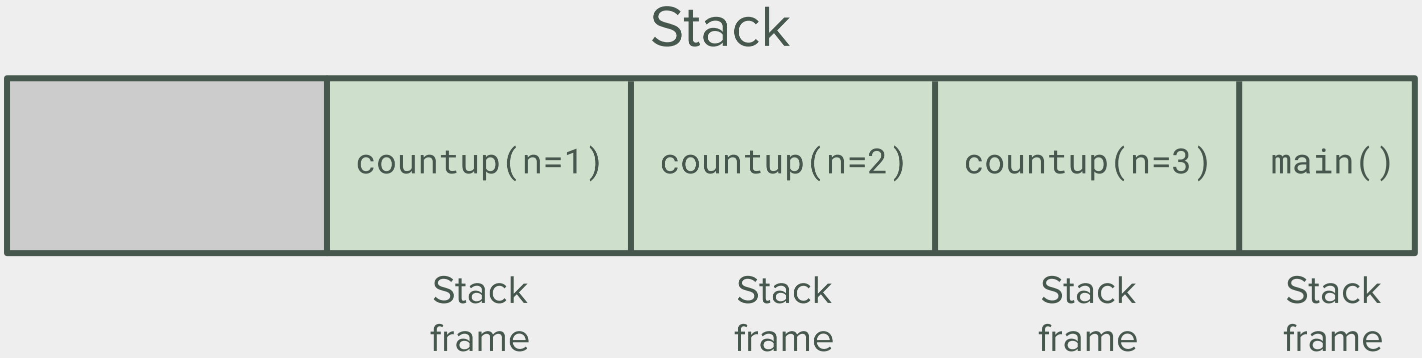
```
void countup(int n) {
  if (n >= 0) {
    countup(n-1);
    printf(“%d...\n”, n);
  }
  return;
}

int main() {
  countup(3);
  return 0;
}
```

Stack frames for countup()

```
countup(n=2)
  if (n >= 0) {
     countup(n-1);
     printf(“%d...\n”, n);
  }
  return;
```

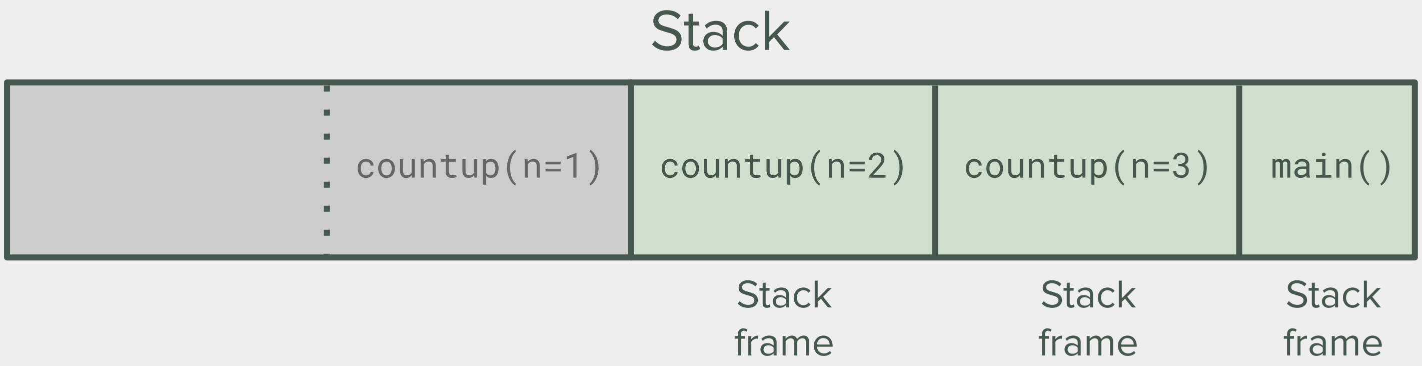
```
countup(n=1):
  if (n >= 0) {
     countup(n-1);
     printf(“%d...\n”, n);
  }
  return;
```



Stack frames for countup()

```
countup(n=2)
  if (n >= 0) {
     countup(n-1);
     printf(“%d...\n”, n);
  }
  return;
```

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

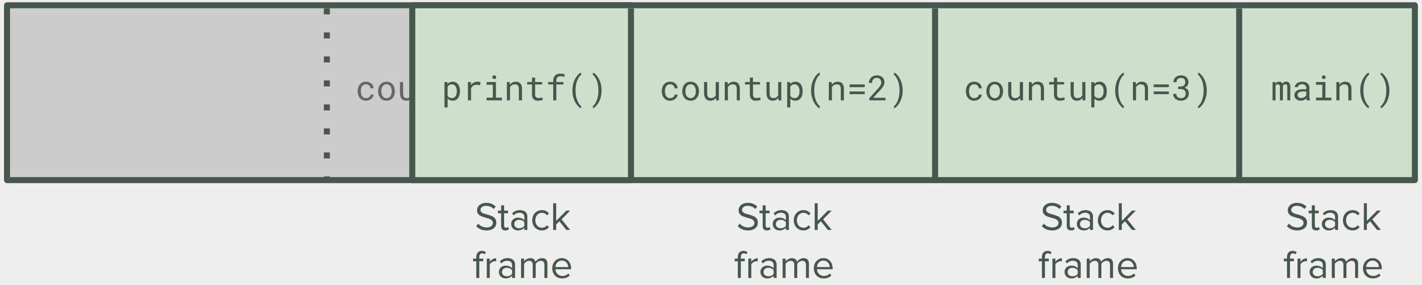


Stack frames for countup()

```
countup(n=2)
  if (n >= 0) {
     countup(n-1);
     printf(“%d...\n”, n);
  }
  return;
```

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

Stack



PURDUE UNIVERSITY

Other recursion examples

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

Other recursion examples

```
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
    - Sometimes called the “base case”
- Don’t return a pointer to something on the stack
- Don’t recurse too deeply
    - You will run out of stack space

For next lecture

- Read 4.10 in K&R
- Review pointers if needed
  - Chapter 5 in K&R, Chapter 12 in Beej
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.