Introduction to C

Memory Model

Instructor: Yin Lou

02/04/2011

Recap: Pointers

- int *ptr;
- Pointers are variables that store memory address of other variables
- Type of variable pointed to depends on type of pointer:

Recap: Pointers

- int *ptr;
- Pointers are variables that store memory address of other variables
- Type of variable pointed to depends on type of pointer:
  - int *ptr points to an integer value
  - char *ptr points to character variable

Recap: Pointers

- int *ptr;
- Pointers are variables that store memory address of other variables
- Type of variable pointed to depends on type of pointer:
  - int *ptr points to an integer value
  - char *ptr points to character variable
  - Can cast between pointer types: myIntPtr = (int *) myOtherPtr;

Recap: Pointers

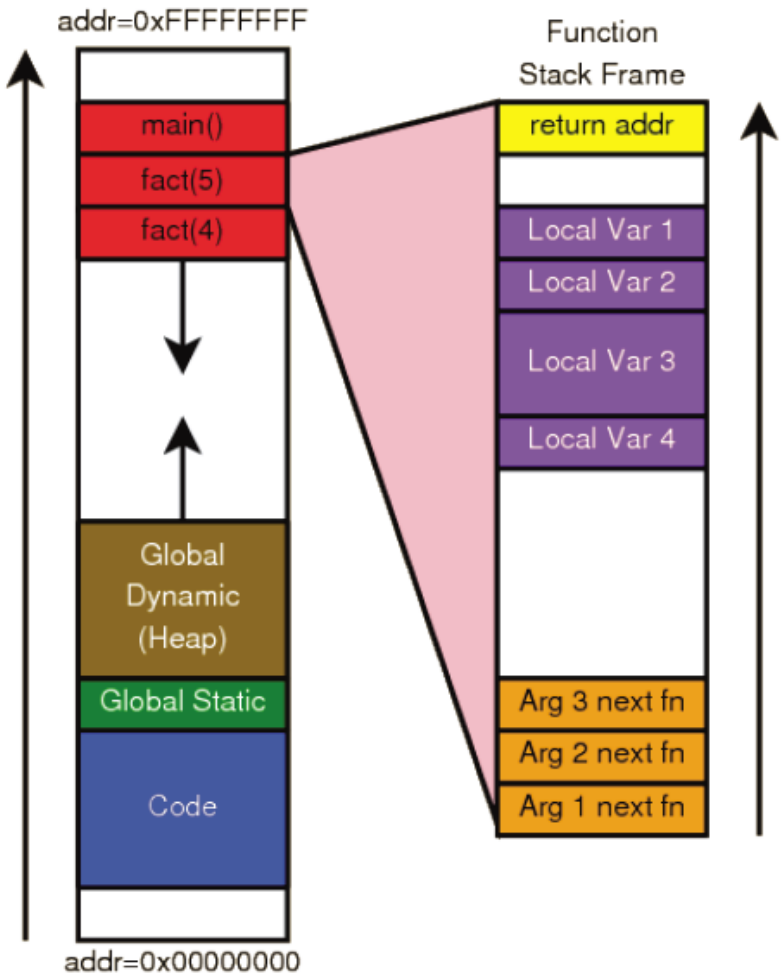
- int *ptr;
- Pointers are variables that store memory address of other variables
- Type of variable pointed to depends on type of pointer:
  - int *ptr points to an integer value
  - char *ptr points to character variable
  - Can cast between pointer types: myIntPtr = (int *) myOtherPtr;
  - void *ptr has an unspecified type (generic pointer); must be cast to a type before used

Recap: Pointers

- Two main operations
  - * dereference: get the value at the memory location stored in a pointer
  - & address of: get the address of a variable
  - int *myPtr = &myVar;
- Pointer arithmetic: directly manipulate a pointer's content to access other locations
  - **Use with caution!**: can access bad areas of memory and cause a crash
  - However, it is useful in accessing and manipulating data structures
- Can have pointers to pointers
  - int **my2dArray;

Memory

- Program code
- Function variables
  - Arguments
  - Local variables
  - Return location
- Global Variables
  - Statically allocated
  - Dynamically allocated



The Stack

Stores

- Function local variables
- Temporary variables
- Arguments for next function call
- Where to return when function ends

The Stack

Managed by compiler

- One stack frame each time function called
- Created when function called
- Stacked on top (under) one another
- Destroyed at function exit

What Can Go Wrong?

```
char *my_strcat(char *s1, char *s2)
{
    char s3[1024];
    strcpy(s3, s1);
    strcat(s3, s2);
    return s3;
}
```

What Can Go Wrong?

```
char *my_strcat(char *s1, char *s2)
{
    char s3[1024];
    strcpy(s3, s1);
    strcat(s3, s2);
    return s3;
}
```

- Recall that local variables are stored on the stack
- Memory for local variables is deallocated when function returns
- Returning a pointer to a local variable is almost always a bug!

What Can Go Wrong?

- Run out of stack space
- Unintentionally change values on the stack
  - In some other function's frame
  - Even return address from function
- Access memory even after frame is deallocated

The Heap

- C can use space in another part of memory: the heap

The Heap

- C can use space in another part of memory: the heap
  - The heap is separate from the execution stack
  - Heap regions are not deallocated when a function returns
  - Note: this is completely unrelated to the *Heap data structure*

The Heap

- C can use space in another part of memory: the heap
  - The heap is separate from the execution stack
  - Heap regions are not deallocated when a function returns
  - Note: this is completely unrelated to the *Heap data structure*
- The programmer requests storage space on the heap
  - C never puts variables on the heap automatically
  - But local variables might point to locations on the heap
  - Heap space must be explicitly allocated and deallocated by the programmer

malloc

- Library function in stdlib.h
  - Stands for memory allocate

malloc

- Library function in stdlib.h
  - Stands for memory allocate
- Requests a memory region of a specified size
  - Syntax: void *malloc(int size)
  - void * is generic pointer type

Usage

```
int main()
{
    int *p = (int *) malloc(10 * sizeof(int));
    if (p == NULL)
    {
        // do cleanup
    }
    // do something
    free(p);
    return 0;
}
```

Usage

```
int main()
{
    int *p = (int *) malloc(10 * sizeof(int));
    if (p == NULL)
    {
        // do cleanup
    }
    // do something
    free(p);
    return 0;
}
```

- MUST check the return value from malloc
- MUST explicitly free memory when no longer in use

What Can Go Wrong?

- Run out of heap space: malloc returns 0
- Unintentionally change other heap data
- Access memory after free'd
- free memory twice

Usage

```
int main()
{
    int *p = (int *) malloc(10 * sizeof(int));
    if (p == NULL)
    {
        // do cleanup
    }
    // do something
    if (p != NULL)
    {
        free(p);
        p = NULL;
    }
    return 0;
}
```

Garbage Collection in C

- Pointers make garbage collection difficult or impossible
- Its very difficult to determine whether memory is still being used
- Javas references are a restricted form of pointers that don't allow arithmetic, just because of this issue
- There are garbage collecting libraries for C, but they aren't guaranteed to work with any program

Garbage Collection in C

- Pointers make garbage collection difficult or impossible
- Its very difficult to determine whether memory is still being used
- Javas references are a restricted form of pointers that don't allow arithmetic, just because of this issue
- There are garbage collecting libraries for C, but they aren't guaranteed to work with any program

Example

```
char *s = (char *) malloc(1024);
s -= 10000;
// nothing points to the allocated memory
// region. Could it be garbage collected?
s += 10000;
// no, because now something points to it again!
```

Multidimensional Arrays

- On the stack: `int a[10][20];`
- Initialization: `int a[][] = {{1, 2, 3}, {4, 5, 6}};`
- Accessing the array: `a[1][0]`

Multidimensional Arrays

- On the stack: int a[10][20];
- Initialization: int a[][] = {{1, 2, 3}, {4, 5, 6}};
- Accessing the array: a[1][0]
- On the heap

```
int **a = (int **) malloc(10 * sizeof(int *));
for (int i = 0; i < 10; ++i)
{
    a[i] = (int *) malloc(20 * sizeof(int));
}
```

- Don't forget to free them!