# C Tutorial - Pointers

CS 537 – Introduction to Operating Systems

## The Stack

- The stack is the place where all local variables are stored
  - a local variable is declared in some scope
  - Example

```
int x;  // creates the variable x on the stack
```

- As soon as the scope ends, all local variables declared in that scope end
  - the variable name and its space are gone
  - this happens implicitly – the user has no control over it

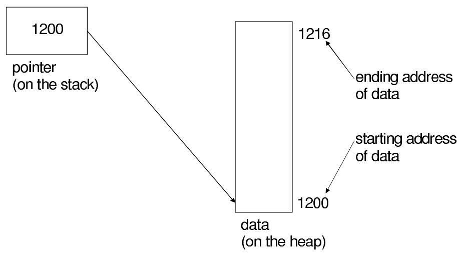
## The Heap

- The heap is an area of memory that the user handles explicitly
  - user requests and releases the memory through system calls
  - if a user forgets to release memory, it doesn't get destroyed
    - it just uses up extra memory
- A user maintains a handle on memory allocated in the heap with a *pointer*

## Pointers

- A pointer is simply a local variable that refers to a memory location on the heap
- Accessing the pointer, actually references the memory on the heap

## Basic Idea



## Declaring Pointers

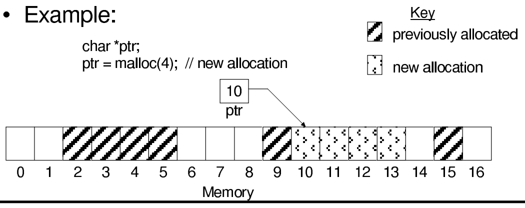
- Declaring a pointer is easy
  - declared like regular variable except that an asterisk (*) is placed in front of the variable
  - example

    ```
    int *x;
    ```
  - using this pointer now would be very dangerous
    - x points to some random piece of data
  - declaring a variable does not allocate space on the heap for it
    - it simply creates a local variable (on the stack) that will is a pointer
    - use *malloc()* to actually request memory on the heap

## malloc

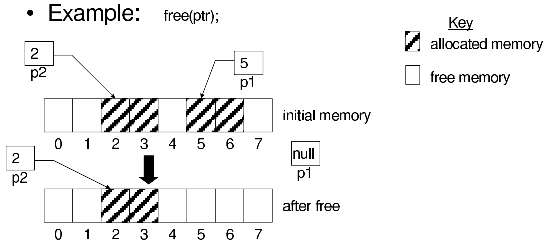
- Prototype: *int malloc(int size);*
  - function searches heap for *size* contiguous free bytes
  - function returns the address of the first byte
  - programmers responsibility to not lose the pointer
  - programmers responsibility to not write into area past the last byte allocated
- Example:

```
char *ptr;
ptr = malloc(4);  // new allocation
```



## free

- Prototype: *int free(int ptr);*
  - releases the area pointed to by ptr
  - ptr must not be null
    - trying to free the same area twice will generate an error
- Example: free(ptr);



## Using a Pointer

- To access a piece of data through a pointer, place an asterisk (*) before the pointer
  - example

```
char *ptr = malloc(1);
 *ptr = ‘a’;
 if(*ptr == ‘a’) { … }
```

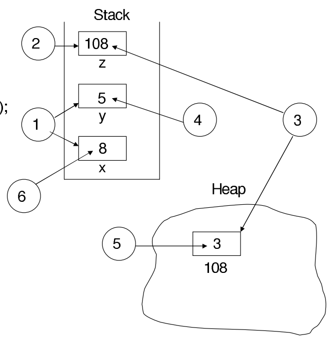
- Using the pointer without the asterisk actually accesses the pointer value
  - not the data the pointer is referencing
  - this is a very common mistake to make when trying to access data

## sizeof() Function

- The *sizeof()* function is used to determine the size of any data type
  - prototype: *int sizeof(data type);*
  - returns how many bytes the data type needs
    - for example: sizeof(int) = 4, sizeof(char) = 1
  - works for standard data types and user defined data types (structures)

## Simple Example

```
   int main() {
1     int x, y;
2     int *z;
3     z = malloc(sizeof(int));

4     y = 5;
5     *z = 3;
6     x = *z + y;
7     free(z);

      return 0;
   }
```



## Simple Example

1. Declare local variables x and y.
2. Declare local pointer z.
3. Allocate space on the heap for single integer. This step also makes z point to that location (notice the address of the space on the heap is stored in z's location on the stack.
4. Set the local variable y equal to 5.
5. Follow the pointer referenced by z to the heap and set that location equal to 3.
6. Grab the value stored in the local variable y and follow the pointer z to grab the value stored in the heap. Add these two together and store the result in the local variable x.
7. Releases the memory on the heap (so another process can use it) and sets the value in the z pointer variable equal to NULL. (this step is not shown on the diagram)

## Common Mistakes

- Using a pointer before allocating heap space

```
int *ptr;
*ptr = 5;
```

- Changing the pointer, not the value it references

```
int *ptr = malloc(sizeof(int));
ptr = 10;   // sets value on stack to 10, not value on the heap
```

- Forgetting to free space on the heap (memory leak)

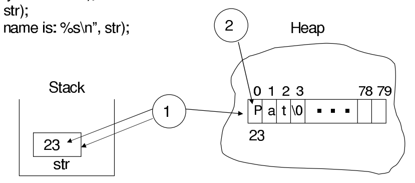
```
int *p1 = malloc(sizeof(int));
int *p2 = malloc(sizeof(int));
p1 = p2;  // making p1 point to p2 is fine, but now you can't free
          // the space originally allocated to p1
```

## Learning to Use Pointers

- DRAW PICTURES
  - when first using pointers it is much easier to draw pictures to learn what is happening
  - remember that an asterisk (*) follows the pointer
  - no asterisk (*) refers to the actual pointer variable on the stack

## One More Example

```
    #include <stdio.h>

    #define MAX_LINE 80

    int main() {
1     char *str = malloc(MAX_LINE * sizeof(char));

      printf("Enter your name: ");
2     scanf("%s", str);
      printf("Your name is: %s\n", str);
3     free(str);

      return 0;
    }
```



## One More Example

1. In one line, declare the pointer variable (gets placed on the stack), allocate memory on the heap, and set the value of the pointer variable equal to the starting address on the heap.
2. Read a value from the user into the space on the heap. This is why scanf takes pointers as the parameters passed in.
3. Release all the space on the stack pointed to by str and set the value of the str pointer on the stack equal to null. (step not shown)

## Dereferencing

- Pointers work because they deal with addresses – not value
  - an operator performs an action at the value indicated by the pointer
  - the value in the pointer is an address
- We can find the value of any variable by dereferencing it
  - simply put an ampersand (&) in front of the variable and you now have the address of the variable

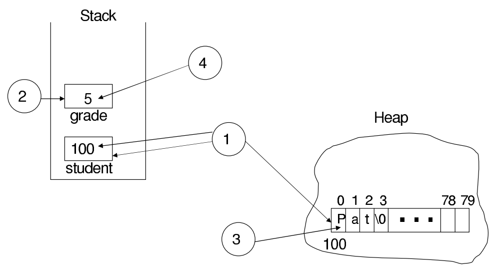
## Revisiting scanf()

- Prototype: *int scanf(char* str, void*, void*, …);*
- What is void*?
  - void* is similar to *object* in Java
  - it can point at anything
- Since the data types being passed into scanf can be anything, we need to use void* pointers
- If you want to scan a value into a local variable, you need to pass the address of that variable
  - this is the reason for the ampersand (&) in front of the variable

## scanf() Example

```
    #include <stdio.h>

    #define MAX_LINE 80

    int main() {
1     char *student = malloc(char * sizeof(char));
2     int grade;

      printf("Enter student's name: ");
3     scanf("%s", student);
      printf("Enter student's grade: ");
4     scanf("%d", &grade);
      printf("%s received a %d\n", student, grade);
5     free(student);

      return 0;
    }
```

## scanf() Example



## scanf() Example

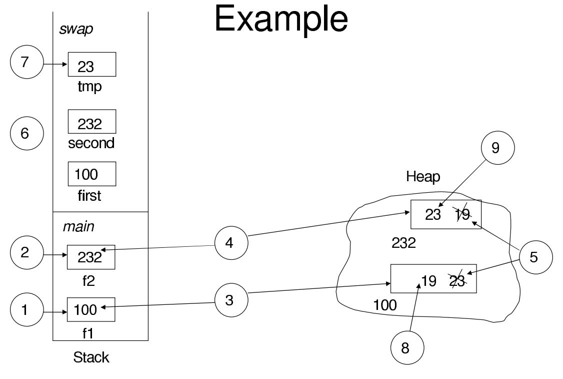
1. In one line, declare the pointer variable (gets placed on the stack), allocate memory on the heap, and set the value of the pointer variable equal to the starting address on the heap.
2. Create the local variable grade on the heap.
3. Read a value from the user into the space on the heap – beginning at the address indicated by the pointer variable on the stack.
4. Read a value from the user into the address referred to by the address of grade.
5. Release all the space on the stack pointed to by student and set the value of the student pointer on the stack equal to null. (step not shown)

## Pointers and Functions

- One limitation of functions is that they only return a single value
- So how to change multiple values in a single function?
  - pass in pointers
  - now any changes that are made are made to the address being referred to
  - this changes the value for the calling function as well as the called function

```
  #include <stdio.h>

  void swap(float*, float*);

  int main() {
1     float *f1, *f2;
2     f1 = malloc(sizeof(float));
3     f2 = malloc(sizeof(float));

      printf("Enter two numbers: ");
5     scanf("%f%f", f1, f2);  // assume the user types 23 and 19
      printf("f1 = %f\tf2 = %f\n", *f1, *f2);
6     swap(f1, f2);
      printf("After swap: f1 = %f\tf2 = %f\n", *f1, *f2);
      free(f1); free(f2);

      return 0;
  }

  void swap(float* first, float* second) {
7     float tmp = *first;
8     *first = *second;
9     *second = tmp;
  }
```

## Example



## Example

1. Declare a pointer, f1, on stack.
2. Declare a pointer, f2, on stack.
3. Allocate space on the heap for a float and place the address in the pointer variable f1.
4. Allocate space on the heap for a float and place the address in the pointer variable f2.
5. Read values from the user. Hand scanf() the pointers f1 and f2 and the data gets put on the heap.
6. Call the swap function. This pushes a new entry in the stack. Copy the value of the pointers f1 and f2 into first and second.
7. Create a new local variable tmp. Follow the pointer of first and place its value into temp.
8. Follow the pointer of second, grab the value, follow the pointer of first, place grabbed value there.
9. Grab the value from tmp, follow the pointer of second, place the grabbed value there.

## Lists

- Remember structures?
  - structures together with pointers can be used to make a list
- Some of the data in a structure will contain the information being stored
- One of the fields in the structure will be a pointer to the next structure in the list

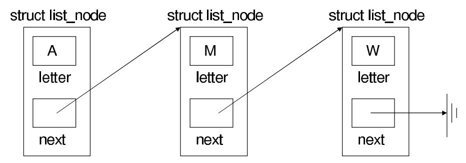
## Lists

- Example of a structure used in a linked list

```
struct list_node {
  char letter;
  struct list_node *next;
}
```

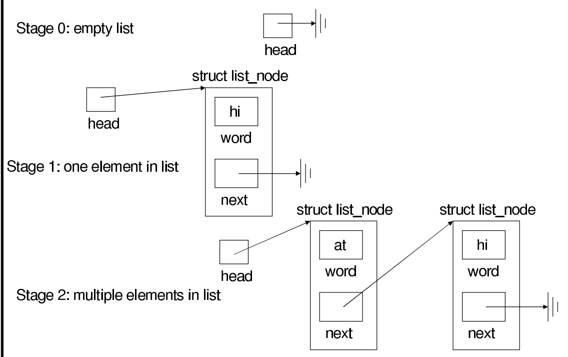
- The *letter* variable is the data to be stored
- The *next* variable will point at the next element in the list
  - or NULL if there are no more elements

## Lists



```
#include <stdio.h>
#include <string.h>

typedef struct list_node {
      char word[20];
      struct list_node* next;
} list_node;

int main() {
      list_node* head = NULL;
      char str[20];

      printf(“Enter a word: “);
      scanf(“%s”, str);
      while(str[0] != ‘\n’) {
         list_node* tmp = (list_node*)malloc(sizeof(list_node));
         strcpy(tmp->word, str);
         if(head)    { tmp->next = tmp; }
         else        { tmp->next = NULL; }
         head = tmp;

         printf(“Enter a word: “);
         scanf(“%s”, str);
      }
      return 0;
}
```

## Example

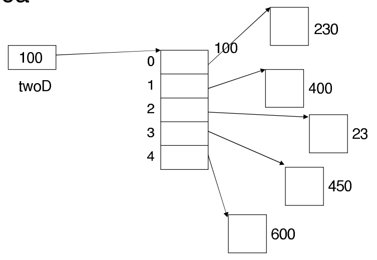


## 2-D Pointers

- To really make things confusing, you can have pointers to pointers
  - and pointers to pointers to pointers …
- This comes in handy whenever a 2-D array is needed
  - you can also declare 2-D arrays, but these go on the stack
  - if dynamic memory is needed, must use pointers
- Declaring a pointer to a pointer
  - just put 2 asterisks (*) in front of the variable
  - example
    - char **names;

## 2-D Pointers

- Basic idea



## argv

- Up until now, main has been written
  - *int main() { … }*
- This is okay, but it's usually written
  - *int main(int argc, char** argv) { … }*
- argc
  - number of command line arguments being passed in
    - this counts the name of the program
- argv
  - this is an array of strings – a 2-D character array
  - each string represents one command line argument

## Example

```
#include <stdio.h>

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

    printf("Number of arguments: %d\n", argc);
    for(i=0; i<argc; i++)
       printf("argument %d: %s", i, argv[i]);

    return 0;
}
```

## Example

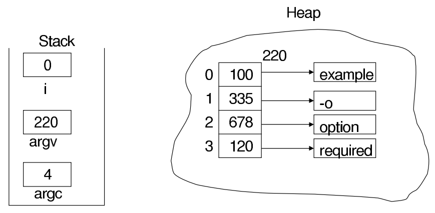
- Given the following command line

  *prompt> example –o option required*

- The output of the sample program

```
Number of arguments: 4
argument 0: example
argument 1: -o
argument 2: option
argument 3: required
```

## Example



## Creating a 2-D Array

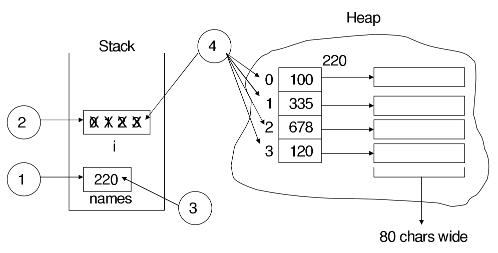
- Assume a 2-D array of characters is needed
  - this is basically an array of strings
- Assume 4 strings with a max of 80 chars

```
  int main() {
1   char** names;
2   int i;

3   names = (char**)malloc(4 * sizeof(char*));
4   for(i=0; i<4; i++)
      names[i] = (char*)malloc(80 * sizeof(char));

5   for(i=0; i<4; i++)
      free(names[i]);
6   free(names);

    return 0;
  }
```

## 2-D Arrays

- Once you really understand this previous example, you are well on your way to understanding pointers
- Let’s take a closer look at exactly what is going on

## 2-D Arrays



## 2-D Arrays

1. Create a pointer on the stack that will point to a group of pointers
2. Create a local variable on the stack
3. Make names point to an array of 5 pointers to characters. This array is located on the heap.
4. Go through each pointer in the array and make it point at an 80 character array. Each of these 80 character arrays is also located on the heap
5. Freeing each of the 80 character arrays. (not shown on diagram).
6. Free the array of pointers. (not shown on the diagram)