Pointers

September 18, 2019 8:06 AM

Declaring pointers

- type of pointer not the same as type it points to
  - ex. int* var1, var2; declares var1 as pointer, but var2 as integer
  - *int *var1, *var2;*

Address operator and dereferencing

- value which pointer points to can be accessed by dereferencing the pointer
- using * operator

  int x = 23;
  int p = &x;*
  x = 47;
  **p = 38;*
- can use pointers as parameters

  int getArraySum(int arr[], int size, int pcount)*

Pointers as parameters

void f1(int arg) { arg = 22 }
void f2(int arg) { *arg = 410)*

- f1(x): arg = 22, prints x = 45
- f2(&x): arg = 410, x = 410
- *arg accesses the value at the address &x, and changes it to 410
- **Swap vars**
  - Using vars:

    void swap(int a, int b) {
    int temp = a;
    temp = a;
    a = b;
    b = temp;
    }
    int x=3; int y=5; swap(x,y)
  - Using pointers:

    void swap(int a, int* b) {*
    int temp = a;
    *temp = *a;*
    **a = *b;*
    **b = temp;*
    }
    int x=3; int y=5; swap(&x,&y)

```
int main() {
  int a = 5;
  int* p = &a; // assume 0x5fbff8cc

  printf("value of p = %p\n", p);
  printf("dereferenced p = %d\n", *p);

  p++;

  printf("value of p+1 = %p\n", p);
  printf("dereferenced p+1 = %d\n", *p);
}
```

- ex. int* p = 0x5fbff8cc, int a = 5

  prints: 0x5fbff8d0, null (no value!)

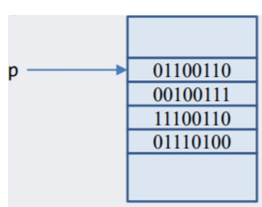
Pointer Types

- int* p points to integer (32-bit address)
- char* p pointer to character (32-bit address)
- dereference p without knowing the type of var it is pointing to? (int=4 bytes, char=1 byte)
- generic pointers can be declared, but must be cast before dereferenced (*void**)

```
int main() {
  int x = 10;
  char ch = 'A';
  void* gp;

  gp = &x;
  printf("integer value = %d\n", *(int*)gp); // outputs 10

  gp = &ch;
  printf("now points to char %c\n", *(char*)gp); // outputs A
```



```
    printf("integer value = %d\n", *(int*)gp); // outputs 10

    gp = &ch;
    printf("now points to char %c\n", *(char*)gp); // outputs A

    return 0;
}
```

Pointer to pointers

- can keep adding levels of poiners until brain explodes or compiler melts
- pointer to pointer: *int = 5; int* p = &x; *p = 6; int** q = &p; int*** r = &q;*

Passing array elements as parameters

double getMAximum(double data[], int size);
double getMaximum(double data, int size)*

- do not need to provide & when specifying address of entire array
- if want to specify address of individual element, need address operator

```
void incrementval(int* num) {
    (*num)++
}
incrementNum(&data[5])
```

Pointer Arithmetic

- if know address of first element,can compute address of other array elements

```
int x = 5;
int* p = &x;
printf("p address: %p\n", p);
printf("p value: %d\n", *p);
printf("p+1 value: %p\n", *(p+1));
```

- address of next element: (p+1)

Dynamic memory

- Arrays declared as local vars must have known size at complie time
- Sometimes don't know how much space we need until runtime
- what if user needs more or less stack space? (change code, or waste memory)

Memory management in C

- Locally declared vars placed on function call stack, managed automatically
- available stack space extremely limited
- placing many large var or data struct on stack leads to stack overflow
- Stack vars only exist as long as function that declared them is running

Dynamic memory allocation

- at run-time, can request extra space on-the-fly from the memory heap
- request memory from the heap, and return allocated memory to heap when no longer needed (deallocation)
- unlike stack memory, items allocated on heap explicitely freed by programmer
- func *malloc* returns pointer to memory block of at least *size* bytes

    ptr = (cast-type) malloc(byte-size);*

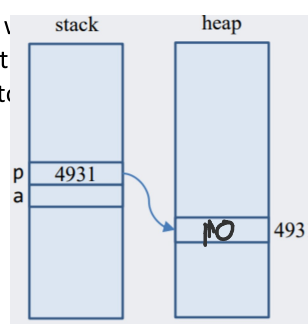
- Function free returns memory block

    free(ptr);

```
int main() {
  int a;
  int *p = (int*) malloc(sizeof(int));
  *p = 10;
}
```

Heap example

- if no free memory left on heap, malloc \
- malloc only allocates space, but not init
- use *calloc* to allocate and clear spaces t



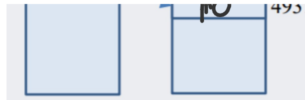

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int* i;
  i = (int*) malloc(10*sizeof(int));
  if (i == NULL) {
    printf("Error: can't get memory...\n");
    exit(1); // terminate processing
  }
```

Allocating dynamic arrays

- can use array index notation to access heap
- can use non constant

notation to access
heap
- can use non constant
variables



```
  if (i == NULL) {
    printf("Error: can't get memory...\n");
    exit(1); // terminate processing
  }

  i[0] = 3;  // equivalent: *(i+0) = 3;
  i[1] = 16; // *(i+1) = 16;
  printf("%d", *i);
  ...
}
```

Allocating dynamic arrays

- Want to allocate space for exactly 10 ints in array
  *int*i*
  i = (int) malloc(10*sizeof(int));*
  i[0] = 3;
  i[1] = 16;
- equivalent to **(i+0) = 3,* *(i+1) = 16

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

int main() {
  int employees, index;
  double* wages;
  printf("Number of employees? ");
  scanf("%d", &employees);

  wages = (double*) malloc(employees * sizeof(double))
  if (!wages) { // equivalent: if (wages == NULL)
    printf("Error: can't get memory...\n");
  }

  printf("Everything is OK\n");
  ...
}
```

See dma_exa

Dangling pointers

- When done with allocated object, free it so that system can reclaim and reuse memory

```
int main() {
    int* i = (int*) malloc(sizeof(int));
    *i =5 ;
    free(i);
    printf("d", *i);
}
```

- if pointer continues to refer to deallocated memory, behave unpredictably when dereferenced
- dangling pointer, set pointer to NULL after freeing i = NULL;

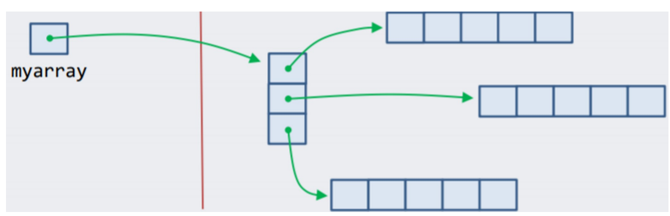
Memory leaks

```
int*arr
int sz = 4;
arr = (int*) malloc(sz*sizeof(int));
arr[2] = 5;
arr = (int*) malloc (sz*sizeof(int));
arr[2]=7;
```

Dynamic allocation of a 2D array

```
int dim_row = 3;
int dim_col = 5;
int** myarray;
```

- ** pointer to pointer



Stack vs heap

- Stack:
  - fast access
  - alloc,dealloc automatically manages
  - memory not becore fragmented
  - local vars only
  - limit on stack size
  - vars cannot be resized
- Heap
  - vars accessible outside declaration scope
  - no limit on memory
  - vars can be resized
  - slower access
  - no guareneed efficient use of space
  - memory management