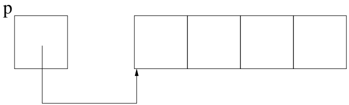
Dynamic Memory

R. Inkulu
http://www.iitg.ac.in/rinkulu/

Types of memory allocations

- auto local
  - allocated on stack and uninitialized by default
  - accessible in the function that it got defined
  - requesting to allocate in *registers* for faster access; the request is not necessarily entertained
- auto global
  - allocated in data area and initialized by default to zeros
  - accessible across the program
- auto static
  - allocated in data area and initialized by default to zeros
  - accessible among all functions defined within a file in which the variable is defined
- dynamic
  - useful if the size to be allocated is not known at compile time
  - allocated on heap and inititalized if requested
  - though not global/static, may be used across functions

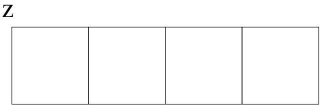
Intro to dynamic memory

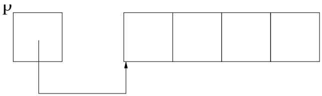



```
double *p = (double *) malloc(count*sizeof(double));
  //allocates count*sizeof(double) bytes contiguously;
  //p has the address of first byte of those allocated
int i;
for (i=0; i<count; i++) {
    p[i] = i*10;
    printf("%lf, ", p[i]);
}
//prints 0.000000, 10.000000, 20.000000, 30.000000,
free(p);   //frees contigous memory referred by p
```

- useful when the number of objects to be allocated is not known at compile-time
- system maintains a table of dynamically allocated memory blocks and their corresp. address ranges

Dynamic vs non-dynamic memory allocation





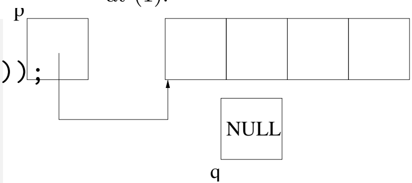

```
double z[4]; //z is from stack
double *p = (double*) malloc(count*sizeof(double));
   //p is from stack, and memory block is from heap
...
free(p);
```

- memory for auto variables (including arrays) comes from *stack*, whereas the dynamic memory comes from *heap* of the process address space

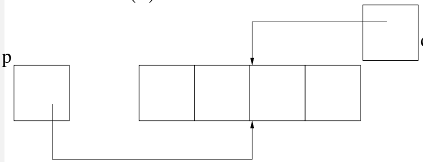
Pointer arithmatic

```
double *p = (double*)
malloc(count*sizeof(double));
double *q = NULL;
...  //denotes irrelevant
     //stmts
//i'm here (1)
p += 2;
...
q = p;
//i'm here (2)
...
free(p-2);
```

at (1):



at (2):



- pointer arithmatic is same whether the memory referred by a pointer is either from the stack or heap

malloc and free pairs

```
double *p=NULL; int *q = NULL;
...
p = (double*) malloc(countA*sizeof(double));
...
q = (int*) malloc(countB*sizeof(int));
...
free(p);
...
free(q);
```

- in any program, *malloc* and *free* invocations must exist in pairs

  avoiding *free* call corresp. to any *malloc* call causes *memory leak*

- *malloc* and *free* corresp. to a block of memory

  not necessarily invoked in the same function

  not necessarily bracketed

Returning pointers from functions

```
double *func(int count) {
     double *p = NULL;
     ...
     p = (double*) malloc(count*sizeof(double));
     ...
     return p; }
void func1(int countA) {
     int count; double *q = NULL;
     ...
     q = func(count);
     ...
     free(q);   //freeing heap memory allocated in func
     ...
     return; }
```

- heap memory referred by a pointer may require to be freed by the caller (precise protocol need to be defined)
- does not make sense to return the address of a local variable (ex. array) from a function

Returning pointers from functions (cont)

```
void *malloc(size_t sizeInBytes) {
    ...
}
```

- *void func()* returns nothing
  whereas *void* func()* returns pointer to a block of memory (could be NULL too) that can contain objects of any *type*
- while 'void' signifies *nothing*; 'void*' denotes pointer to objects of *any type*
- any 'type*' is implicitly typecasted to 'void*' whereas the other way around requires an explicit typecast

Dynamic memory: advantages and disadvantages

advantages:

- useful when the number of objects to be allocated is not known at compile-time
- gives flexibility to allocate and deallocate memory based on the need; careful user of this primitive can extract benefits
- although not global, available till the memory is freed

disadvantages:

- slow due to free/allocated heap space maintainance involved together with the defragmentation overhead

  due to intermittent *malloc*s and *free*s
- forgetting to deallocate memory causes memory leak

Non-dynamic memory: advantages and disadvantages

advantages:

- compiler will deallocate the memory automatically for all the global, static, and local memory that it allocated: no memory leak hassles

disadvantages:

- memory allocation and deallocation are not in the control of user
- for auto local variables, memory is always deallocated at the end of the function
- for auto global/auto static variables, memory is always deallocated at the end of the program

A note on modern compilers: simulating array behavior with dynamic memory[1]

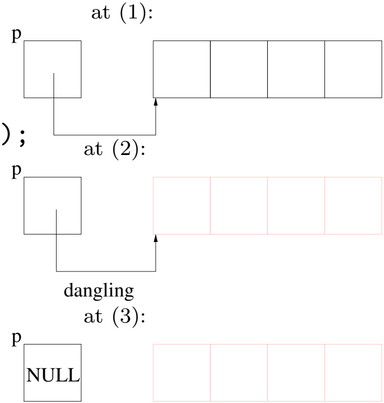
```
...
double A[count];
printf("%p, %p, %p, %p", &A, A, &A[0], (A+2));
//prints 0xbf99c9c8, 0xbf99c9c8, 0xbf99c9c8, 0xbf99c9d8
...
```

- memory allocation is done on heap
- buffer allocated is named with the array name
- heap memory is deallocated at the end of the scope of the referring variable
- disadv: code may not be portable across all the compilers

[1] for this course purpose, we always assume ANSI standard; hence, this kind of usage is not permitted

Avoid dangling pointers after freeing

```
...
double *p = (double*)
malloc(count*sizeof(double));
...
//i'm here (1)
free(p);
...
//i'm here (2)
p = NULL;
//i'm here (3)
...
```



- avoid dangling pointers by resetting pointer variable value immediately after the free

Few memory related errors

- out-of-bounds read or write
- reading/writing already freed memory
- freeing non-dynamically allocated memory
- freeing a block of heap memory multiple times
- memory that has no pointer in the program (*memory leak*)
- program has only a pointer to the middle of allocated memory (*potential memory leak*)
- dereferncing a null pointer
- reading uninitialized memory

Few useful functions from stdlib.h

void *malloc(size_t numBytes)

— avoid malloc(0) as this is not portable

void free(void *p)

void *calloc(size_t numObj, size_t sizeOfAObject)

— same as malloc but the allocated memory is initialized with zeros; in some systems, allocates memory but fills with zeros when a write happens

void *realloc(void *oldMem, size_t numBytes)

— resizes and where necessary relocates the block pointed by p; moves the contents of *p to the new location; frees the old memory

— when oldMem is NULL, works same as malloc

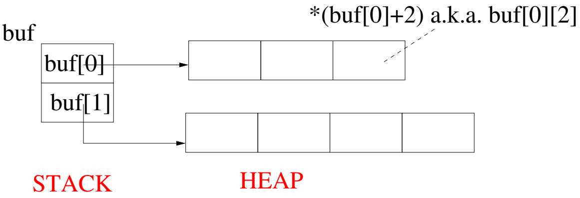
void *memcpy(void *to, void *from, int numBytes);

— cannot handle the overlap

void *memmove(void *to, void *from, int numBytes);

— same as memcpy but handles the overlap

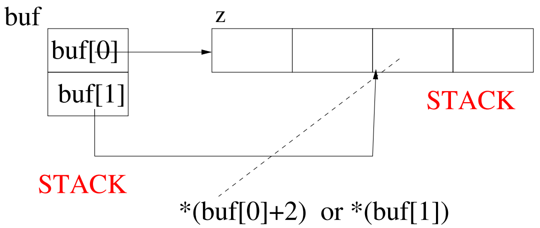
Array of pointers to varying sized arrays




```
double *buf[2];
buf[0] = (double*) malloc(countA*sizeof(double));
buf[1] = (double*) malloc(countB*sizeof(double));
printf("%d, %d \n", sizeof(buf[0]), sizeof(buf));
   //prints sizeof(void*), 2*sizeof(void*)
...
free(buf[1]);
free(buf[0]);
```

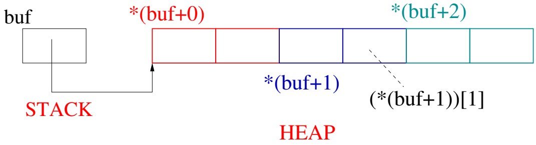
- buf is an array[2] of pointers, each entry of which points to a block of memory that contains zero or more doubles

Array of pointers to varying sized arrays (cont)




```
double z[4];
double *buf[2];
buf[0] = &z[0];
buf[1] = &z[2];
printf("%d, %d \n", sizeof(buf[0]), sizeof(buf));
   //prints 4, 8
...
//no need of free calls
```

Array of pointers to fixed size arrays




```
double (*buf)[2] =
  (double (*)[2])malloc(count*sizeof(double [2]));
printf("%d, %d, %d, %d \n",
   sizeof(double), sizeof(buf[0]),
   sizeof(buf[2]), sizeof(buf));
   //prints 8, 16, 16, 4 when count is 3
...
free(buf);
```

- buf points to count number of array[2]s of doubles
  in other words, buf[0] is the zeroth array[2]; buf[1] is the first array[2]; etc., (in other words, buf is a two dimensional array with dimensions *count* $\times$ 2: count number of rows while each row has two columns)

Definitions to memory-layouts (review)

(i) double a[2];

double *p = &a[0];

(ii) double *p = (double *) malloc(count*sizeof(double));

(iii) double a[2][3];

(iv) double *a[2];

a[0] = (double *) malloc(countA*sizeof(double));

a[1] = (double *) malloc(countB*sizeof(double));

(v) double *a[2], b[2][3];

a[1] = b[0];

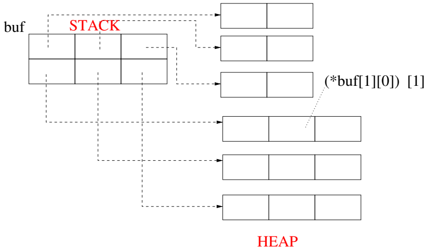
(vi) double (*b)[2];

b = (double (*)[2]) malloc(count*sizeof(double [2]));

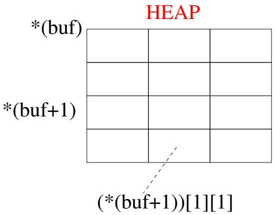
(vii) double (*b)[2], c[4][2];

b = c;

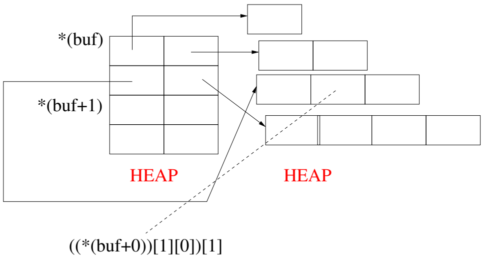
Memory-layouts to definitions



*double *buf[2][3]*



*double (*buf)[2][3]*



*double *(*buf)[2][2]*