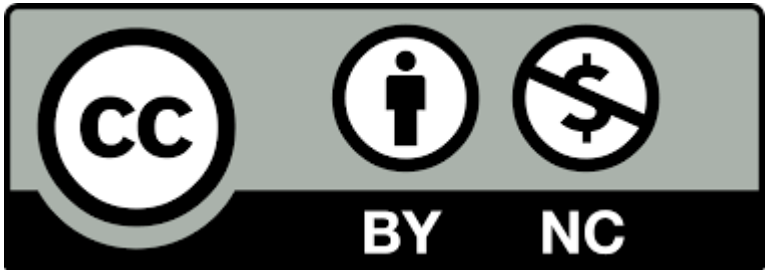
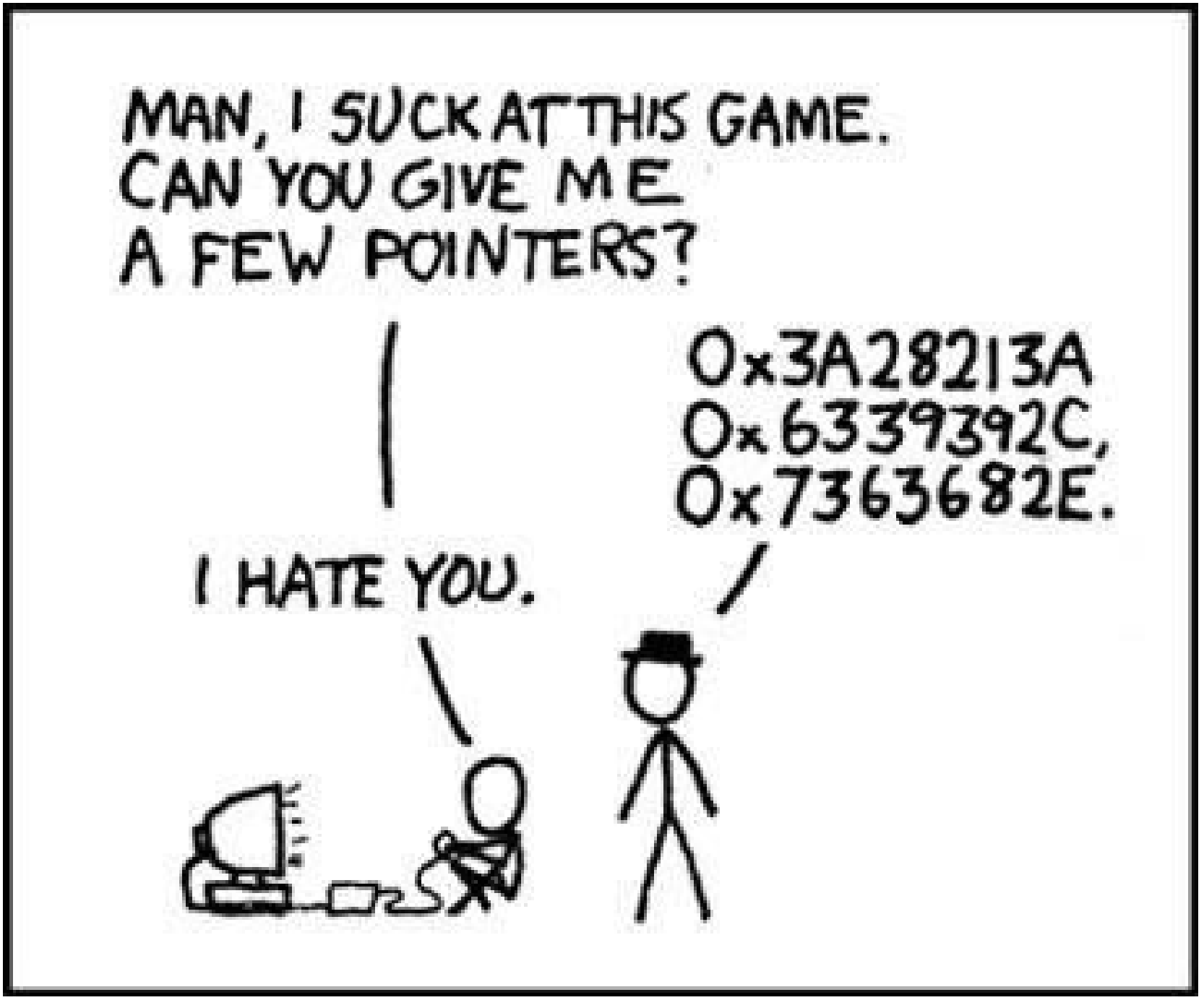
Pointers in C

Abhijit A.M.
abhijit13@gmail.com





Courtsey: Programmer Jokes on telegram

Pointer

- **Pointer is a variable which can store addresses**
- **Pointer has a “type” (except void pointer)**
  - **e.g. int *p; char *cp; double *dp;**
  - **Here p, cp, dp are respectively pointers to integer, character, and double**
- **Size of pointer is decided by compiler**

Operations on (and related to) Pointers

&

*****

=

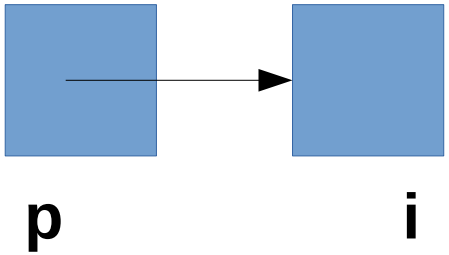
+int -int

-

[ ]

Operations related to pointers: &

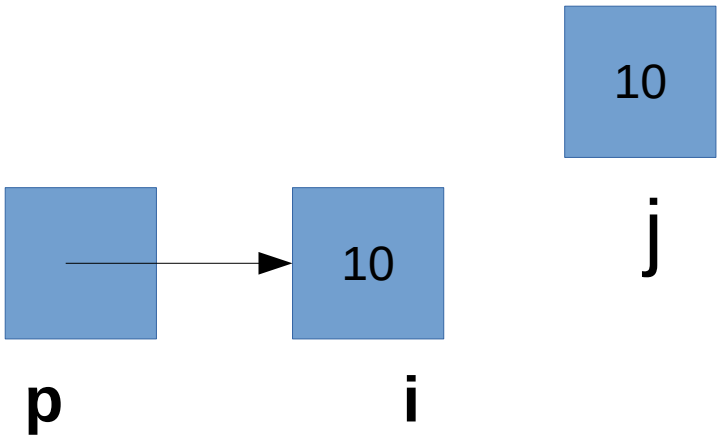
```
int *p, i;
p = &i;
```



- & fetches the address of variable
  - Called Referencing operator
  - Here, &i is address of i
  - RHS is address of integer, LHS is variable which can store address of integer
- Diagram of this operation is shown on left side
- No need of assuming some value for address (e.g. address 1028). Just the diagram is sufficient to understand the concept

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

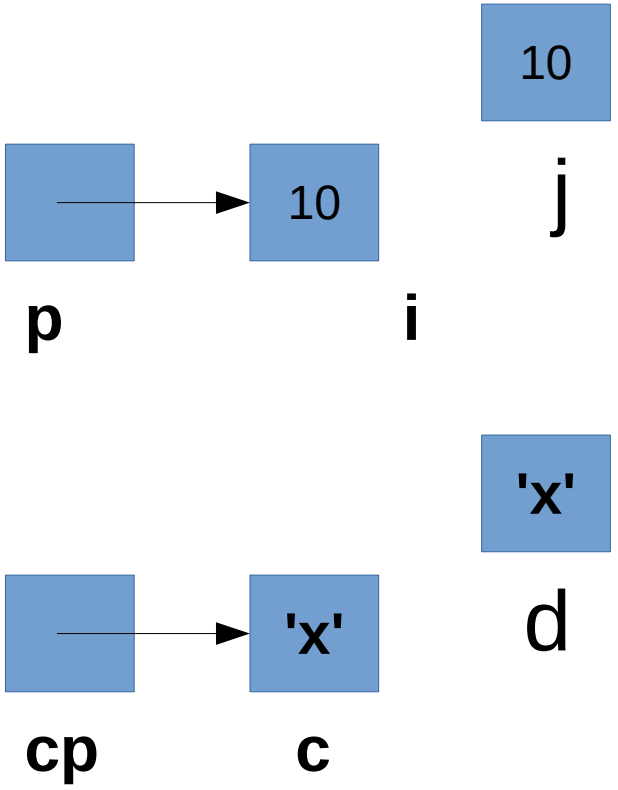


- * fetches the value stored at a given address
  - Called dereferencing operator
- *p : here “value of p” is itself an address *(of i)*, so *p is value stored at “value of p” that is i
- Diagram's make it easy to understand, *p is simply the contents of box p points to

Operations on pointers: *

```
int *p, i, j;
i = 10;
p = &i;
j  = *p;
```

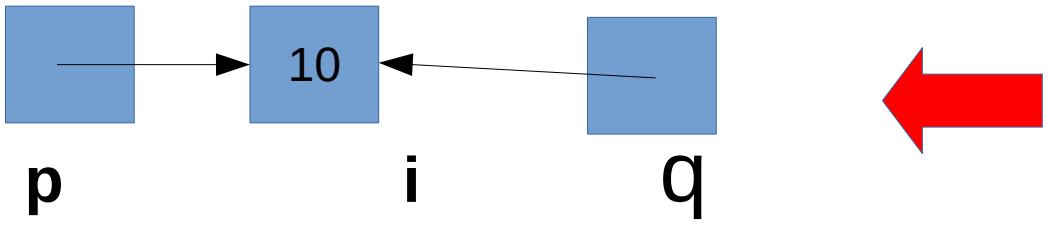
```
char *cp, c, d;
c = 'x';
cp = &c;
d  = *cp;
```

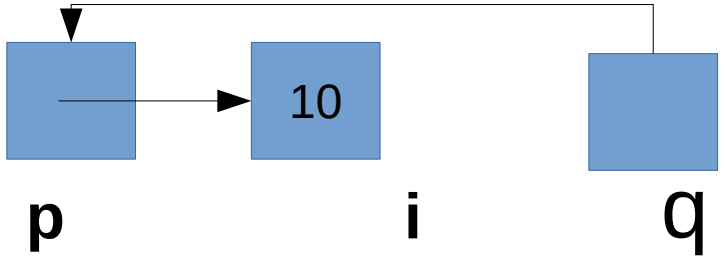


- What is the difference between * in the two codes?
- The *p fetches the value at given address in sizeof(int) bytes, while *cp fetches the value at given address in sizeof(char)=1 bytes
- * works based on the size of the type!

Operations on pointers: =

```
int *p, i, *q;
i = 10;
p = &i;
q  = p;
```

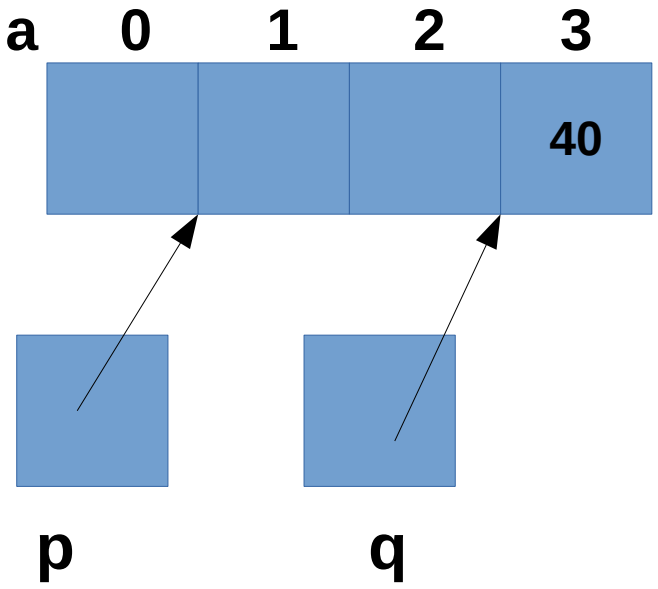




- **Pointers can be copied**
  - **Can arrays be copied?**
- **Thumb rule:**
  - **After pointer copy, both pointers point to same location**
- **Common Mistake**
  - **One pointer pointing to another**

Operations on pointers: +- int

```
int *p, a[4], *q;
p =  &a[1];
q = p + 2;
*q = 40;
```



- C allows adding or substracting an int from a pointer!
  - Weird, but true!
  - e.g. int *p, n; p + n;
- The result is a pointer of the same type
- The resultant pointer points *n type locations ahead(for +) or before (for -)*
- *A type location is equal to sizeof(type).*

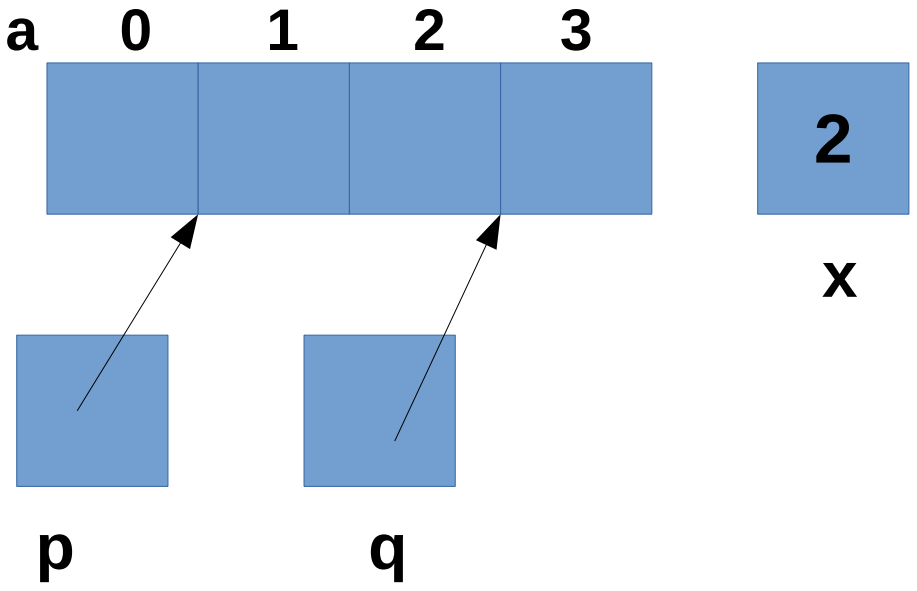
Problems: Draw diagrams for the code

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 2;
    p = &a[1];
    q = p + 1;
    p = q – b;
    *p = 30;
}
```

```
int main() {
    int *p, *q, a[3], b;
    a[0] = 10; b = 1;
    p = &a[3];
    q = p - 3;
    p = q + 1;
    *(q + 1) = 30;
    *(p – 1) = 20;
}
```

Operations on pointers: substracting two pointers

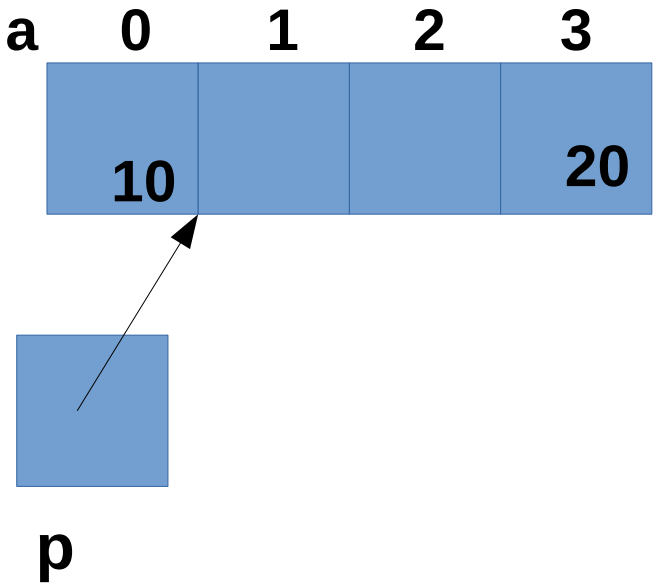
```
int *p, a[4], *q, x;
p =  &a[1];
q = p + 2;
x = q – p;
```



- **Two pointers of the *same type* can be substracted**
  - **Result type is int**
  - **Result value = no. Of elements of sizeof(type) between two pointers**
- **Logically derives from adding/substracting int to pointers**
  - **p = q + 2 ==> p – q = 2**

Operations on pointers: [ ]

```
int *p, a[4], *q;
p =  &a[1];
p[2] = 20;
p[-1] = 10;
```



- Interestingly, C allows [ ] notation to be applied to all pointers!
  - You must be knowing that [ ] is normally used for arrays
- p [i] means * (p + i)
  - P is a pointer and i is an integer (or i is a pointer and p is a pointer is also allowed)

A peculiar thing about arrays

- **Name of an array is equivalent to the address of (the zeroeth element) the array**

  int a[3];

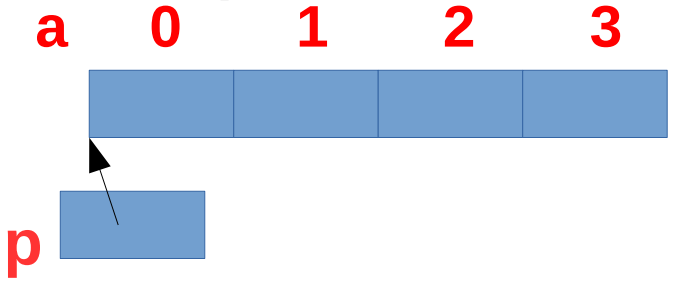
  Now

  a* means *&a[0]

- **Because it's an address, it can be stored in a pointer**

  int a[3], *p;

  p = a;



- **What do the following mean?**

  Int a[3] = {1, 2, 3}, *p;

  a + 1;

  ***(a + 1);**

  p = (a + 2)

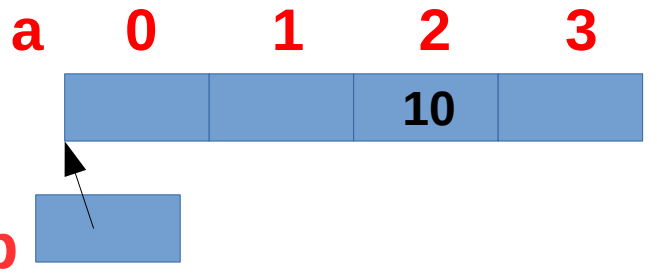
- **Exception: when passed to sizeof() operator, the name is not the address**

Pointer *as if it was an array*

- **Combine the concepts of**
  - **Pointer arithmetic (+- int)**
  - **[ ] notation for pointers**
  - **Array name as address of array**

```
int a[3], *p;
p = a;
p[2] = 10;
```

- **Here we are using p as if it was an array name**
  - **Possible only if p was pointing to array base**



Pointers != Arrays

- Array is a continuous collection of elements of one type.
- Array has name, the name also stands for the address of the array
- [ ] is allowed operation on arrays
- Array name can't be reassigned to another address

- Pointer is a variable that can store an address
- Pointers can be of various types
- [ ], = , +- int, substraction are operations on pointers
- Pointers can be reassigned to point to different addresses

Arrays as arguments to functions

```
char f(char *c) {
    return c[0];
}
int main() {
    char arr[16], x;
    x = f(arr);
}
```

- Here actual argument is ‘arr’
- Name of array is address of array, that is &arr[0]
  - Address of char
- So formal argument is char *

2-d array as argument to function

```
char f(char **c) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
You can answer by just applying the rules we learnt

2-d array as argument to function

```
char f(char (*c)[6]) {
return c[0][2];
}
int main() {
char arr[16][6], x;
x = f(arr);
}
```

```
char f(char c[][6]) {
 return c[0][2];
}
int main() {
 char arr[16][6], x;
 x = f(arr);
}
```

Which one is correct?
(The second one is same as on the earlier slide)
You can answer by just applying the rules we learnt

Dynamic Memory Allocation

Concept of (Binding) “Time”

- **Compile Time**
  - **When you are running commands like**

    cc program.c -o program
- **Load time**
  - **After you type commands like**

    ./program

    Before the main() starts running
- **Program Run Time**
  - **After you type commands like**

    ./program

    When the main() of the program starts running, till it exits
- **Function Call Time**
  - **Interval between the call of a function and before the called function starts running. Part of “Run Time”.**

Lifetime of variables and Memory Allocation

- **Global Variables, Static Variables (g, t, and s here)**
  - Allocated Memory at *load time*
  - They are alive (available) till the program exits
- **Local Variables, Formal Arguments (i, j, k in main; a, b, x, y in f)**
  - Allocated Memory on *function call*
  - They are alive (available) as long as function is running
- **Dynamically allocated memory (Run time allocation)**
  - Allocated on explicit call to functions like *malloc()*
  - Alive as long as functions like *free()* are not called on the memory

```
int g;
static int t = 20;
int f(int a, int b) {
    int x, y;
    static int s = 10;
    x = a + b + 5 + g + s;
    return x;
}
int main() {
    int i, j, k, *p;
    g = 10;
    i = 20; j = 30;
    p = malloc(8);
    k = f(i, j);
}
```

Man pages

Understanding library functions:

Red the manual pages, using ‘man’ command

```
> man sqrt
> man 3 printf
> man 3 open
# see section 3 of man pages for  C library functions
```

malloc()

- malloc() is a standard C library function for allocating memory dynamically (at run time)
- #include <stdlib.h> for malloc()
- Function prototype
  - Run “man malloc” to see it

    ```
    void *malloc(size_t size);
    ```

  - size_t is a typedef in stdlib.h
    - size_t is unsigned long
  - Reads a number, allocates those many bytes and returns the address of allocated memory (zero'th byte)
- Additional info: malloc() gets the memory from the OS and then gives to your program

void *

- **A void pointer is a typeless pointer;**
  - **Pure address**
  - **No type --> No “size” information about the type**
- **You can declare a void pointer**

  void *p, *q
- **Void pointer can store *any* address**

  void *p; int a; p = &a; char c; p = &c;
- **Void pointers can be copied**

  void *p, *q; int a; p = &a; q = p;
  - **q also stores address of a now**
- **The Dereferencing operator has no meaning when applied to a void pointer**

  void *p; int a; p = &a;
  - **What does *p mean now? * Needs “size” of the type for its work. Void pointers have no type and so no size information associated with it.**
  - **Note: [ ] is also dereferencing**

malloc()

- **malloc() returns a “void *”**
  - **Returns a pure address**
  - **This address can be stored in a “void *” variable**
  - **This address can also be stored in any pointer variable**
- **Suppose we do**
  - **void *p; p = malloc(8);**
  - **Now what meaningful operations can we do with the malloced memory? --> only copy!**
  - **So normally return value of malloc is stored using some typed pointer**

malloc()

```
int *p;
p = malloc(8)
```

- This code allocates 8 bytes and then pointer p will point to the malloced memory
- This code can result in a “warning” because we are converting “void *” to a “int *” with '='

malloc()

```
int *p;
p  = (int *) malloc(8);
```

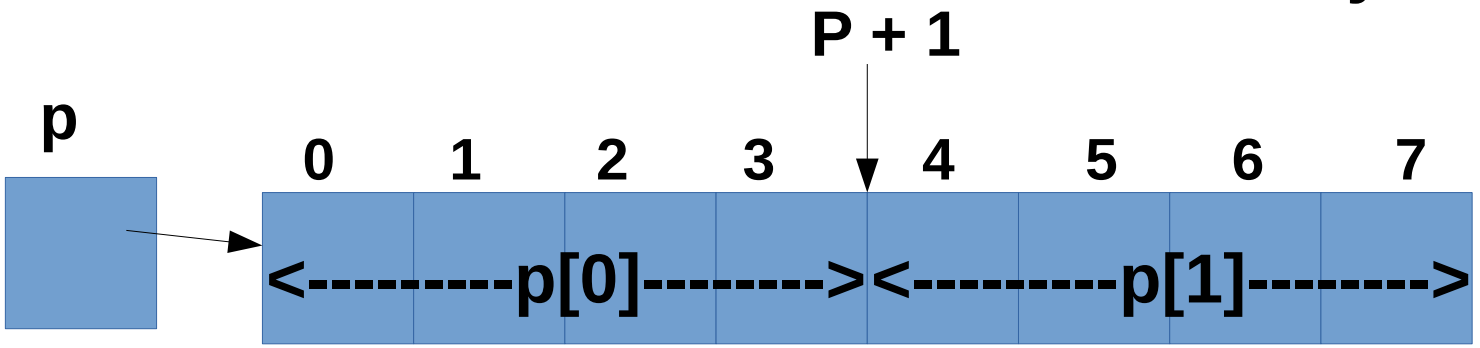
- This code does away with the warning as we are converting the “void *” into “int *” using explicit type casting
- Suppose size of integer was 4 bytes, then what does this code mean?

malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !



malloc()

```
int *p;
p  = (int *) malloc(8);
```

- p points to 8 byte location
- However now, *p means deferencing in “4 bytes”
  - as size of int is 4 bytes

- p[0] means *(p + 0) that is *p
- p[1] means *(p + 1) where (p + 1) is a pointer 4 bytes ahead of p
  - Using this trick we are treating the 8 bytes as if it was a 2 integer array !

Malloc(): Allocating arrays dynamically

```
int *p;
p  = (int *) malloc(
        sizeof(int) * 4);
```

- We can allocate arrays of any type dynamically using malloc()
- Use of sizeof(int) here makes sure that the code is *portable*, appropriately sized array will be allocated irrespective of size of integer
  - Code on earlier slide assumes 4 byte integer
- This code allocates array of 4 integers
  - Can be acessed as p[0], p[1], p[2] and p[3]

malloc(): Allocating array of structures

```
typedef struct test {
        int a, b;
        double g;
}test;
test *p;
p  = (test *) malloc(
        sizeof(test) * 4);
```

- This code allocates an array of 4 structures
- p points to the array of structures
- p[0] is the 0^{th} structure, p[1] is the 1^{st} structure ...
- p[0].a, p[1].g is the way to access the inner elements of structures

Homework

- **Write code using malloc() to**
  - **Allocate an array of 10 doubles**
  - **Read n from user and allocate an array of n shorts**
- **What does the following code mean?**

  int *p; p = malloc(9); // suppose sizeof(int) is 4

  Note: No need to write this sort of code in this course!
- **What does the following code mean?**

  int *ip; Char *p; p = malloc(8); ip = p; p[0] = 10;

  Note: again, no need to write code like this in this course!

free()

- free() will give the malloced memory back
- malloc() and free() work together to manage what is called as “heap memory” which the memory management library has obtained from the OS
- Usage

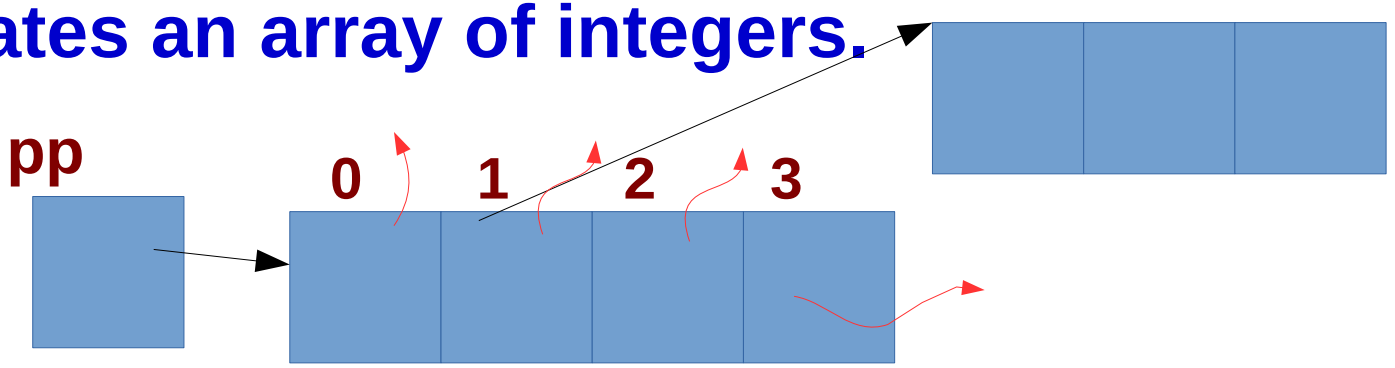
  ```
  void free(void *p);
  ```

- free() must be given an address which was returned by malloc()
- Rule: Every malloc() must have a corresponding free()

Pointer to Pointer

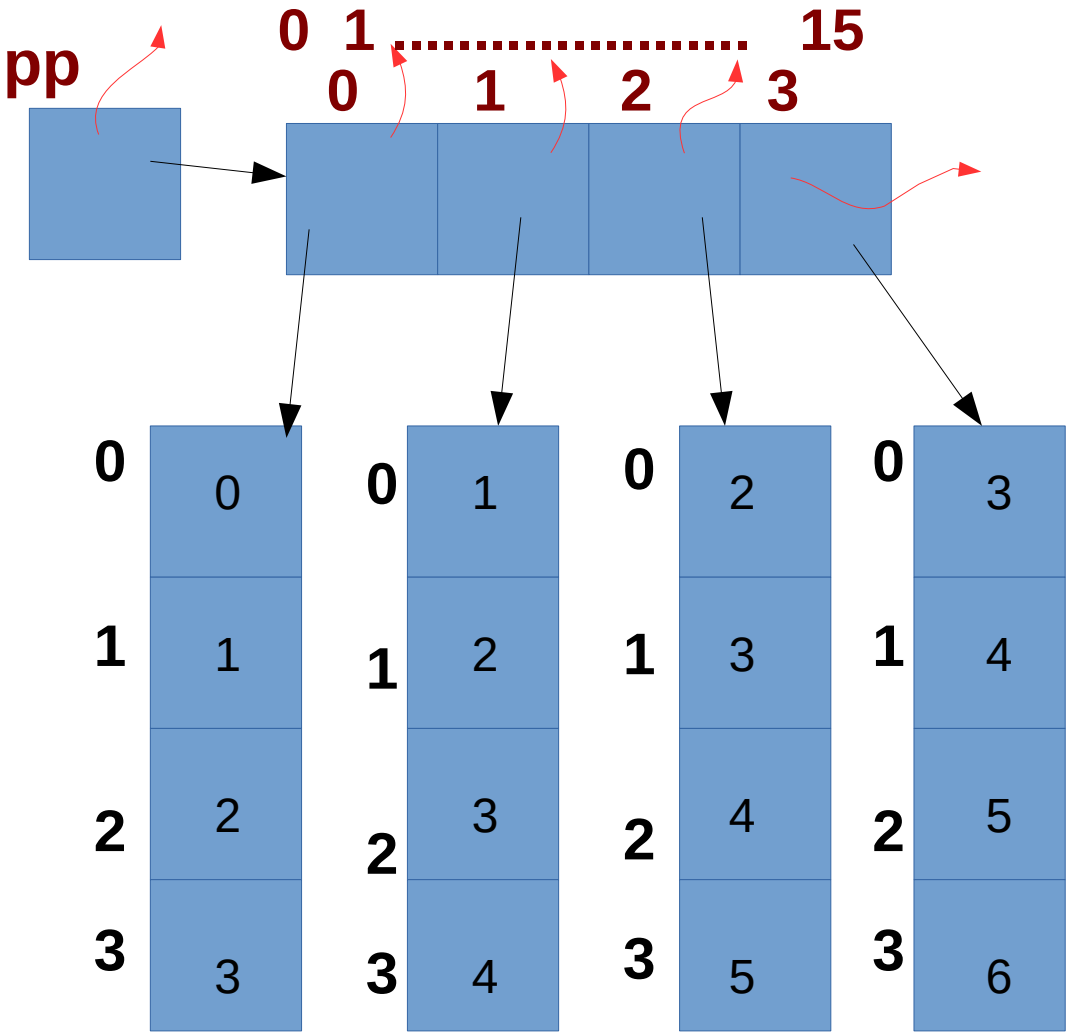
```
int **pp, *p, p;
pp = (int **) malloc(sizeof(int *) * 4);
pp[1] = (int *)malloc(sizeof(int) * 3);
```

- A pointer to pointer, is essentially a pointer!
  - Can store an address, has a type
  - The type is “pointer to pointer” so all * or [ ] operations work with sizeof(pointer) which is typically 4 bytes
- The code above allocates an array of pointers, and then allocates an array of integers.
  -



2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```



2-d array with pointer to pointer

```
#include <stdlib.h>
#define N 4
int main() {
    int **pp, i, j;
    pp = (int **) malloc
                (sizeof(int *) * N) ;
    for(i = 0; i < N; i++)
        pp[i] = (int *)
                malloc(sizeof(int) * N);
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            pp[i][j] = i + j;
    return 0;
}
```

- How does pp[i][j] work here?
- pp[i] is *(pp + i) which is the pointer at the i'th location in the array of pointers allocated
  - Here * works with *size* = *sizeof a pointer*
- pp[i][j] is *(pp[i] + j)
  - Since pp[i] is a integer pointer, so here * works with sizeof(int)
  - pp[i] is the address of array (indicated in red), so pp[i] + j is pointer to the j'th element of that array

Self Referential Pointers in Structures

Self Referential Pointers

- **“Self Referential Pointer” is a kind of a misnomer**
- **C allows structures like this**

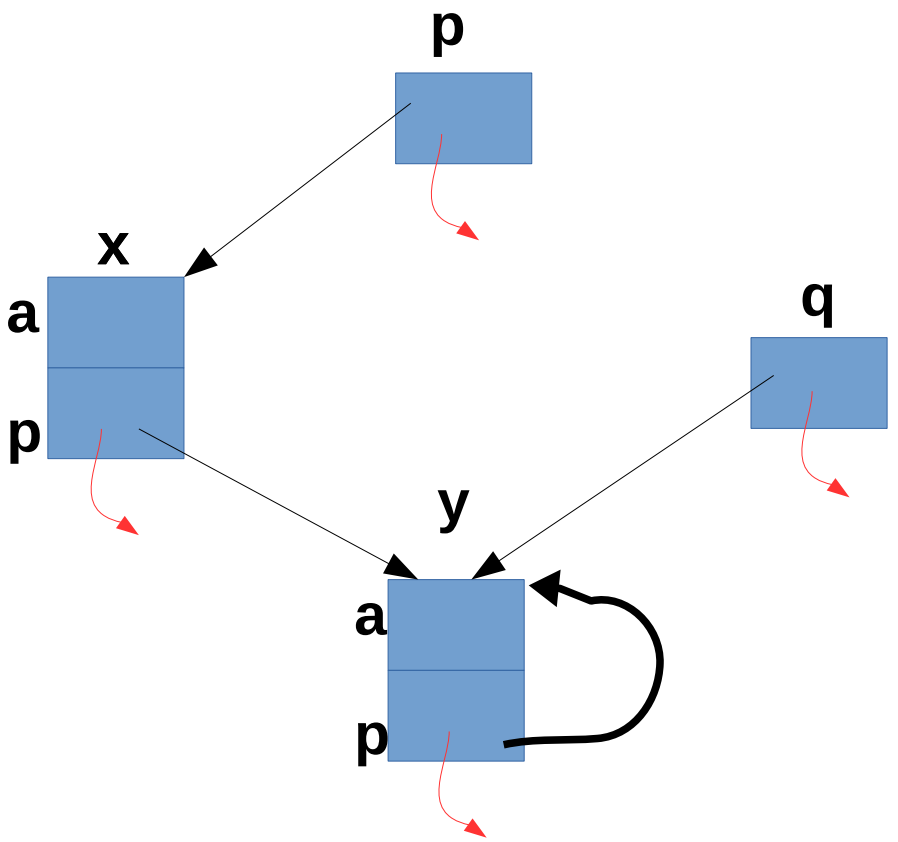
```
struct test {
   int a;
   struct test *p;
};
```

- **The pointer p, can point to any variable of the type “struct test” (or be NULL)**

Self Referential Pointers

- **Consider following code**

```
typedef struct test {
    int a;
    struct test *p;
}test;
test  x, y, *p, *q;
p = &x;
x.p = &y;
q = &y;
y.p = &y;
```

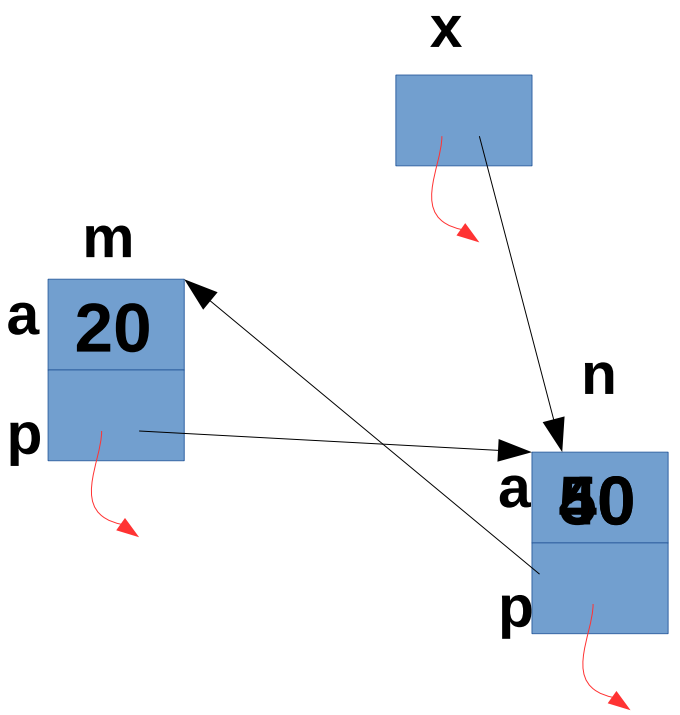


Self referential structures allow us to create a variety of “linked” *structures of data*

-> notation

```
typedef struct test {
    int a;
    struct test *p
}test;
test  m, n, *x;
m.a = 20;
x = &n;
(*x).a = 40;
x->a = 50;
x->p = &m;
x->p->p = x;
```

- (*x) is the entire structure to which x points
  - (*x).a is the variable 'a' in that structure
- x->a is another notation for (*x).a



2 Common Problems involving Pointers:

(Dereferencing) Dangling Pointers
And
Garbage Memory

NULL

- NULL is a symbolic constant defined in stdio.h
  - #define NULL 0
- This is a special pointer value, defined by C language
  - The number 0 is not the same as the address 0 ! They are different types !
- It is guaranteed that a pointer is NOT NULL, unless programmer sets it to NULL

A side note on NULL

- **NULL is not the number 0**
- **NULL is not necessarily the address 0**
- **NULL is just a special value for pointers told to us by C language.**
  - **Very often we need special values for a certain type**
  - **E.g. the value '\0' for a character is universally taken to be a special value indicating end of a character sequence in an array**
  - **INT_MAX , INT_MIN are values #defined in limits.h for integers**
  - **These type of values are used in programs to indicate either an *unused variable* or *empty variable* or *error value* on that variable**
- **Int i = NULL; char c = NULL; works**
  - **Why do you want to do it? Instead of saying int i = 0; char c = 0;**

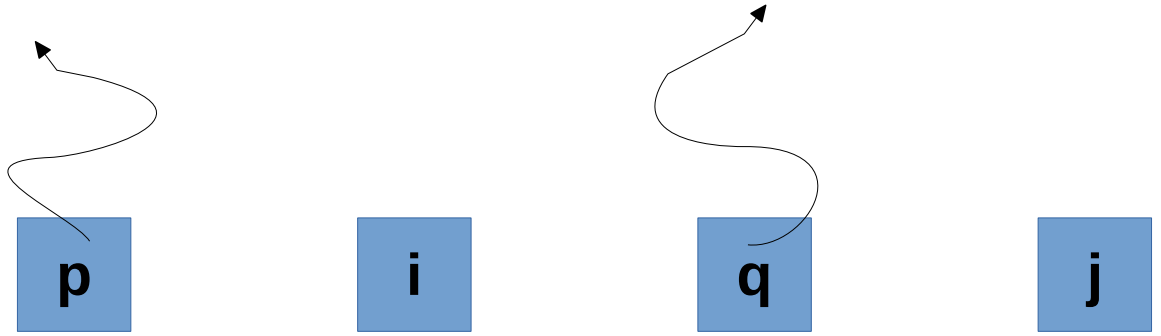
Dangling Pointers

- Total 3 Possibilities for a pointer value
  - Points to memory owned by program
    - Local variables, Global Variables, Malloced Memory
  - = NULL
  - Dangling
    - Some texts differentiate between ‘wild’ (uninitialized) and ‘dangling’ pointers
- Dereferencing (that is *) a dangling (or NULL) pointer is NOT to be done
  - Even if you find that dereferencing a dangling pointer “*does not create problem”* in your code, it is still not to be done!
  - Result: The OS punishes your code by terminating it, and saying “Segmentation Fault”

Dangling Pointers

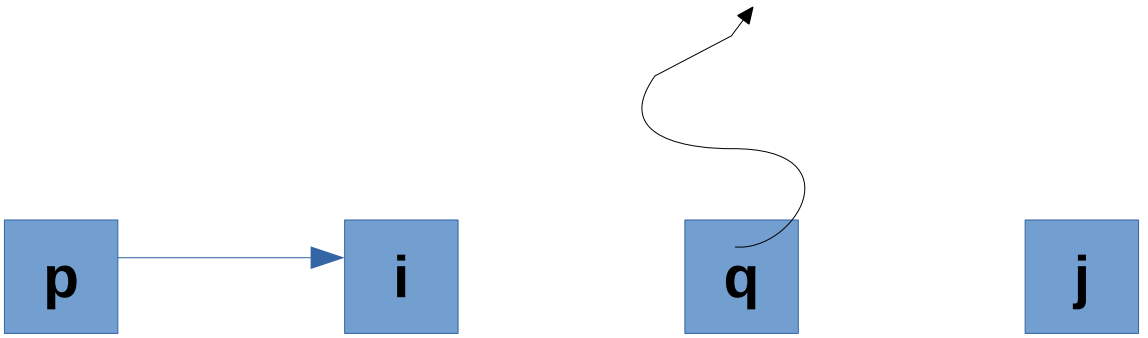
- A pointer is dangling when declared & not initialized
- A pointer becomes non dangling, when assigned to a “good” memory location like local variables, global variables, malloc-ed memory
- A pointer can become dangling again due to
  - Mistakes in pointer arithmetic
  - Mistakes in manipulation of dynamically allocated memory, e.g. Linked list pointers
  - On deallocation of malloced memory using free()
  - etc

```
int main() {
        int *p, i, j, *q;// p,q are dangling

}
```



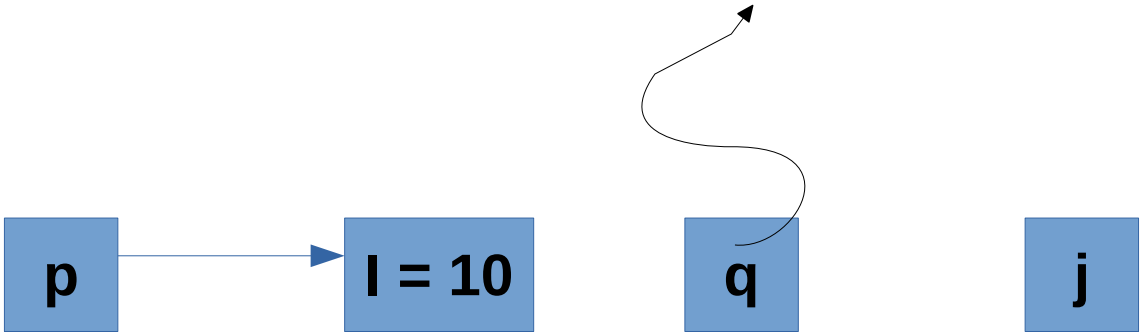
Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling

}
```



Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10

}
```

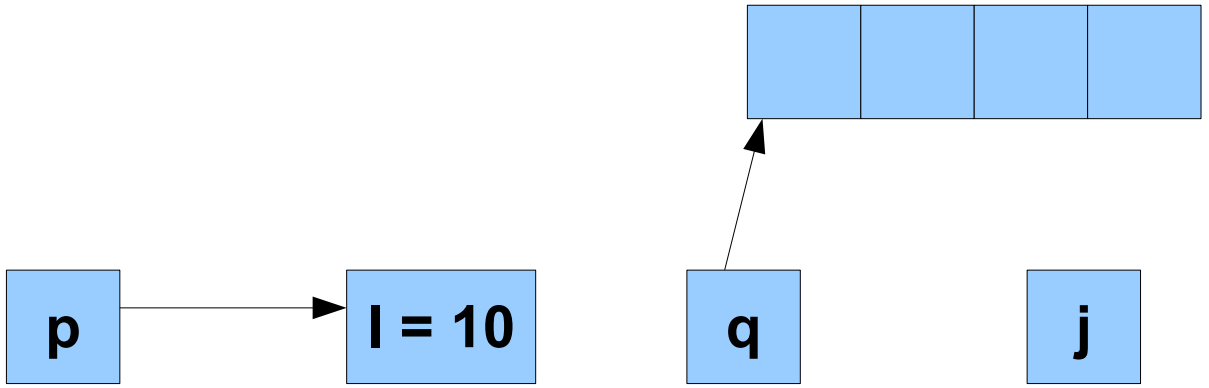


Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling

}
```

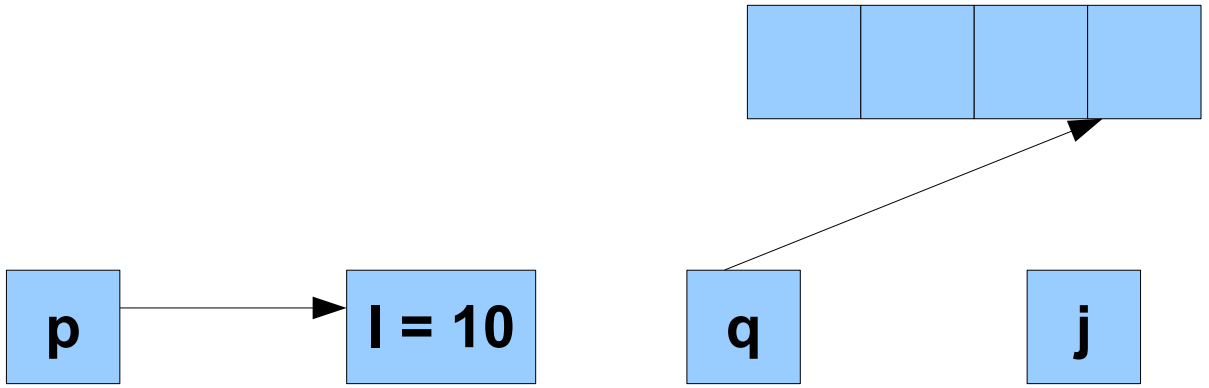
Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                        // q not dangling

}
```



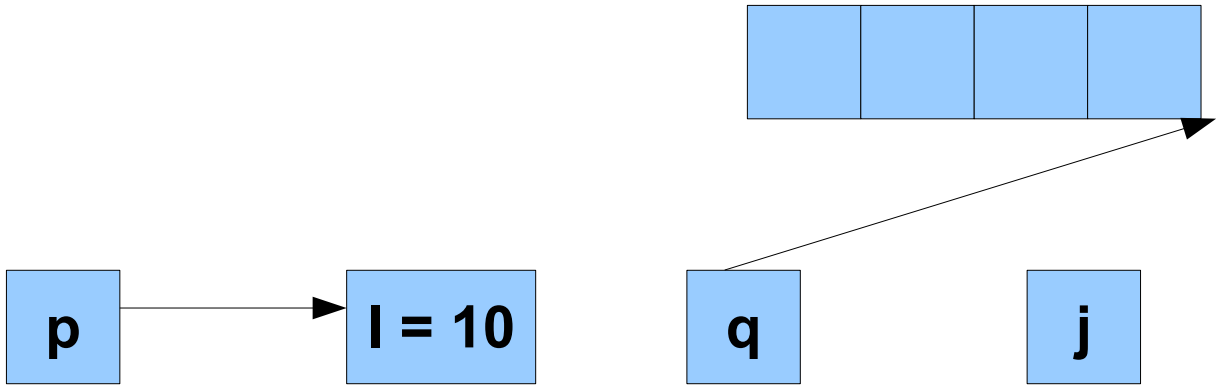
Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                              // q not dangling
        q = q + 3; // q not dangling
       }
```



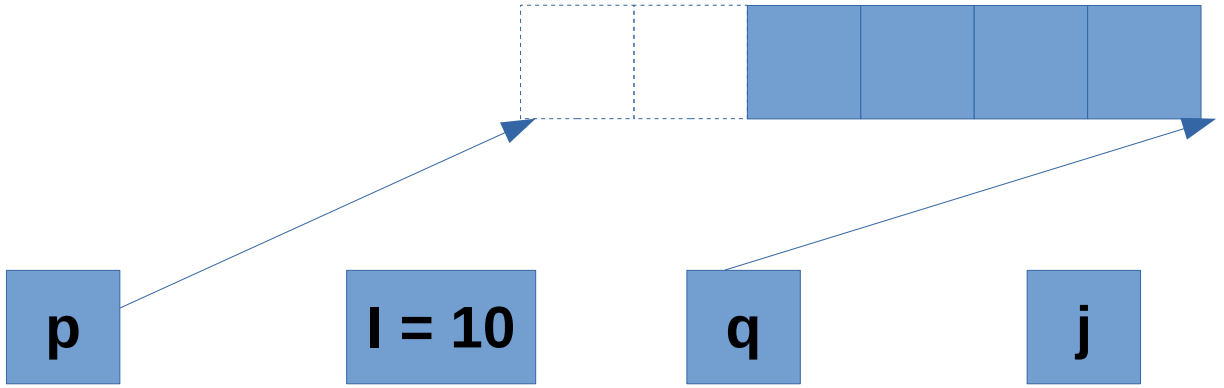
Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now

}
```



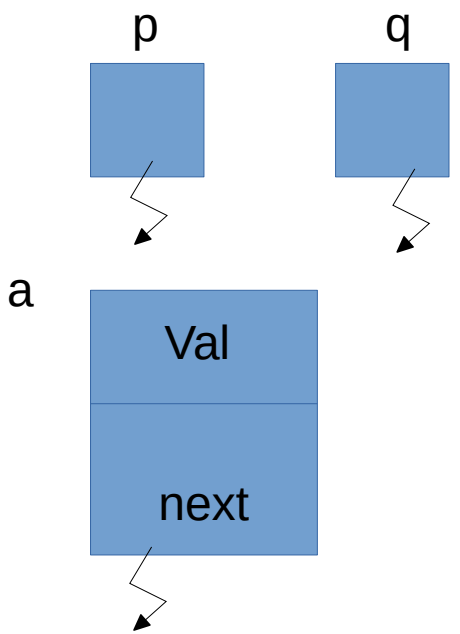
Dangling Pointers Example: 1

```
int main() {
        int *p, i, j, *q;// p,q are dangling
        p = &i; // p not dangling
        i = 10; // *p = 10
        q = &j; // q not dangling
        q = (int *)malloc(sizeof(int) * 4);
                                // q not dangling
        q = q + 3; // q not dangling
        q = q + 1; // q IS dangling now
        p = q - 6; // p IS also dangling
}
```



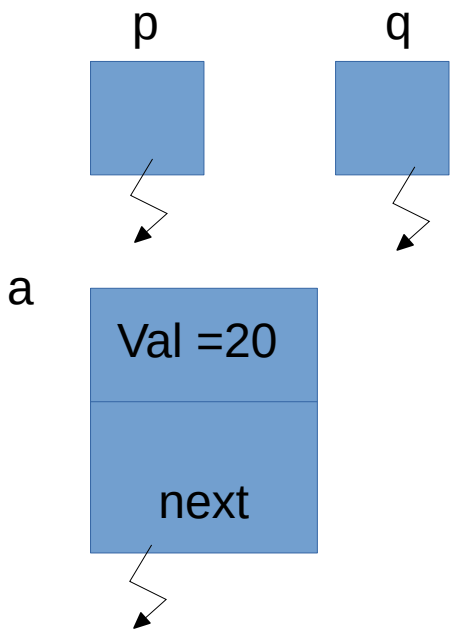
Dangling Pointers Example: 1

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
}
```



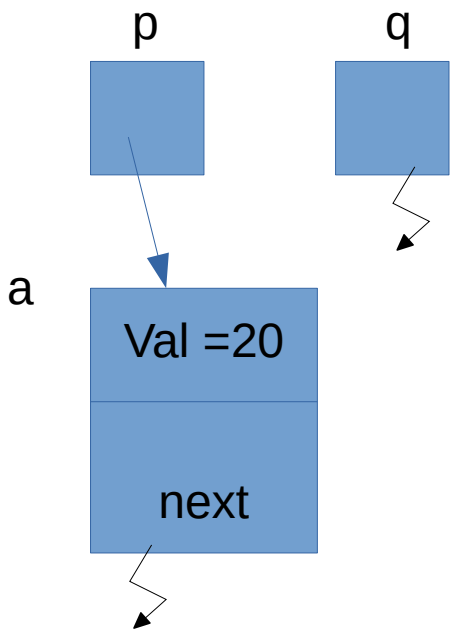
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
}
```



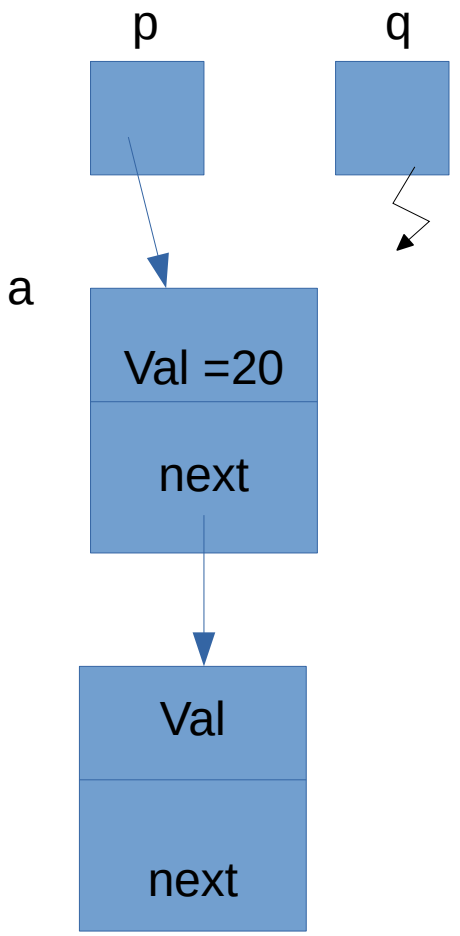
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;

}
```



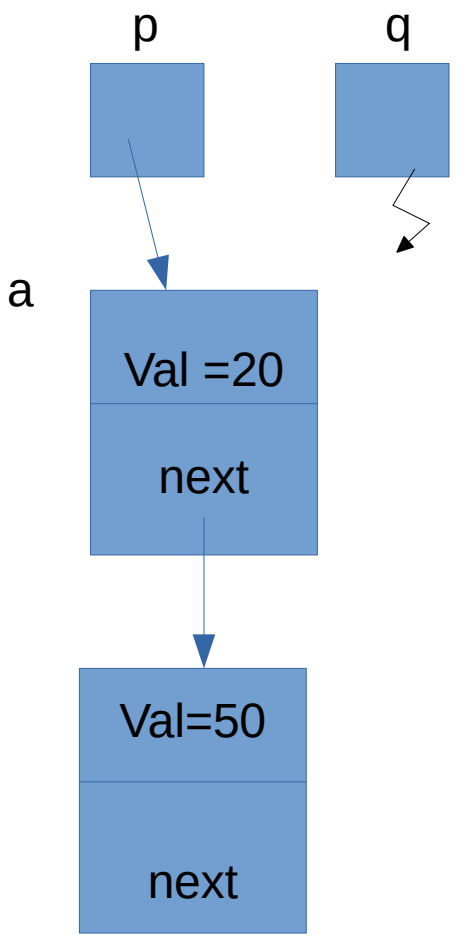
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));

}
```



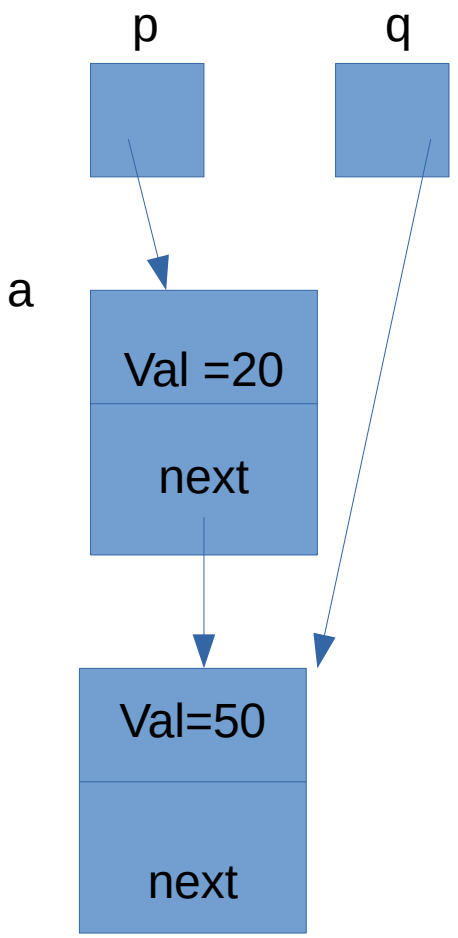
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;

}
```



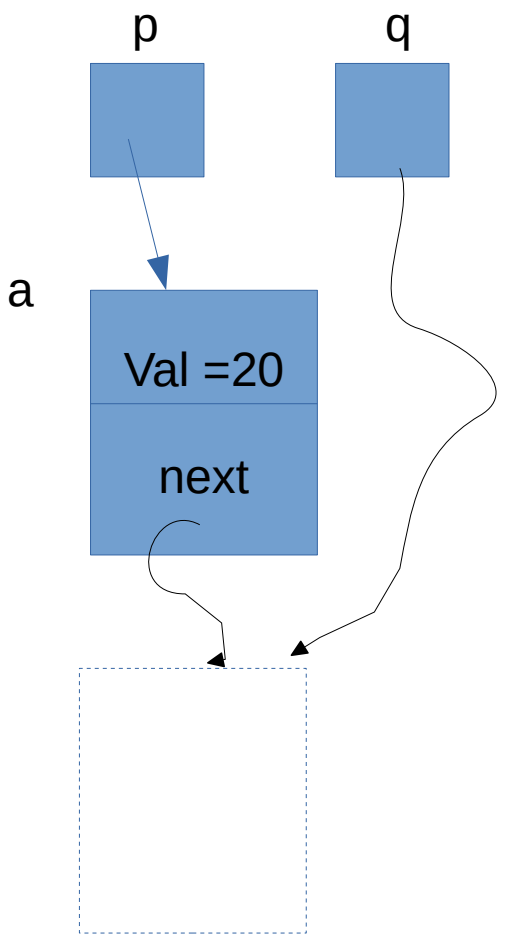
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;

}
```



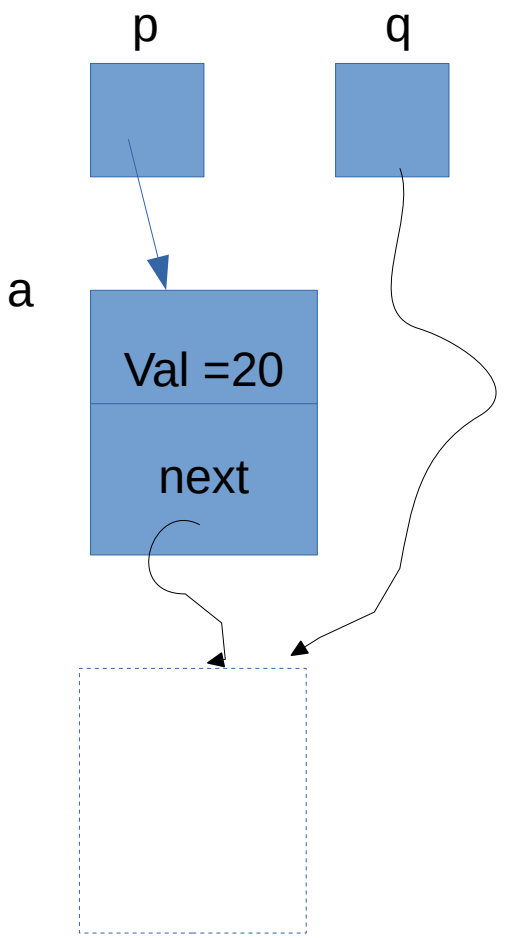
Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);

}
```



Dangling Pointers Example: 2

```
typedef struct node {
        int val;
        struct node *next;
}node;
int main() {
        node *p, *q;
        node a;
        a.val = 20;
        p = &a;
        a.next = (node *)malloc(sizeof(node));
        a.next->val = 50;
        q = a.next;
        free(a.next);
        q->val = 100; // segfault

}
```



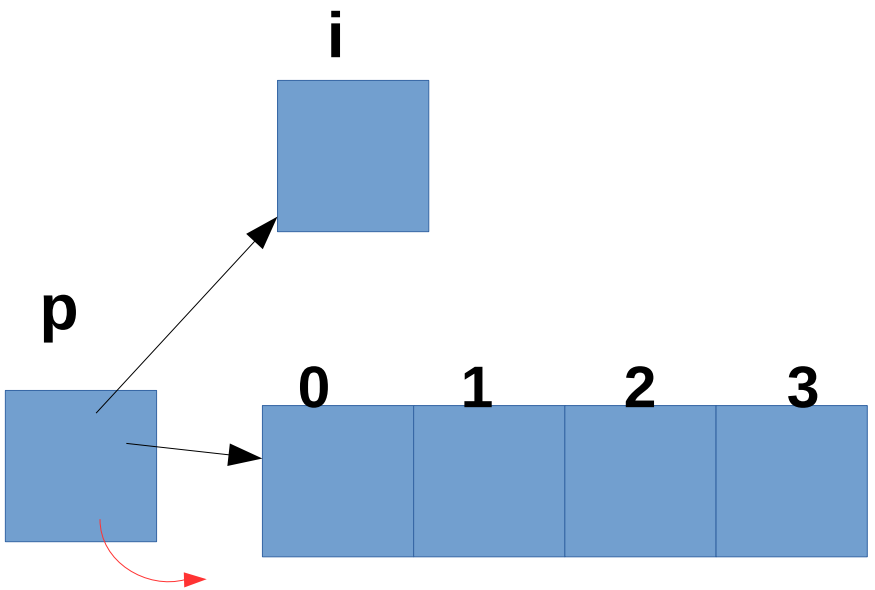
Dangling Pointers Example: 2

Remember

- Dereferencing a dangling pointer is a cause of segmentation fault
- Dereferencing can occur using * or [ ]
- Having a dangling pointer does not cause segmentation fault
  - Dereferencing it causes
  - Having dangling pointers is not wrong, but why have them?

Garbage Memory

```
int *p, i;
p = malloc
      (sizeof(int) * 4);
p = &i;
```



- p was pointing to malloced memory
- In the end p points to i
- How to access the malloced memory now?
  - No way! It's Lost!
  - It became “garbage memory”
- Problems like this result in memory waste
  - Also called as Memory leak
- Solution
  - Keep another point pointing to malloced memory

    ```
    p = malloc(sizeof(int) * 4);
    q  = p;
    p = &i;
    ```

  - Malloced memory is available through pointer q

Segmentation Fault

What is segmentation fault?

- A program accessing an “illegal” memory location
  - Is punished by the operating system for doing so
  - The program gets “terminated” (killed)
  - “segmentation” comes from “memory management” concepts in operating systems
- It is ALWAYS a fault of the programmer!
- Beware
  - Bad compilers may generate machine code which hide some memory violations
  - If a programmer does illegal memory access, segmentation fault may not occur sometimes!
    - OS may “forgive” your program :-;

Some Typical Reasons for Seg-fault

Deferencing Dangling Pointers

Array Index Violation

Incorrect Function Arguments

Dangling Pointer Dereference: Some examples

```
int *p, i;
*p = 20;
```

```
int *p, i;
p = malloc(
        sizeof(int)*2);
free(p);
p[2] = 20;
```

```
    int *p, i;
    p = &i;
    p[1] = 20;
```

```
int *f(int *p) {
    int x = *p + 2;
    return &x;
}
int main() {
    int i, *q;
    scanf("%d", &i);
    q = f(&i);
     *q = 20;
}
```

Don't return address of a local variable! The variable disappears after function call, and the returned address is a dangling pointer.

Array Index Violation

Valid indices for an array of size n: 0 .. n-1

Accessing any index <=-1 or >=n *may* result in seg-fault (it may not result in some cases, but it is STILL WRONG to do it!)

Try this code:

At what value of i does the code seg-fault? Try 2-3 times.

```
#include <stdio.h>
int main() {
    int a[16], i = 0;
    while(1) {
        a[i] = i;
        printf("%d\n", a[i]);
        i++;
    }
}
```

Reason: OS may allocate more than 16-integer size memory for the program. So the seg fault *may not* occur at index 16 or slightely higher indices also. Rule: wrong to access index >=n

Functions: Pointer Arguments

- Rule: When you want a function to change ACTUAL arguments
  - Function takes pointer arguments
  - Call passes address of “actual argument”
- Example: Swap function (correct code)

```
void swap(int *a, int *b ) {
        int temp;
        temp = *a;
        *a = * b;
        *b = temp;
}
int main() {
    int m = 20, n = 30;
    swap(&m, &n);
```

Incorrect Pointer Aruments - Segfault

```
int i;

scanf("%d", i);
```

Scanf tries to do something like

*i = value // segfault here

Segfault occurs in scanf, although the reason is call to scanf

- Note that this is basically a dangling pointer dereference which took place inside scanf()

Guidelines to avoid segfaults

- Always initialize pointers to NULL
  - This will not avoid segfaults, but may lead to early detection of a problem
- While accessing arrays, make sure that array index is valid
- Never return address of local variable of a function
- DO NOT IGNORE compiler's warnings
  - Compilers often warn about potential dangling references, type conversions which have a potential for segfaults
  - Make sure that you rewrite code to remove all warnings
  - Use “-Wall” option with gcc. > cc -Wall program.c -o program

Let’s draw diagrams for some programs using pointers, malloc, free, ...

Introduction to Linux Desktop
and
Command Line
11 Jan 2022

Abhijit A. M.
abhijit.comp@coep.ac.in

Why GNU/Linux ?

Why GNU/Linux ?

1. Programmer's Paradise : most versatile, vast, all pervasive programming environment

2. Free Software (or Open Source?) : Free as in freedom. *Freely* Use, copy, modify, redistribute.

3. Highly Productive : Do more in less time

4. Better quality, more secure, very few crashes

Why Command Line ?

1. Not for everyone ! Absolutely !

2. Those who do it are way more productive than others

3. Makes you think !

4. Portable. Learn once, use everywhere on all Unixes, Linuxes, etc.

Few Key Concepts

- ***Files*** **don't open themselves**
  - **Always some application/program open()s a file.**
- **Files don't display themselves**
  - **A file is displayed by the program which opens it. Each program has it's own way of handling ifles**

Few Key Concepts

- **Programs don't run themselves**
  - **You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program**
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - **and programs open files**

Path names

- Tree like directory structure
- Root directory called /
- A complete path name for a file
  - /home/student/a.c
- Relative path names

  concept: every running program has a *current* working directory

  . current directory

  .. parent directory

  ./Desktop/xyz/../p.c

A command

- Name of an executable file
  - For example: 'ls' is actually “/bin/ls”
- Command takes arguments
  - E.g. ls /tmp/
- Command takes options
  - E.g. ls -a

A command

- Command can take both arguments and options
  - E.g. ls -a /tmp/
- Options and arguments are basically argv[] of the main() of that program

Basic Navigation Commands

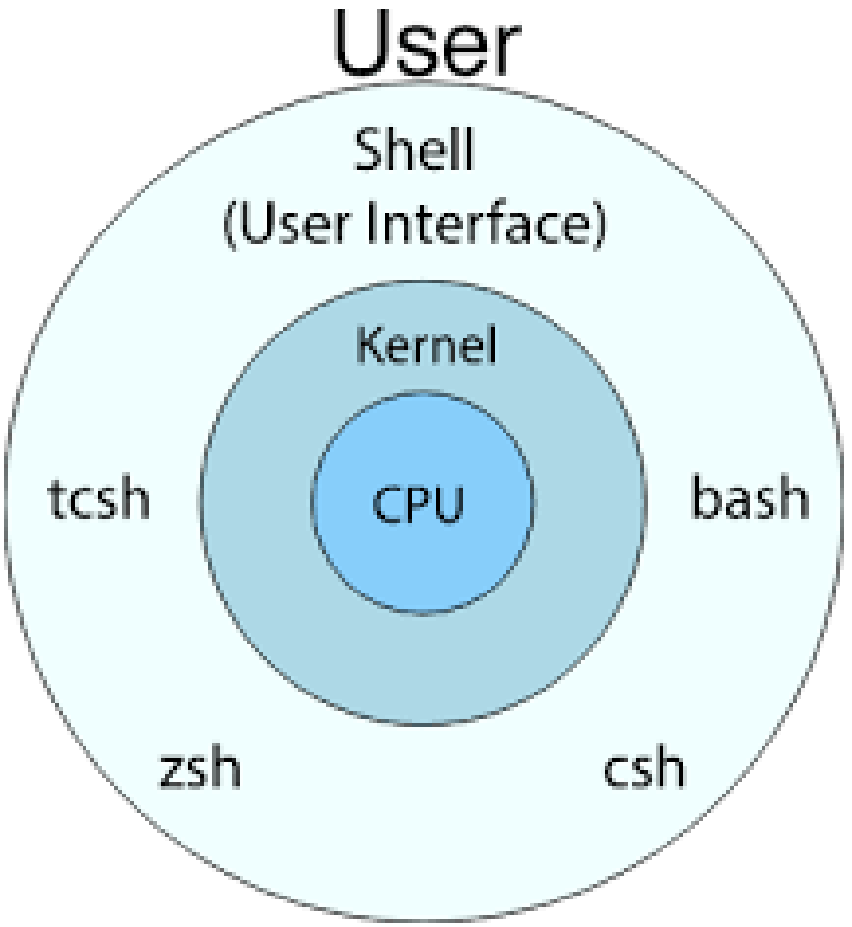
- pwd
- ls
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a

    \ls -F
- cd
  - cd /tmp/
  - cd
  - cd /home/student/Desktop
- notation: ~
  - cd ~
  - cd ~/Desktop
  - ls ~/Desktop

Map these commands to navigation using a graphical file browser

The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the *Applications***
- **Talks with Users and Applications and does some talk with OS**



Not a very accurate diagram !

http://code.snipcademy.com/tutorials/linux-command-line/environment/

The Shell

Shell waits for user's input

Requests the OS to run a program which the user has asked to run

Again waits for user's input

GUI is a Shell !

Let's Understand fork() and exec()

```
#include <unistd.h>
int main() {
        fork();
        printf("hi\n");
        return 0;
}
```

```
#include <unistd.h>
int main() {
        printf("hi\n");
        execl("/bin/ls", "ls",
NULL);
        printf("bye\n");
        return 0;
}
```

A simple shell

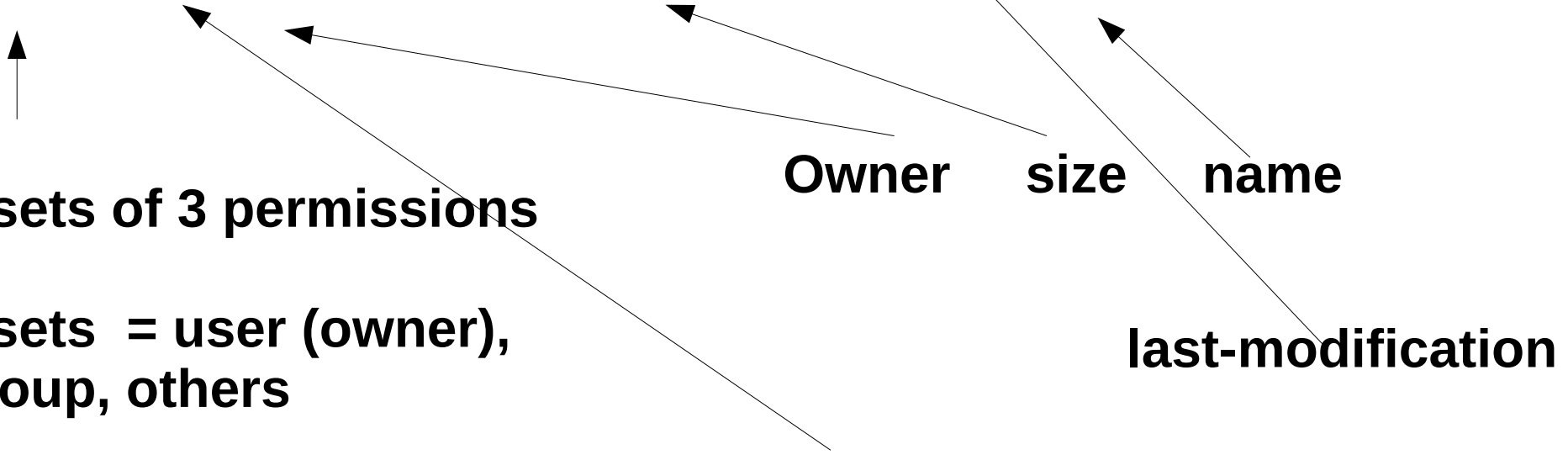
```
#include <stdio.h>
#include <unistd.h>
int main() {
        char string[128];
        int pid;
        while(1) {
                printf("prompt>");
                scanf("%s", string);
                pid = fork();
                if(pid == 0) {
                        execl(string, string, NULL);
                } else {
                        wait(0);
                }
        }
```

File Permissions on Linux

- Two types of users
  - root and non-root
  - Users can grouped into 'groups'
- 3 sets of 3 permission
  - Octal notation
  - Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- chmod command uses these notations
-

File Permissions on Linux

```
-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01_linux_basics.ppt
-rw-r--r-- 1 abhijit abhijit  341736 May 17 10:39 Debian Family Tree.svg
drwxr-xr-x 2 abhijit abhijit    4096 May 17 11:16 fork-exec
-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp
```



3 sets of 3 permissions

3 sets = user (owner), group, others

3 permissions = read, write, execute

hard link count

Owner

size

name

last-modification

File Permissions on Linux

- r on a file : can read the file
  - open(.... O_RDONLY) works
- w on a file: can modify the file
- x on a file: can ask the os to run the file as an executable program
- r on a directory: can do 'ls'
- w on a directory: can add/remove files from that directory (even without 'r'!)
- x on a directory: can 'cd' to that directory

Access rights examples

- **-rw-r--r--**
  Readable and writable for file owner, only readable for others
- **-rw-r-----**
  Readable and writable for file owner, only readable for users belonging to the file group.
- **drwx------**
  Directory only accessible by its owner
- **-------r-x**
  File executable by others but neither by your friends nor by yourself. Nice protections for a trap...

Man Pages

- Manpage
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir
  - 2 System calls
    - int mkdir(const char *, …);
  - 3 Library calls
    - int printf(const char *, …);
  - 4 Device drivers and network protocols
    - /dev/tty
  - 5 Standard file formats
    - /etc/hosts
  - 6 Games and demos
    - /usr/games/fortune
  - 7 Misc. files and docs
    - man 7 locale
  - 8 System admin. Cmds
    - /sbin/reboot

GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
/dev/	Files representing devices
	/dev/hda: first IDE hard disk
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

GNU / Linux filesystem structure

/lost+found	Corrupt files the system tried to recover
/media	Mount points for removable media: **/media/usbdisk, /media/cdrom**
/mnt/	Mount points for temporarily mounted filesystems
/opt/	Specific tools installed by the sysadmin **/usr/local/** often used instead
/proc/	Access to system information **/proc/cpuinfo, /proc/version ...**
/root/	root user home directory
/sbin/	Administrator-only commands
/sys/	System and device controls (cpu frequency, device power, etc.)

GNU / Linux filesystem structure

/tmp/ Temporary files

/usr/ Regular user tools (not essential to the system)

/usr/bin/, **/usr/lib/**, **/usr/sbin**...

/usr/local/ Specific software installed by the sysadmin

(often preferred to **/opt/**)

/var/ Data used by the system or system servers

/var/log/, **/var/spool/mail** (incoming mail), **/var/spool/lpd** (print jobs)...

Files: cut, copy, paste, remove,

- cat <filenames>
  - cat /etc/passwd
  - cat fork.c
  - cat <filename1> <filename2>
- cp <source> <target>
  - cp a.c b.c
  - cp a.c /tmp/
  - cp a.c /tmp/b.c
  - cp -r ./folder1 /tmp/
  - cp -r ./folder1 /tmp/folder2
- mv <source> <target>
  - mv a.c b.c
  - mv a.c /tmp/
  - mv a.c /tmp/b.c
- rm <filename>
  - rm a.c
  - rm a.c b.c c.c
  - rm -r /tmp/a
- mkdir
  - mkdir /tmp/a /tmp/b
- rmdir
  - rmdir /tmp/a /tmp/b

Useful Commands

- **echo**
  - echo hi
  - echo hi there
  - echo “hi there”
  - j=5; echo $j
- **sort**
  - sort
  - sort < /etc/passwd
- **firefox**
- **libreoffice**
- **grep**
  - grep bash /etc/passwd
  - grep -i display /etc/passwd
  - egrep -i 'a|b' /etc/passwd
- **less <filename>**
- **head <filename>**
  - head -5 <filename>
  - tail -10 <filename>

Useful Commands

- **alias**

  alias ll='ls -l'

- **tar**

  tar cvf folder.tar folder

- **gzip**

  gzip a.c

- **touch**

  touch xy.txt

  touch a.c

- **strings**

  strings a.out

- **adduser**

  sudo adduser test

- **su**

  su administrator

Useful Commands

- **df**

  df -h

- **du**

  du -hs .

- **bc**
- **time**
- **date**
- **diff**
- **wc**

Network Related Commands

- ifconfig
- ssh
- scp
- telnet
- ping
- w
- last
- whoami

Unix job control

- **Start a background process:**
  - gedit a.c &
  - gedit
    hit ctrl-z
    bg
- **Where did it go?**
  - jobs
  - ps
- **Terminate the job: kill it**
  - kill *%jobid*
  - kill *pid*
- **Bring it back into the foreground**
  - fg %1

Shell Wildcards

- ? (question mark)
  - Any ene character
  - ls a?c
  - ls ??c
- *
  - any number of characters
  - ls *
  - ls d*
  - echo *
- [ ]
  - Matches a range
  - ls a[1-3].c
- {}
  - ls pic[1-3].{txt,jpg}

Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in */etc*
- */etc/*passwd and */etc/*shadow
  - Text files containing user accounts
- */etc/*resolv.conf
  - DNS configuration
- */etc/network/interfaces*
  - *Network configuration*
- */etc/*hosts
  - Local database of Hostname-IP mappings
- */etc/*apache2/apache2.conf
  - Apache webserver configuration

~/.bashrc file

- **~/.bashrc**

  Shell script read each time a bash shell is started
- **You can use this file to define**
  - **Your default environment variables (PATH, EDITOR...).**
  - **Your aliases.**
  - **Your prompt (see the bash manual for details).**
  - **A greeting message.**
- **Also ~/.bash_history**

Special devices (1)

Device files with a special behavior or contents

- **/dev/null**
  The data sink! Discards all data written to this file.
  Useful to get rid of unwanted output, typically log information:

  ```
  mplayer black_adder_4th.avi &> /dev/null
  ```

- **/dev/zero**

  Reads from this file always return **\0** characters
  Useful to create a file filled with zeros:

  ```
  dd if=/dev/zero of=disk.img bs=1k count=2048
  ```

See **man null** or **man zero** for details

Special devices (2)

- **/dev/random**
  Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness ("entropy").
  Reads can be blocked until enough entropy is gathered.

- **/dev/urandom**
  For programs for which pseudo random numbers are fine.
  Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See **man random** for details.

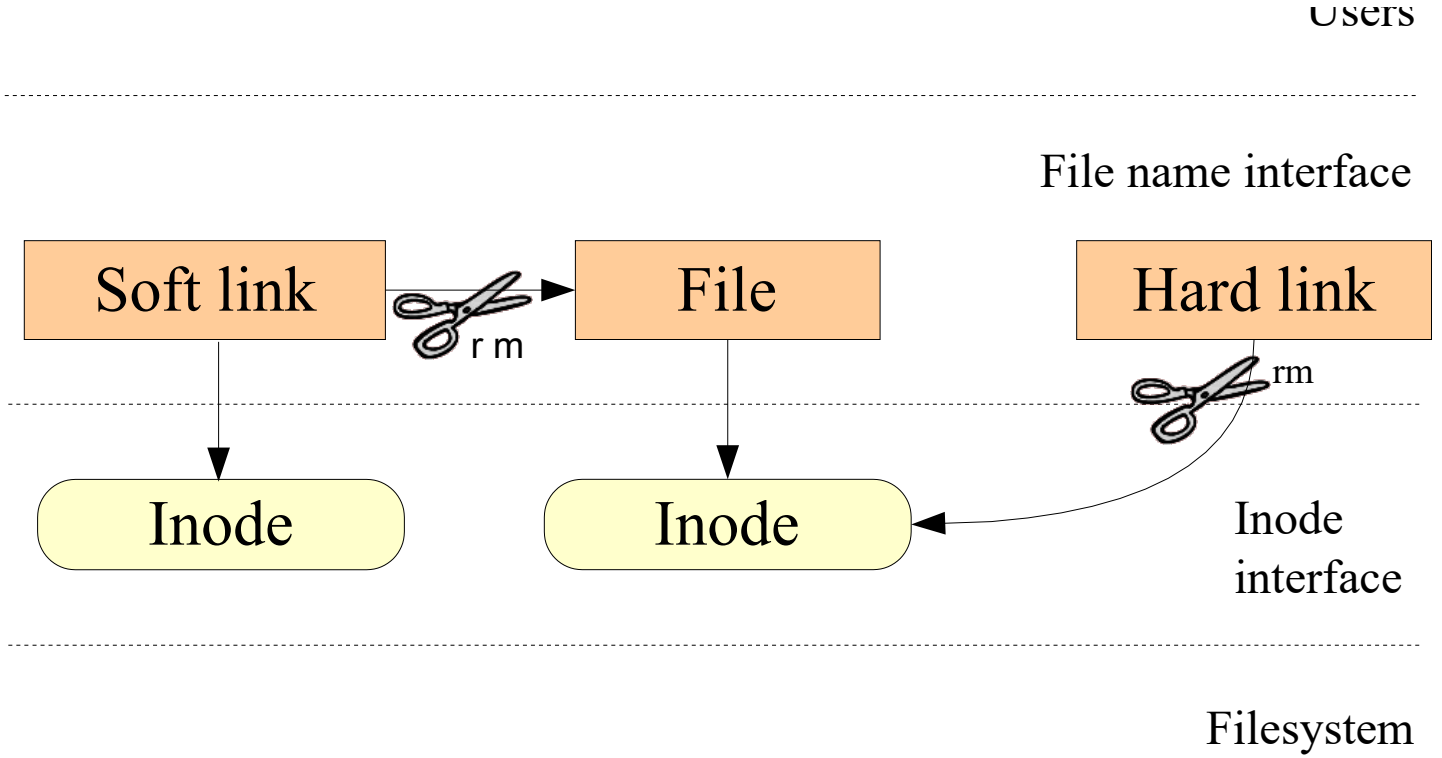
Special devices (3)

- **/dev/full**
  Mimics a full device.
  Useful to check that your application properly handles this kind of situation.

See **man full** for details.

Files names and inodes

Hard Links Vs Soft Links



Shell Programming

Shell Programming

- is “programming”
- Any programming: Use existing tool to create new utilities
- Shell Programming: Use shell facilities to create better utilities
  - Know “commands” --> More tools
  - Know how to combine them

Shell Variables

- **Shell supports variables**
  - **Try:**
    - **j=5; echo $j**
    - **No space in j=5**
  - **Try**
    - **set**
    - **Shows all set variables**
  - **Try**
    - **a=10; b=20; echo ab**
  - **What did you learn?**
  - **All variables are strings**

Shell's predefined variables

- **USER**
  - Name of current user
- **HOME**
  - Home directory of current $USER
- **PS1**
  - The prompt
- **LINES**
  - No. of lines of the screen
- **HOSTNAME**
  - Name of the computer
- **OLDPWD**
  - Previous working directory
- **PATH**
  - List of locations to search for a command's executable file
- **$?**
  - Return value of previou command

Redirection

- **cmd > filename**
  - Redirects the output to a file
  - Try:
    - ls > /tmp/xyz
      cat /tmp/xyz
    - echo hi > /tmp/abc
      cat /tmp/abc
- **cmd < filename**
  - Reads the input from a file instead of keyboard
  - Think of a command now!

Pipes

- Try
  - last | less
  - grep bash /etc/passwd | head – 1
  - grep bash /etc/passwd | head – 2 | tail -1
- Connect the output of LHS command as input of RHS command
- Concept of *filters* – programs which read input only from stdin (keyboard, e.g. scanf, getchar), and write output to *stdout* (e.g. printf, putchar)
- Programs can be connected using pipes if they are filters
- Most Unix/Linux commands are filters !

The *test* command

- **test**
  - **test 10 -eq 10**
  - **test “10” == “10”**
  - **test 10 -eq 9**
  - **test 10 -gt 9**
  - **test “10” >= “9”**
  - **test -f /etc/passwd**
  - **test -d ~/desktop**
  - **…**

Shortcut notation for calling test

[ ]

[10 -eq 10]

Note the space after '[' and before ']'

The *expr* command and backticks

- expr
  - expr 1 + 2
  - a=2; expr $a + 2
  - a=2; b=3; expr $a + $b
  - a=2;b=3; expr $a * $b
  - a=2;b=3; expr $a | $b
- Used for mathematical calculations

- backticks ``
  - j=`ls`; echo $j
  - j=`expr 1 + 2`; echo $j
-

if then else

```
if [ $a -lt $b ]
then
    echo $a
else
    echo $b
fi
```

```
if [ $a -lt $b ];then
echo $a; else  echo $b;
fi
```

0 TRUE

Nonzero FALSE

while do done

```
while [ $a -lt $b ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
while [ $a -lt $b ]
do
    echo $a
    a=$(($a + 1))
done
```

```
while [ $a -lt $b ]; do
echo $a;  a=`expr $a +
1`; done
```

for x in ... do done

```
for i in {1..10}
do
    echo $i
done
```

```
for i in *
do
    echo $i
done
```

```
for i in *; do   echo $i;
done
```

```
read space
case $space in
      [1-6]*)
        Message="one to 6"
        ;;
      [7-8]*)
        Message="7 or 8"
        ;;
      9[1-8])
       Message="9 with a number"
        ;;
      *)
       Message="Default"
        ;;
esac
echo $Message
```

case ... esac

Syntax

;;

) after option

* for default

esac

Try these things

- Print 3rd line from /etc/passwd, which contains the word bash
- Print numbers from 1 to 1000
- Create files named like this: file1, file2, file3, ... filen where n is read from user
  - Read i%5th file from /etc/passwd and store it in file*i*
- Find all files ending in .c or .h and create a .tar.gz file of these files
-

The Golden Mantra

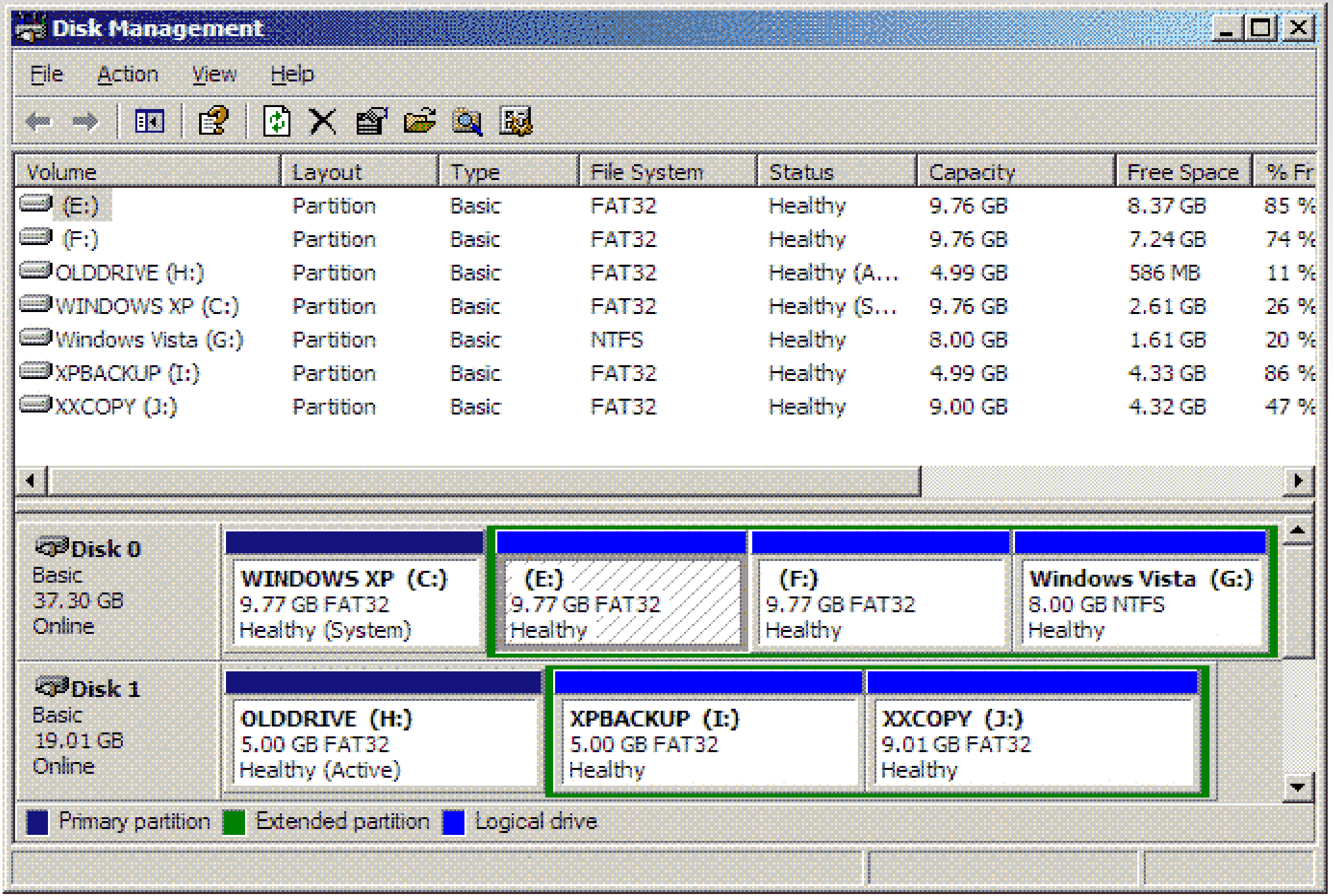
Everything can be done from command line !

Command line is far more powerful than graphical interface

Command line makes you a better programmer

Mounting

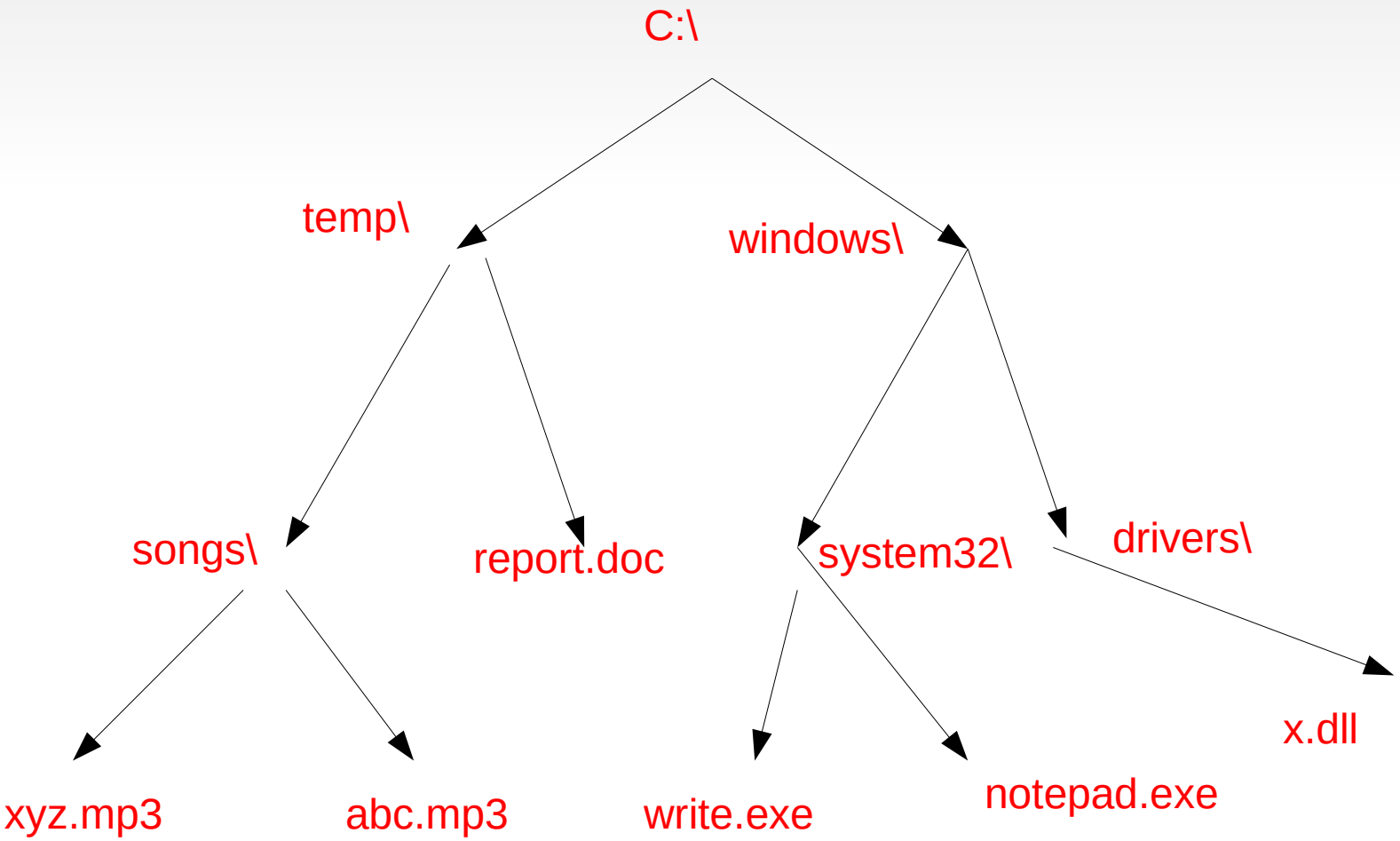
Partitions


Disk Management
File Action View Help
Volume Layout Type File System Status Capacity Free Space % Fr
(E:) Partition Basic FAT32 Healthy 9.76 GB 8.37 GB 85 %
(F:) Partition Basic FAT32 Healthy 9.76 GB 7.24 GB 74 %
OLDDRIVE (H:) Partition Basic FAT32 Healthy (A... 4.99 GB 586 MB 11 %
WINDOWS XP (C:) Partition Basic FAT32 Healthy (S... 9.76 GB 2.61 GB 26 %
Windows Vista (G:) Partition Basic NTFS Healthy 8.00 GB 1.61 GB 20 %
XPBACKUP (I:) Partition Basic FAT32 Healthy 4.99 GB 4.33 GB 86 %
XXCOPY (J:) Partition Basic FAT32 Healthy 9.00 GB 4.32 GB 47 %
Disk 0
Basic
37.30 GB
Online
WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)
(E:)
9.77 GB FAT32
Healthy
(F:)
9.77 GB FAT32
Healthy
Windows Vista (G:)
8.00 GB NTFS
Healthy
Disk 1
Basic
19.01 GB
Online
OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)
XPBACKUP (I:)
5.00 GB FAT32
Healthy
XXCOPY (J:)
9.01 GB FAT32
Healthy
Primary partition
Extended partition
Logical drive

Windows Namespace

c:\temp\songs\xyz.mp3

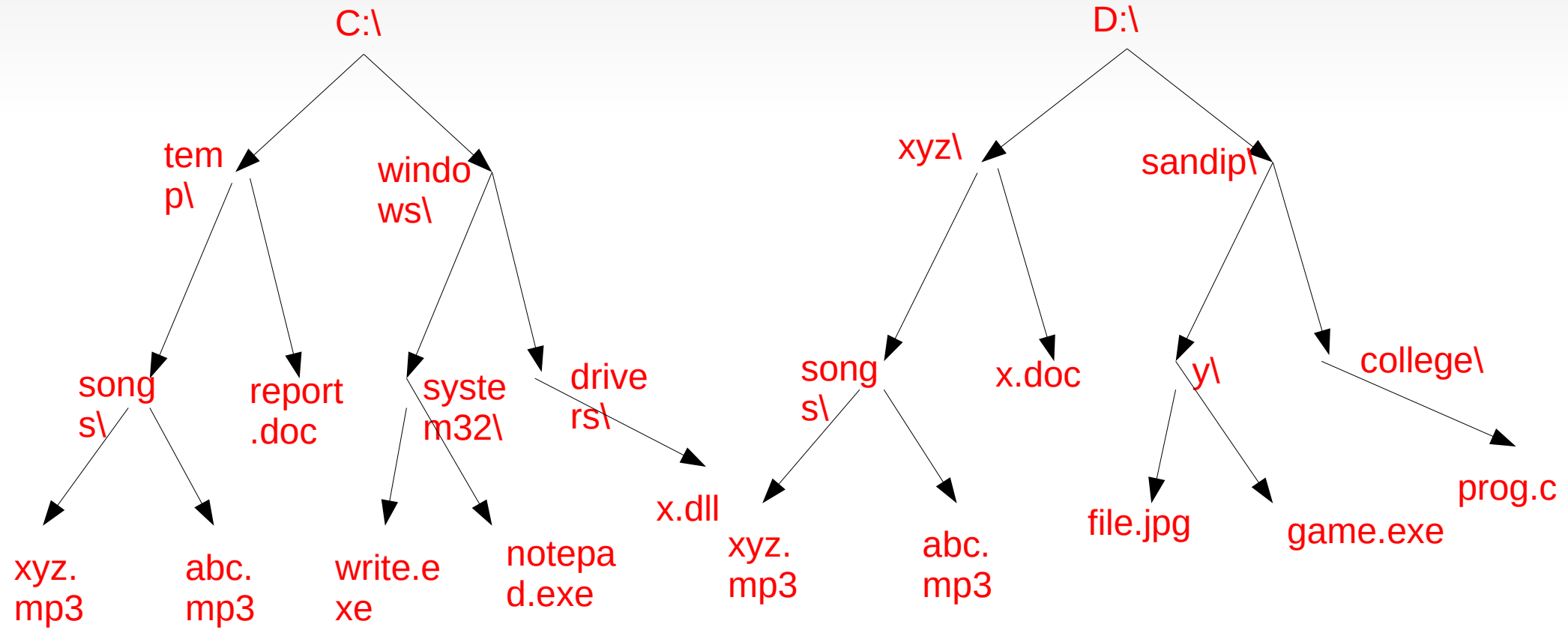
- Root is C:\ or D:\ etc
- Separator is also “\”



Windows Namespace

- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data

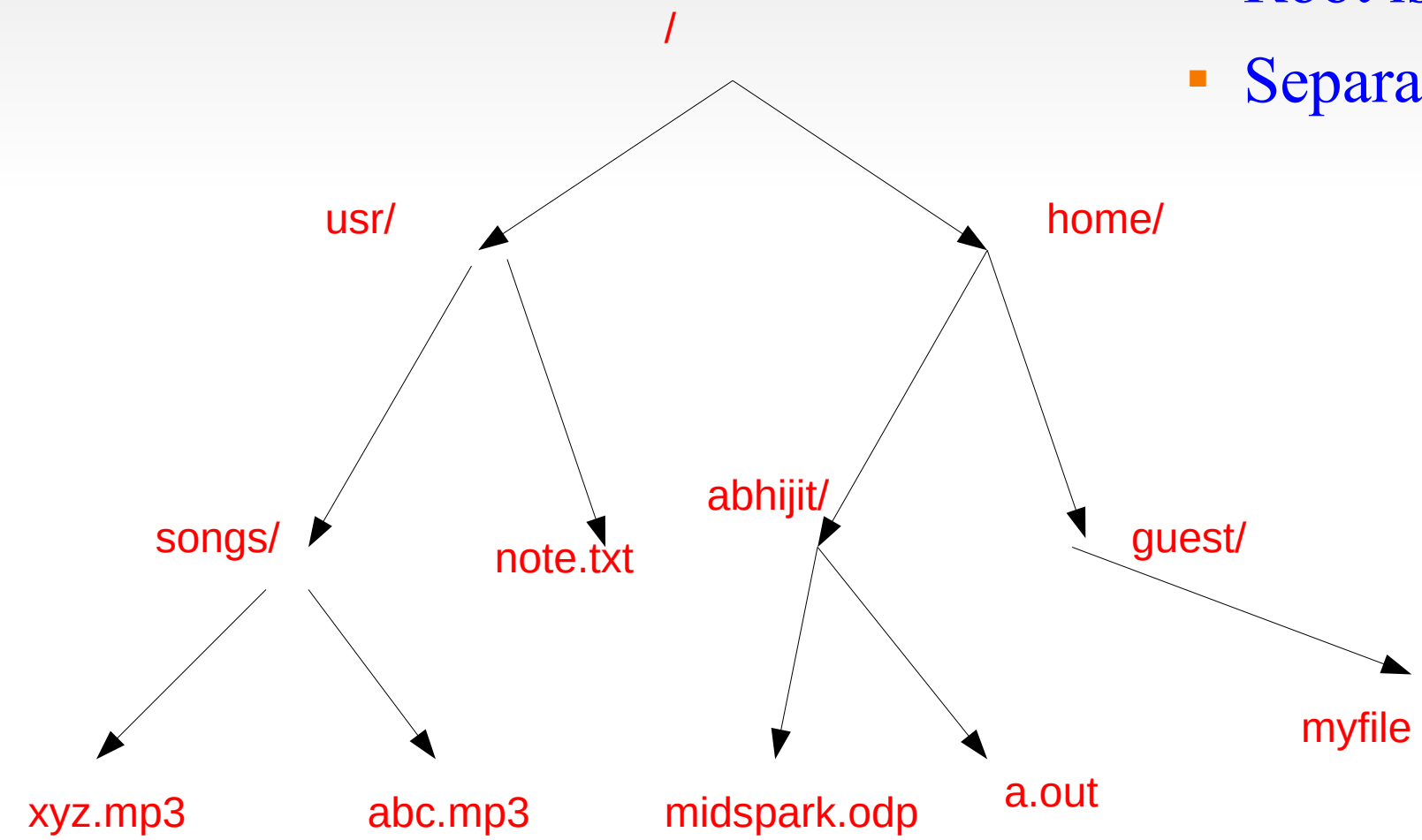
- One “tree” per partition
  - Together they make a “forest”



Linux Namespace: On a partition

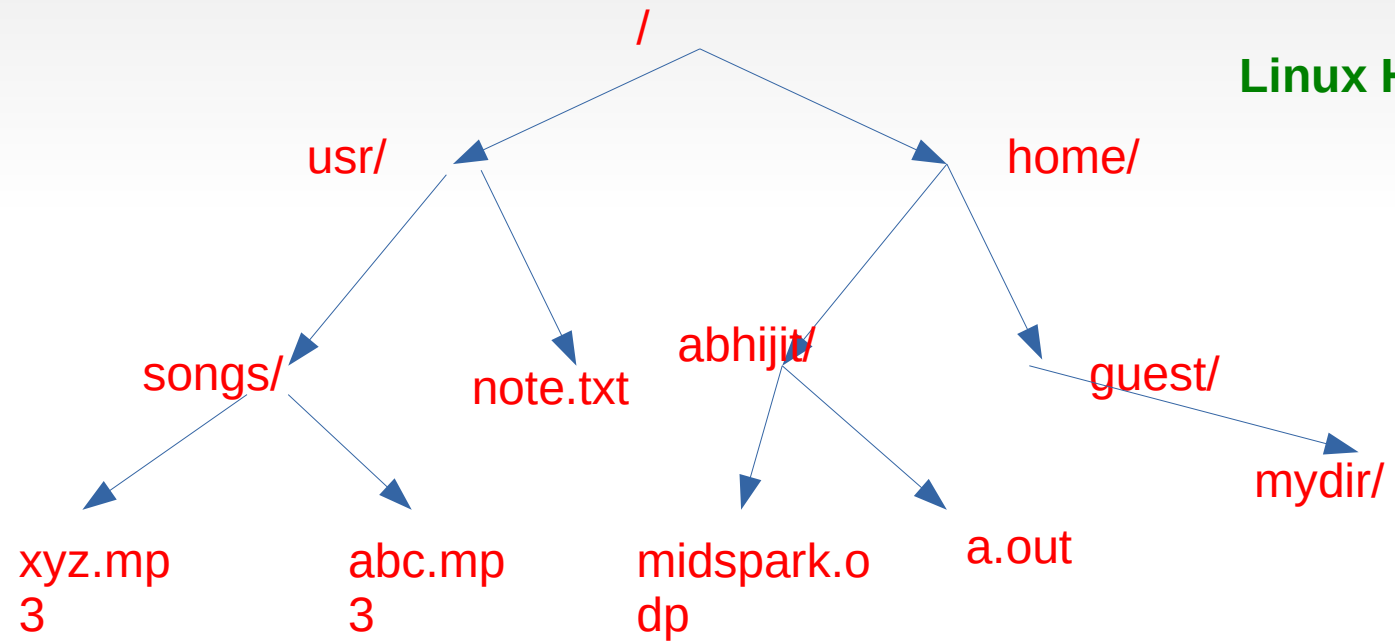
/usr/songs/xyz.mp3

- On every partition:
  - Root is “/”
  - Separator is also “/”

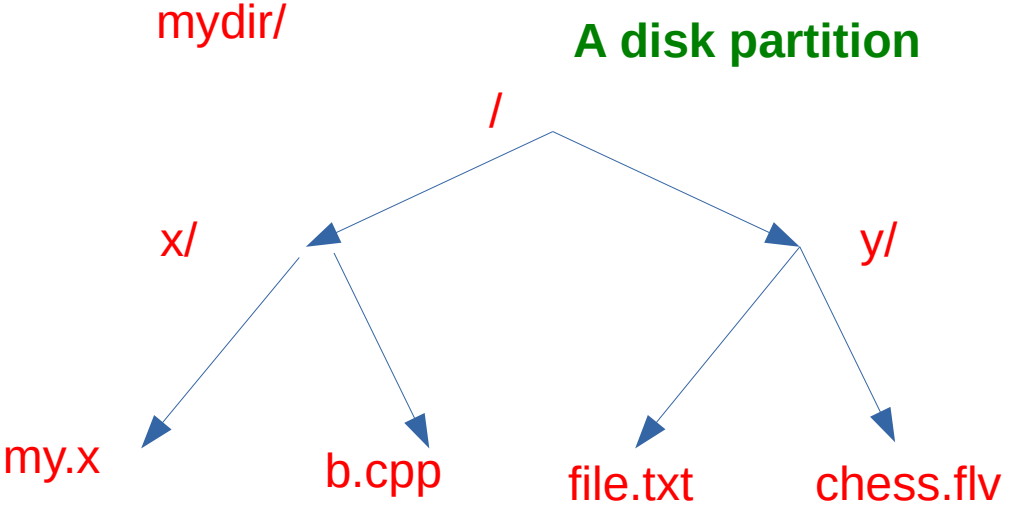


Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”

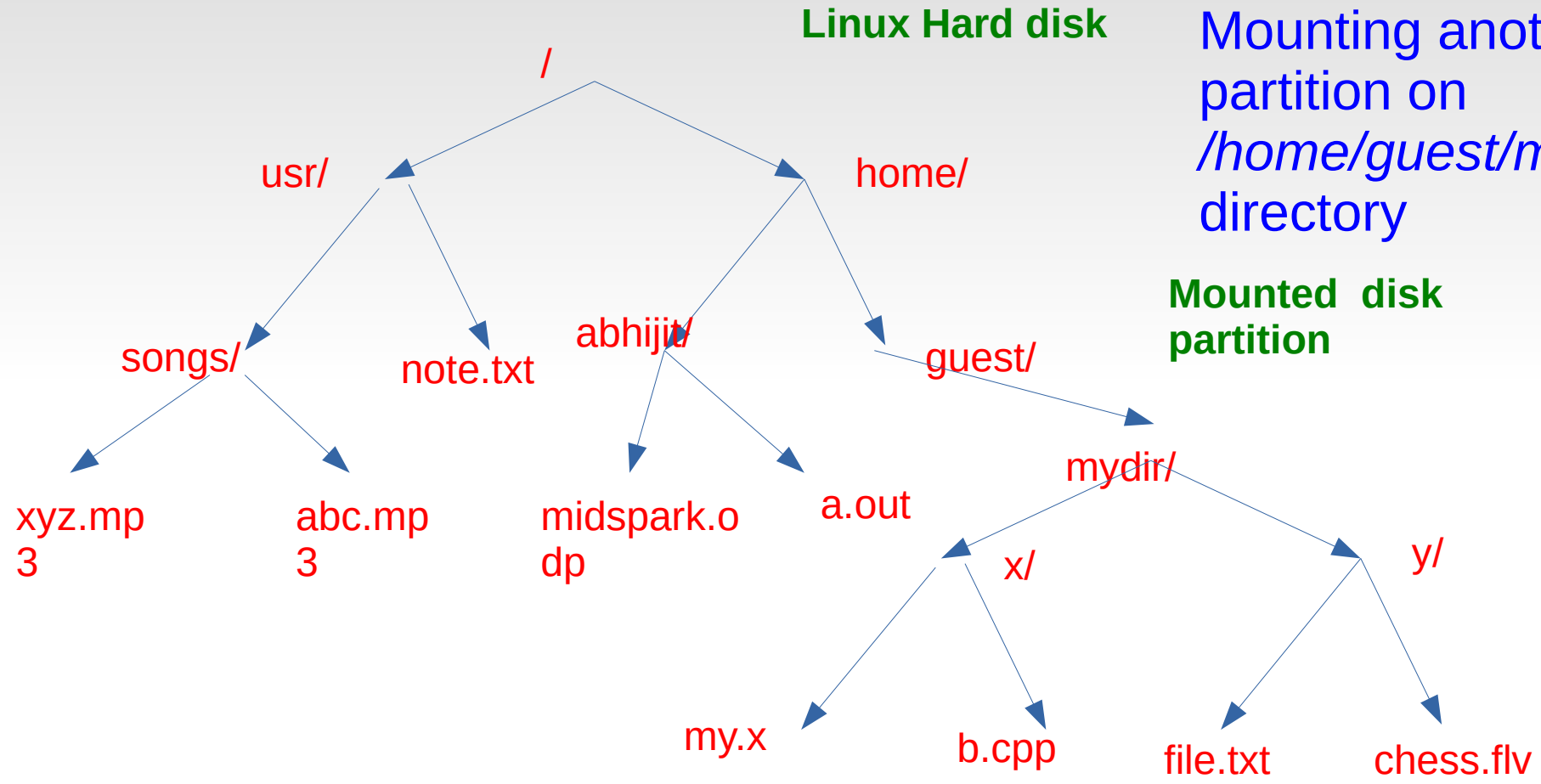


Linux Hard disk



A disk partition

Linux namespace
Mounting a partition



Mounting another partition on */home/guest/mydir/* directory

Mounted disk partition

/home/guest/mydir/x/b.cpp → way to access the file on the other disk partition

Let's go for a live installation Demo !

Some Shell Gimmics

Terminal Tricks

Ctrl + n : same as Down arrow.

Ctrl + p : same as Up arrow.

Ctrl + r : begins a backward search through command history.(keep pressing Ctrl + r to move backward)

Ctrl + s : to stop output to terminal.

Ctrl + q : to resume output to terminal after Ctrl + s.

Terminal Tricks

Ctrl + a : move to the beginning of line.

Ctrl + e : move to the end of line.

Ctrl + d : if you've type something, Ctrl + d deletes the character under the cursor, else, it escapes the current shell.

Ctrl + k : delete all text from the cursor to the end of line.

Ctrl + t : transpose the character before the cursor with the one under the cursor

Terminal Tricks

Ctrl + w : cut the word before the cursor; then Ctrl + y paste it

Ctrl + u : cut the line before the cursor; then Ctrl + y paste it

Ctrl + _ : undo typing.

Ctrl + l : equivalent to clear.

Ctrl + x + Ctrl + e : launch editor defined by $EDITOR to input your command.

Run from history

First: What's history?

Ans: Run 'history

$ history

$!53

$!!

$!cat

$!c

Math

$ echo $((10 + 5)) #15

$ x=1

$ echo $((x++)) #1 , notice that it is still 1, since it's post-incremen

$ echo $((x++)) #2

More Math

```
$ seq 10|paste -sd+|bc
# How does that work ?
```

Using expr

```
$ expr 10+20 #30
$ expr 10\*20 #600
$ expr 30 \> 20 #1
```

More Math

Using bc

$ bc

```
obase=16
ibase=16
AA+1
AB
```

More Math

Using bc

$ bc

```
ibase=16

obase=16

AA+1

07 17

# what went wrong?
```

Fun with grep

```
$ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
# above will only search for IP addresses!
$ grep -v bbo filename
$ grep -w ‘abhijit’ /etc/passwd
$ grep -v '^#' file.txt
$ grep -v ^$  file.txt
```

xargs: convert stdin to args

$ find . | grep something | xargs rm

xargs is highly powerful

rsync

The magic tool to sync your folders!

$ rsync -rvupt ~/myfiles /media/abhijit/PD

$ rsync -rvupt --delete ~/myfiles /media/abhijit/PD

find

```
$ find .
$ find . -type f
$ find . -type d
$ find . -name '*.php'
$ find / -type f -size +4G
$ find . -type f -empty -delete
$ find . -type f | wc -l
```

Download : wget, curl

$ wget foss.coep.org.in

$ wget –r foss.coep.org.in

$ wget -r –convert-links foss.coep.org.in

$ wget -r –convert-links --no-parent foss.coep.org.in/fossmeet/

$ curl https://raw.githubusercontent.com/onceupon/Bash-Oneliner/master/README.md | pandoc -f markdown -t man | man -l -

 # curl is more powerful than wget. Curl can upload. Curl supports many protocols, wget only HTTP/FTP.

Random data

```
# shuffle numbers from 0-100, then pick 15 of them randomly
$ shuf -i 0-100 -n 15

# Random pick 100 lines from a file
$ shuf -n 100 filename

#generate 5 password each of length 13
$ pwgen 13 5

echo $((RANDOM % 10))
```

Run commands remotely

$ ssh administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in

$ ssh -X administrator@foss.coep.org.in firefox

System Information

Show memory usage,. **# print 10 times, at 1 second interval**

$ free -c 10 -mhs 1

 # Display CPU and IO statistics for devices and partitions.
 # refresh every second

$ iostat -x -t 1

 # Display bandwidth usage on an network interface (e.g. enp175s0f0)

$ sudo iftop -i wlo1

 # Tell how long the system has been running and number of users

$ uptime

Surf the web

$ w3m

$ links

Add a user without commands

Know how to edit the */etc/*passwd and */etc/shadow files*

More tricks

Show 10 Largest Open Files

```
$ lsof / | awk '{ if($7 > 1048576) print $7/1048576 "MB" " " $9 " " $1 }' | sort -n -u | tail
```

Generate a sequence of numbers

```
$ echo {01..10}
```

More tricks

 # Rename all items in a directory to lower case

```
$ for i in *; do mv "$i" "${i,,}"; done
```

 # List IP addresses connected to your server on port 80

```
$ netstat -tn 2>/dev/null | grep :80 | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head
```

Credits:
https://onceupon.github.io/Bash-Oneliner/

http://www.bashoneliners.com/

User Administration

Users and Groups

- There is a privileged user called “root”
  - Can do anything, like “administrator” on Windows
  - Can't login in graphical mode !
- Other users are normal users
- Some users are given “sudo” privileges: called *sudoers*
  - Sudo means “do as a superuser”
  - Password is asked, when the otherwise normal user tries to do administrative task
  - The first user account created on Ubuntu, is by default with sudo privileges

Adding/Deleting/Changing users

- System → Administration → Users and Groups
- Click on “Add” to add a user
  - Asks for password !
  - Provide the details asked for
  - Verify the user was created, by doing 'switch user'
- Try 'deleting' the user created
- Groups: Various groups of users meant for different purposes
  - Every user by default belongs to her own group
  - Add the user explicitly to other groups

Software installation

Some terms

- **.deb**
  - **The “setup” file. The installer package. Similar to Setup.exe on windows.**
  - **Contains all *binary* files and some *shell scripts***
- **repository**
  - **A collection of .deb files, categorized according to type (security, main, etc.)**
- **Software source/ ubuntu mirror**
  - **A computer on internet having all .deb files for ubuntu**

Software Installation Concept

- **Online installation**
  - Use the “Ubuntu Software Center” to select the software, click and install !
  - Software is fetched automatically and installed !
  - Much easier than Windows !
- **Offline installation**
  - Collect ALL .deb files for your application
  - Select all, and install using package manager

Software Installation

- **When we install using Software Center**
  - **.deb files are stored in /var/cache/apt/archives folder**
- **One needs to be a *sudoer* to install software**
- **Try installing some software on your own and try them out !**

Network configuration

Setting up network for a desktop

- **DHCP**
  - Nothing needs to be done !
  - Default during installing Linux
- **Static I/P**
  - System → Preferences → Network connections
  - System → Administration → Network
  - System → Administration → Network tools

Network setup for wireless

- **Just plug and Play !**
- **Network icon shows available wireless network, just click and connect.**

Reliance/Tata/Idea USB devices

- Each has a different procedure
- One needs to search the web for setting it up
  - Most of the devices work plug and play on Ubuntu 12.04
  - Some do not work, as fault of the providers, they have not given an installer CD for Linux!
    - Still Linux community have found ways to work around it !
    - My Reliance netconnect worked faster and more steadily on Linux than Windows !

Disk Management

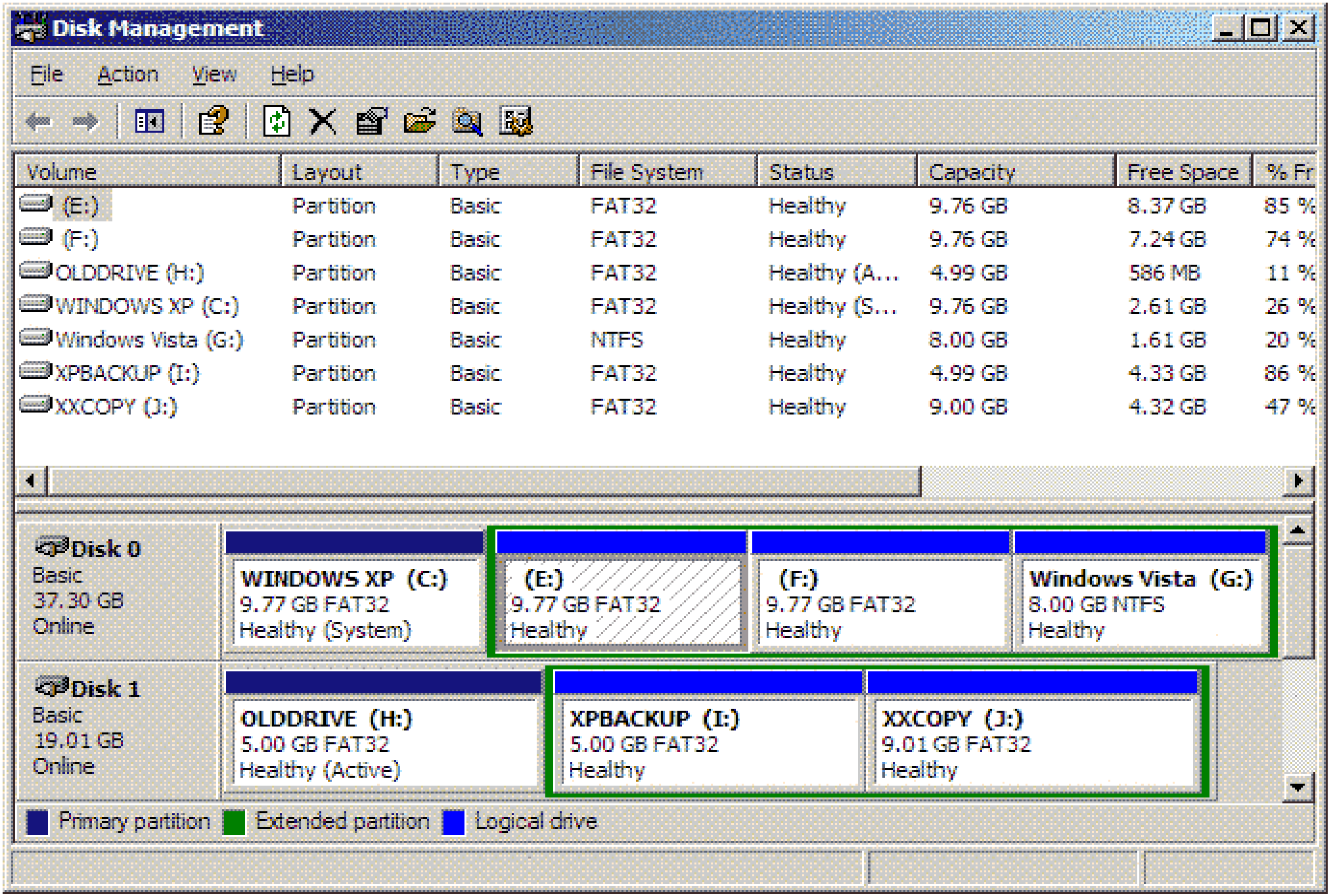
Partition

- What is C:\ , D:\, E:\ etc on your computer ?
  - "Drive" is the popular term
  - Typically one of them represents a CD/DVD RW
- What do the others represent ?
  - They are "partitions" of your "hard disk"

Partition

- Your hard disk is one contiguous chunk of storage
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- A logical chunk of storage is partition
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

Partitions


Disk Management
File Action View Help
Volume Layout Type File System Status Capacity Free Space % Fr
(E:) Partition Basic FAT32 Healthy 9.76 GB 8.37 GB 85 %
(F:) Partition Basic FAT32 Healthy 9.76 GB 7.24 GB 74 %
OLDDRIVE (H:) Partition Basic FAT32 Healthy (A... 4.99 GB 586 MB 11 %
WINDOWS XP (C:) Partition Basic FAT32 Healthy (S... 9.76 GB 2.61 GB 26 %
Windows Vista (G:) Partition Basic NTFS Healthy 8.00 GB 1.61 GB 20 %
XPBACKUP (I:) Partition Basic FAT32 Healthy 4.99 GB 4.33 GB 86 %
XXCOPY (J:) Partition Basic FAT32 Healthy 9.00 GB 4.32 GB 47 %
Disk 0
Basic
37.30 GB
Online
WINDOWS XP (C:)
9.77 GB FAT32
Healthy (System)
(E:)
9.77 GB FAT32
Healthy
(F:)
9.77 GB FAT32
Healthy
Windows Vista (G:)
8.00 GB NTFS
Healthy
Disk 1
Basic
19.01 GB
Online
OLDDRIVE (H:)
5.00 GB FAT32
Healthy (Active)
XPBACKUP (I:)
5.00 GB FAT32
Healthy
XXCOPY (J:)
9.01 GB FAT32
Healthy
Primary partition
Extended partition
Logical drive

Managing partitions and hard drives

- System → Administration → Disk Utility
- Had drive partition names on Linux
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, ... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- Pen drives can also be managed from here
- Formatting can also be done from here