

# C Arrays and Pointers

# Pointers and Arrays

- An array name is considered pointer to first element
  - **a** is pointer to **a[0]**
  - **pa = &a[0]** and **pa=a** mean the same thing
  - **a+1** means L-value of a[0] plus as many bytes as are needed to store value of elements of a's type
    - Pointer arithmetic is an address calculation with respect to the underlying architecture
- An array name is not a variable
  - **a++** and **a = pa** are ILLEGAL!

# Pointer Arithmetic

```
int *k; k=&j;
```

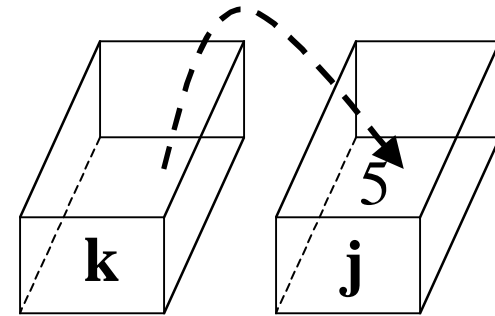
**(\*k+1)**, legal R-value, illegal L-value

**(\*k+1)** means **((\*k)+1)**

```
h = *k + 1;
```

**\*(k+1)** legal (but not meaningful) L-value, legal R-value

**/\*need to know layout of storage to see to what (k+1) points, to the byte that is 4 bytes beyond the L-value of k, since adding 1 is like adding storage for 1 int (4 bytes)\*/**



**k++** has same properties when used with **\***

# Multipleptr.c

```
#include<stdio.h>
/*program to show multiple levels of dereference*/
main( )
{
    int j, k, l;
    int *q;
    int **s;
    j = 99;
    q = &j; /* q = j is ILLEGAL */
    s = &q; /* s = q is ILLEGAL */
/*all these names **s, *q, j are synonyms or aliases
  for the same storage location at this program point*/
    printf(" %d %d %d\n",**s, *q, j);
}
/* output
47 scherzo!c> gcc multipleptr.c
48 scherzo!c> a.out
99 99 99 */
```

# Problems with Pointers

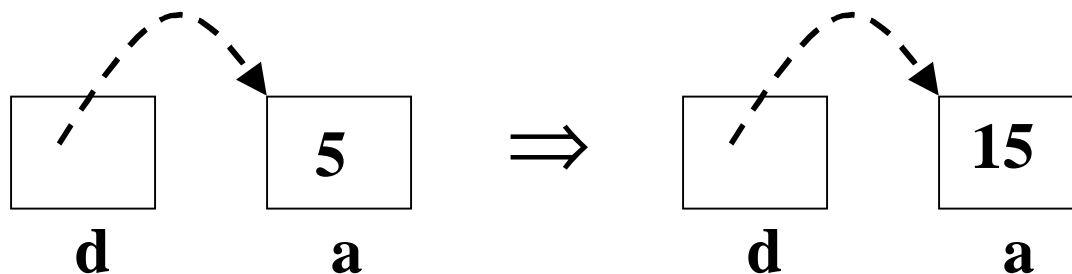
- **Uninitialized pointers**

- **int \*a; \*a = 12;**

- Can get *segmentation fault* error (value in **a** is meaningless address)
    - Can actually store **12** into some memory location accessible to your program, whose address corresponds to the random bits in **a**

# Problems with Pointers

- **Null** doesn't point to anything by definition so cannot dereference it
  - **a = 0;** /\* makes **a**'s value NULL \*/
  - if (a == NULL) ... /\* tests for a NULL pointer value\*/
- **Multiple level pointers**
  - Can be used as L-values or R-values



```
int a;  
int *d;  
d = &a;  
a = 5;  
*d = 10 + *d;  
a = 10 + 5
```

# More Problems with Pointers

- **Dangling pointer**
  - **Storage pointed to is freed, but pointer is not set to NULL**
  - **Then you are able to access storage whose values are not meaningful**
- **Garbage**
  - **Pointer itself is freed (perhaps by execution going out of its declaring scope) but heap locations pointed to are not freed**
  - **Then, there is no way to access this heap storage**

# *Stack* vs **Heap**

- *Procedure activations, statically allocated local variables, parameter values*
- *Lifetime same as block in which variables are declared*
- *Stack frame with each invocation of procedure*
- **Dynamically allocated data structures, whose size may not be known in advance**
- **Lifetime extends beyond block in which they are created**
- **Must be explicitly freed or garbage collected**

# Managing the Heap

- **Explicit allocation and deallocation**
  - **Garbage and dangling pointers**
  - **Memory leaks: not freeing storage appropriately**
  - **E.g., Pascal, C, C++**
- **Implicit allocation and deallocation**
  - **Garbage collection**
    - **Overhead at run-time to have a separate process analyze usage of heap and recover pieces of storage no longer reachable from user pointers**
    - **Execution time cost traded for easier job for user**
    - **E.g., functional languages such as Scheme, Java**

# Casting

- **Safe uses of casting**

- **For pointers returned from *malloc***

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

- **For simulating subtyping safely in C**

```
struct s{  
    int a;  
    int b;  
    double c;  
}
```

```
struct t{  
    int a;  
    int b;  
}
```

**s** is like a subtype of **t**  
because it has same  
fields as **t** plus an  
extra field.

# Heap Storage

- *void \*malloc (size\_t n)*
    - returns pointer to block of contiguous storage of *n* bytes (chars), if possible
    - if not enough memory left for allocation, *malloc* returns NULL pointer
      - So you ALWAYS have to check return the value
    - to allocate storage of a different type requires sending *malloc* the proper amount of bytes needed and casting the return pointer value appropriately
- ```
head = (listcell *) malloc(sizeof (listcell));
```