Arrays- II

CS10001: Programming & Data Structures

Sudeshna Sarkar
Dept. of Computer Sc. & Engg.,
Indian Institute of Technology Kharagpur

Reading Array Elements

```
/* Read in student midterm and final grades and store them in two arrays*/
#define MaxStudents 100
int midterm[MaxStudents], final[MaxStudents];
int NumStudents ;     /* actual no of students */
int i, done, Smidterm, Sfinal;

printf (“Input no of students :”);
scanf(“%d”, &NumStudents) ;
if (NumStudents > MaxStudents)
      printf (“Too many students”) ;
else
      for (i=0; i<NumStudents; i++)
                scanf(“%d%d”, &midterm[i], &final[i]);
```

Reading Arrays - II

```
/* Read in student midterm and final grades and store them in 2 arrays */
#define MaxStudents 100
int midterm[MaxStudents], final[MaxStudents];
int NumStudents ;     /* actual no of students */
int i, done, Smidterm, Sfinal;
done=FALSE; NumStudents=0;
while (!done)     {
    scanf(“%d%d”, &Smidterm, &Sfinal);
    if (Smidterm !=-1 || NumStudents>=MaxStudents)
          done = TRUE;
    else  {
          midterm[NumStudents] = Smidterm;
          final[NumStudents] = Sfinal;
          NumStudents++;
    }
}
```

Size of an array

- How do you keep track of the number of elements in the array ?
  - 1. Use an integer to store the current size of the array.

    ```
    #define MAX 100
    int  size;
    float cost[MAX] ;
    ```

  - 2. Use a special value to mark the last element in an array. If 10 values are stored, keep the values in cost[0], ... , cost[9], have cost[10] = -1
  - 3. Use the 0th array element to store the size (cost[0]), and store the values in cost[1], ... , cost[cost[0]]

Add an element to an array

1. cost[size] = newval; size++;

2. for (i=0; cost[i] != -1; i++) ;

```
            cost[i] = newval;
        cost[i+1] = -1;
```

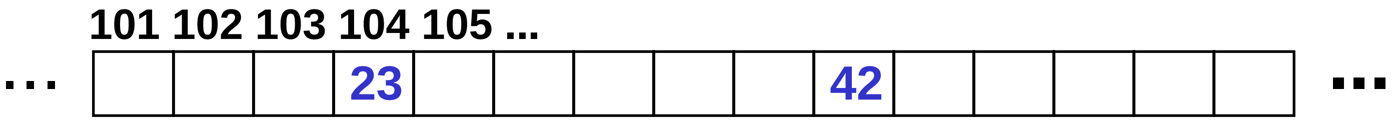
3. cost[0]++;

```
        cost[cost[0]] = newval;
```

Address vs. Value

- Each memory cell has an **address** associated with it.
- Each cell also stores some value.

- Don’t confuse the **address** referring to a memory location with the value stored in that location.

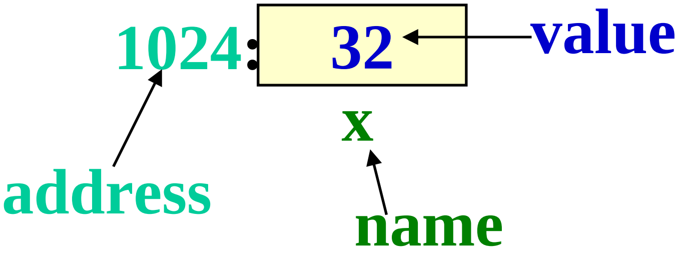


101 102 103 104 105 ...

... 23 42 ...

Values vs Locations

- Variables name memory locations, which hold values.



New Type : Pointer

Pointers

- A pointer is just a C variable whose value is the address of another variable!
- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet. We can either:

  - make it point to something that already exists, or
  - allocate room in memory for something new that it will point to… (next time)

Pointer

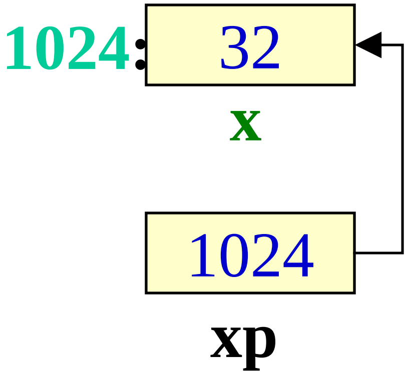
```
int x;
```

pointer to int

```
int * xp ;

xp = &x ;
```

address of x



```
*xp = 0;        /* Assign 0 to x */
*xp = *xp + 1; /* Add 1 to x */
```

Pointers Abstractly

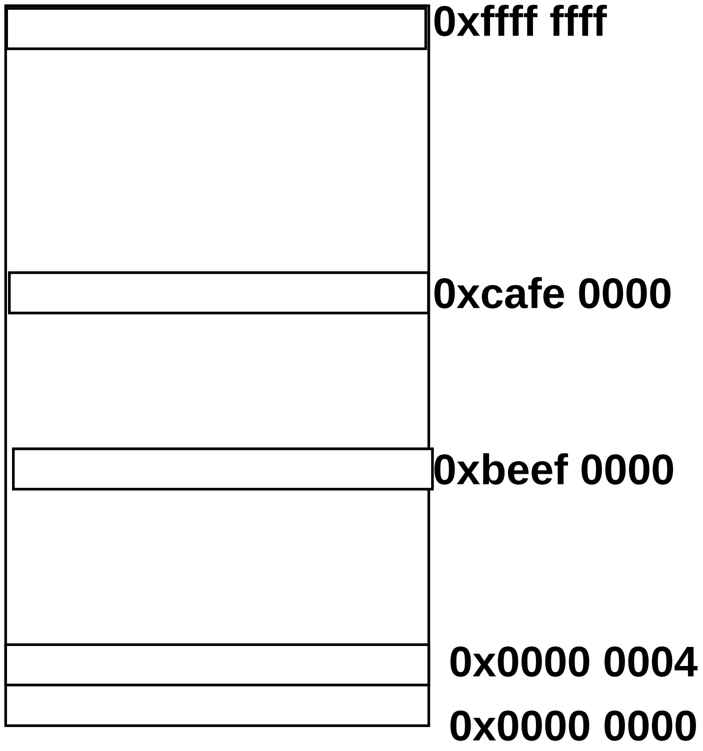
```
int x;
int * p;
p=&x;
...
(x == *p)    True
(p == &x)    True
```

Pointers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!

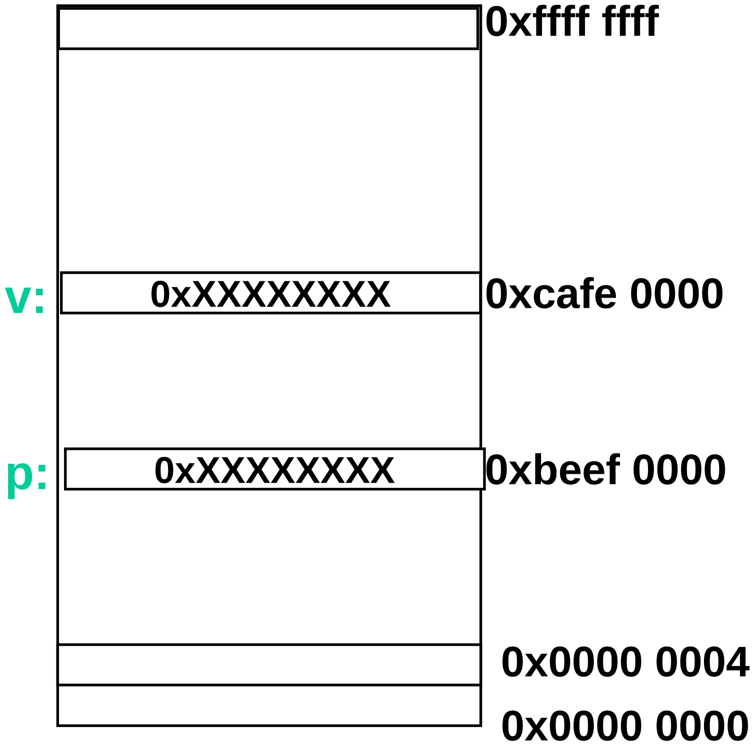
- Local variables in C are not initialized, they may contain anything.

Pointer Usage Example



Memory and Pointers:

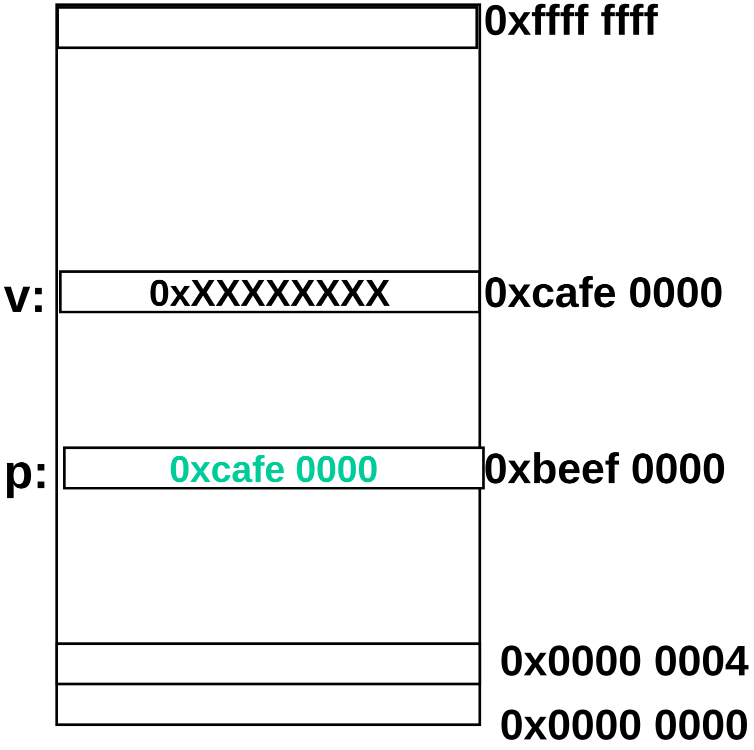
Pointer Usage Example



Memory and Pointers:

int *p, v;

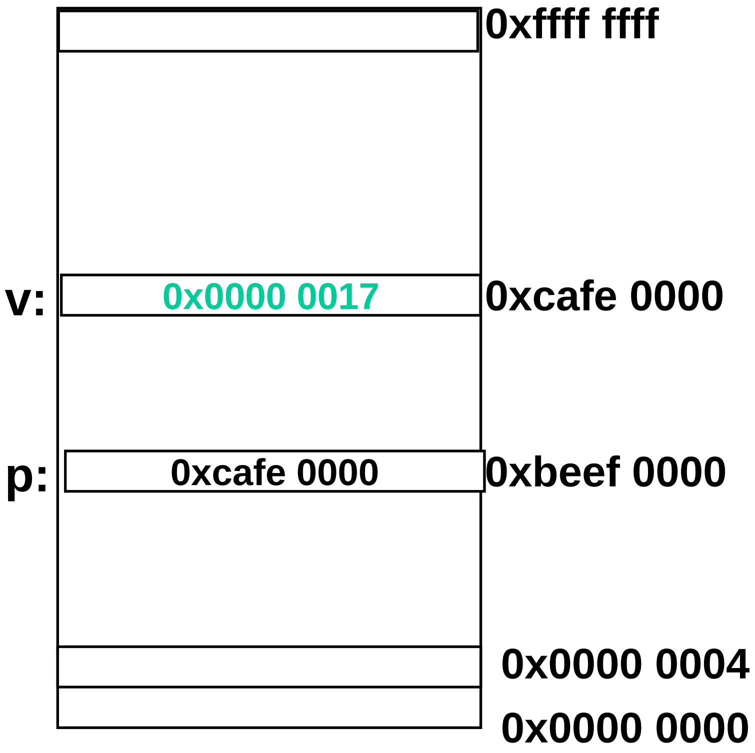
Pointer Usage Example



Memory and Pointers:

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

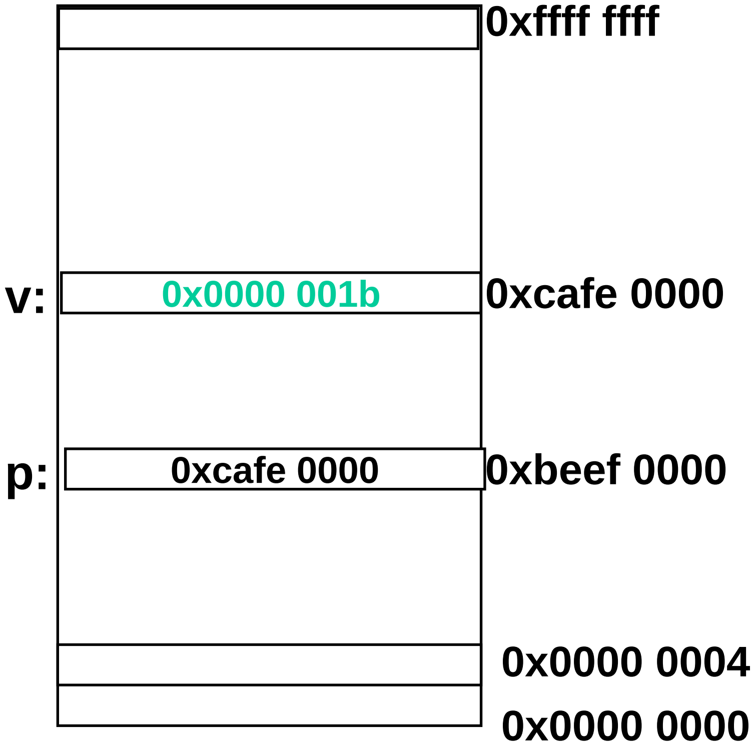
Pointer Usage Example



Memory and Pointers:

```
int *p, v;
p = &v;
v = 0x17;
```

Pointer Usage Example



0xffff ffff

v: 0x0000 001b 0xcafe 0000

p: 0xcafe 0000 0xbeef 0000

0x0000 0004

0x0000 0000

Memory and Pointers:

```
int *p, v;
p = &v;
v = 0x17;
*p = *p + 4;
V = *p + 4
```

Arrays and pointers

- An array name is an address, or a pointer value.

- Pointers as well as arrays can be subscripted.

- A pointer variable can take different addresses as values.

- An array name is an address, or pointer, that is fixed.

It is a CONSTANT pointer to the first element.

Arrays

- Consequences:
  - `ar` is a pointer
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- Declared arrays are only allocated while the scope is valid

```
char *foo() {
    char string[32]; ...;
    return string;
}
```

is incorrect

Pointer Arithmetic

- Since a pointer is just a mem address, we can add to it to traverse an array.

- `p+1` returns a ptr to the next array elt.

- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, `p+1` doesn't add `1` to the memory address, it adds the size of the array element.

Pointer Arithmetic

- So what's valid pointer arithmetic?
  - Add an integer to a pointer.
  - Subtract 2 pointers (in the same array).
  - Compare pointers (`<, <=, ==, !=, >, >=`)
  - Compare pointer to `NULL` (indicates that the pointer to nothing).

Pointer Arithmetic

- We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

  - C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a `char`, 4 bytes for an `int`, etc.)

Pointer Arithmetic

- C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.
- So the following are equivalent:

```
int get(int array[], int n)
{
    return  (array[n]);
    /* OR */
    return *(array + n);
}
```

Arrays

- Wrong rather bad practice

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```

- Right rather recommended

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? SINGLE SOURCE OF TRUTH
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10

Arrays

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find;
    be careful!
  - You'll learn how to debug these in lab…

Arrays In Functions

- **An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.**
  - Can be incremented

```
int strlen(char s[])          int strlen(char *s)
{                             {


}                             }
```

Arrays and pointers

int a[20], i, *p;

- The expression a[i] is equivalent to *(a+i)
- p[i] is equivalent to *(p+i)
- When an array is declared the compiler allocates a sufficient amount of contiguous space in memory. The base address of the array is the address of a[0].
- Suppose the system assigns 300 as the base address of a. a[0], a[1], ...,a[19] are allocated 300, 304, ..., 376.

Arrays and pointers

#define N 20

int a[2N], i, *p, sum;

- **p = a; is equivalent to p = *a[0];**
- **p is assigned 300.**
- **Pointer arithmetic provides an alternative to array indexing.**
- **p=a+1; is equivalent to p=&a[1]; (p is assigned 304)**

```
for (p=a; p<&a[N]; ++p)
      sum += *p ;
```

```
p=a;
for (i=0; i<N; ++i)
      sum += p[i] ;
```

```
for (i=0; i<N; ++i)
      sum += *(a+i) ;
```

Arrays and pointers

```
int a[N];
```

- a is a **constant pointer.**

- ~~a=p; ++a; a+=2;~~ illegal

Arrays as parameters of functions

- An array passed as a parameter is not copied

- An array name is a constant whose value serves as a reference to the first (index 0) item in the array.

Arrays as parameters of functions

- **Since constants cannot be changed, assignments to array variables are illegal.**
- **Only the array name is passed as the value of a parameter, but the name can be used to change the array's contents.**
- **Empty brackets [] are used to indicate that the parameter is an array. The no of elements allocated for the storage associated with the array parameter does not need to be part of the array parameter.**

Array operations

```
#define MAXS 100
int insert (int[], int, int, int) ;
int delete (int[], int, int) ;
int getelement (int[], int, int) ;
int readarray (int[], int) ;
int main ()        {
    int a[MAXS];
    int size;
    size = readarray (a, 10) ;
    size = insert (a, size, 4, 7) ;
    x = getelement (a, size, 3) ;
    size = delete (a, size, 3) ;
}
```

Array operations

```
#define MAXS 100
int insert (int[], int, int, int) ;
int delete (int[], int, int) ;
int getelement (int[], int, int) ;
int readarray (int[], int) ;
int main ()          {
    int a[MAXS];
    int size;
    size = readarray (a, 10) ;
    size = insert (a, size, 4, 7) ;
    x = getelement (a, size, 3) ;
    size = delete (a, size, 3) ;
}
```

```
int readarray (int x[], int size)   {
      int i;
      for (i=0; i<size; i++)
            scanf(“%d”, &x[i]) ;
      return size;
}
```

```
int getelement (int x[], int size, int pos){
    if (pos <size) return x[pos] ;
    return -1;
}
```

```
int insert (int x[], int size, int pos. int val){
    for (k=size; k>pos; k--)
          x[k] = x[k-1] ;
    x[pos] = val ;
    return size+1;
}
```

```
void reverse (int x[],   int size)  {




}
```

```
int findmax (int x[], int size)
                              {






}
```

```
void reverse (int x[], int size)  {
    int i;
    for (i=0; i< (size/2); i++)
            temp = x[size-i-1] ;
            x[size-1-1] = x[i] ;
            x[i] = temp;
}
```

```
int findmax (int x[], int size) {
    int i, max;
      max = x[0];
    for (i=1; i< size; i++)
            if (x[i] > max)
                max = x[i] ;
      return max;
}
```