CS 2213
Advanced Programming

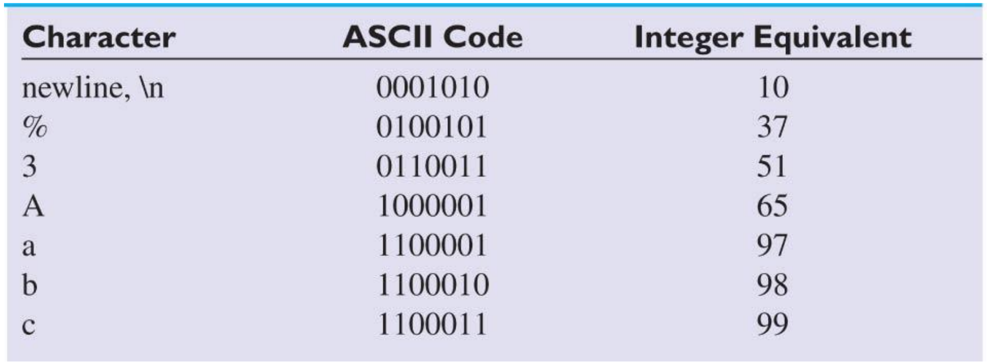
Ch 2 – Overview – C programming Language
Data types – Pointers – Arrays - Structures

Turgay Korkmaz

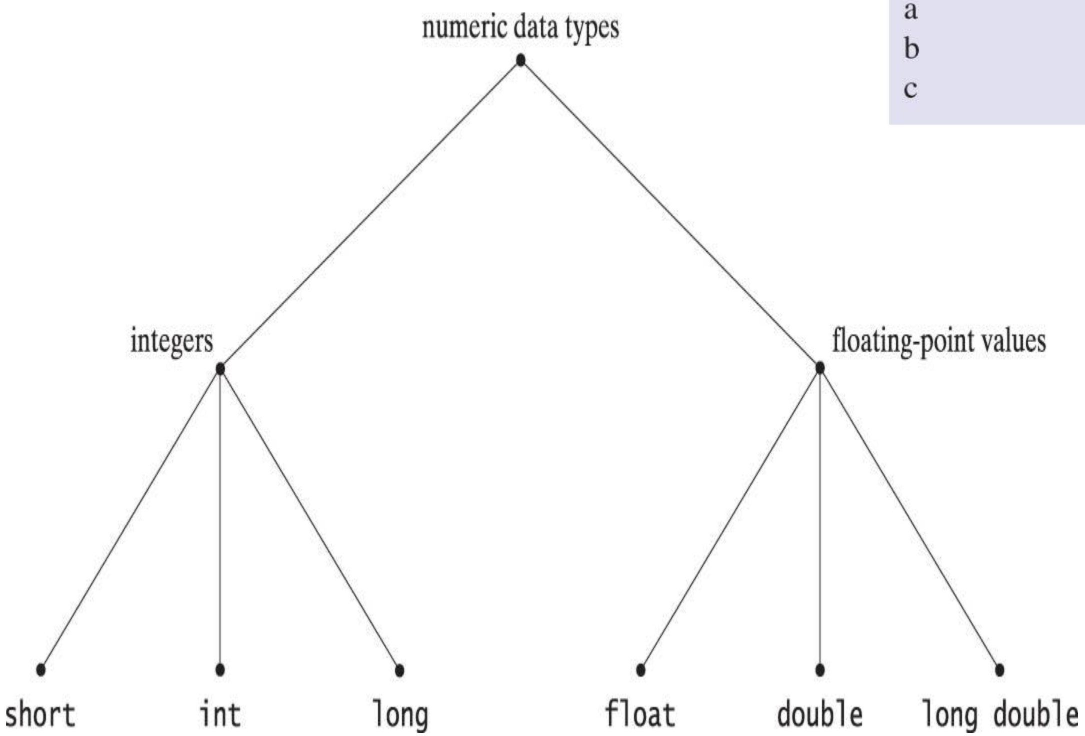
Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

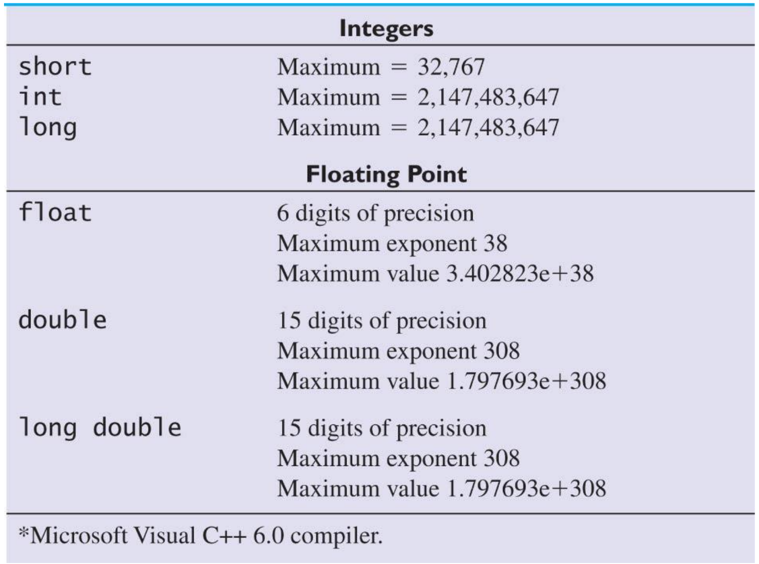
Built-in Atomic Data Types in C

`char`



Character	ASCII Code	Integer Equivalent
newline, \n	0001010	10
%	0100101	37
3	0110011	51
A	1000001	65
a	1100001	97
b	1100010	98
c	1100011	99





Integers	
`short`	Maximum = 32,767
`int`	Maximum = 2,147,483,647
`long`	Maximum = 2,147,483,647
Floating Point	
`float`	6 digits of precision Maximum exponent 38 Maximum value 3.402823e+38
`double`	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308
`long double`	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308

*Microsoft Visual C++ 6.0 compiler.

Define a new type, rename an old type using **typedef**

- What is the difference between

```
int partNumberT, serialNumberT;
typedef int partNumberT, serialNumberT;
partNumberT x, y;    vs. int x, y;
```

  - New types may provide more information
  - Easily change the underlying representation

```
typedef long partNumberT, serialNumberT;
```

genlib.h defines two new types

```
typedef int bool; // or
typedef enum {FALSE, TRUE} bool;
typedef char *string;
```

New Atomic Data Types in C

- C allows to define new atomic types called **enumeration types**

typedef enum { *element-list* **}** *type_name*;

- For example:

```
typedef enum {
  North, East, South, West
} directionT;
directionT dir;/* declare a variable */
```

Internal representation of enum types

- Stored as integers starting with 0
  - `North=0, East=1, South=2, West=3.`
- We can change these values

```
typedef enum {
  Penny=1, Nickle=5, Dime=10, Quarter=25,
  HalfDollar=50
} coinT;
typedef enum {
  January=1, February, March …
} monthT;
```

```
#define January 1
#define February 2
#define March 3
…
```

Scalar types

- enum types, char, and int, short, long etc are called **scalar** type and automatically converted to integer
- So any operations on scalar type are the same as for integers

```
directionT RightFrom(directionT dir)
{
  return ((dir+1)%4);
}
```

```
…
for(m=January; m <=December; m++)
…
```

Scalar types (cont’d)

- You can use scalar types as integers

  printf(“direction is %d\n”, North); will print 0

- If you need to print the name itself you need a function like the following

```
String DirectionName(directionT dir)
{
   switch(dir) {
      case North:   return(“North”);
      case East:    return(“East”);
      case South:   return(“South”);
      case West:    return(“West”);
  }
}
```

Data and Memory

- How is the information stored in a computer?
- RAM -- Random Access Memory
- Bits, bytes, words (size required to hold int, 16, 32 or 64)
- Memory addresses
- `sizeof(type_name) sizeof var_name`

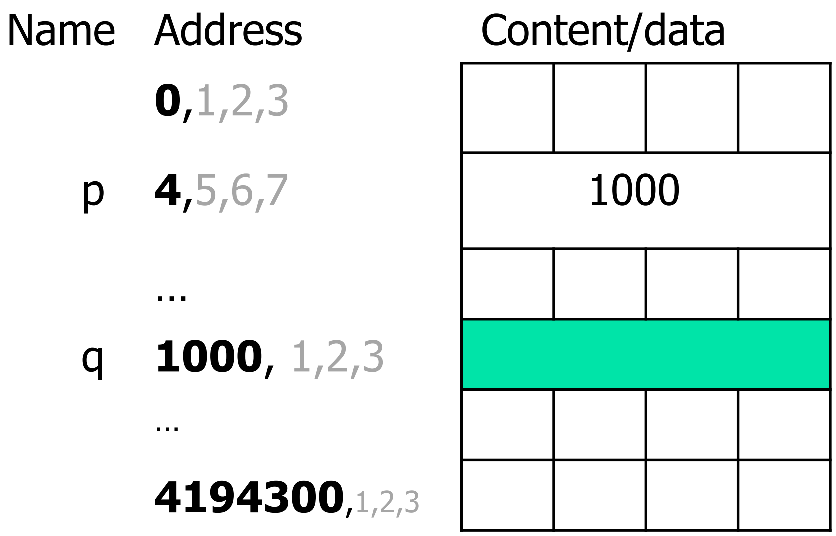
Memory addresses

- Every byte is identified by a numeric address
- Each byte of memory holds one character (8-bit info)
- Values larger than char are stored in consecutive bytes
  - Identified by the address of the first byte
- Suppose we declare

```
char ch = 'A';
int x = 305;
```

Name	Address	Content/data
	0	
	1	
ch	2	65
	…	
x	1000	🟩
	1001	🟩
	1002	🟩
	1003	🟩
	…	
	4194303	

Suppose we have 4MB memory



Name	Address	Content/data
	0,1,2,3	
p	**4**,5,6,7	1000
	…	
q	**1000**, 1,2,3	🟩
	…	
	4194300,1,2,3	

Pointers

- A pointer is a variable that contains the *address* of a variable.
  - Addresses are used as data values
- Using pointers, we can directly access memory at this address and update its content
  - Why not just use the variable name!
- Java hides pointers from the programmer…
- One of the key concepts that we will learn!

Why do we need pointers

- To refer to a large data structure in a **compact way**
- Facilitate **data sharing** between different parts of the program
- Make it possible to reserve **new memory** during program execution
- Allow to represent **record relationships** among data items (data structures: link list, tree etc…)

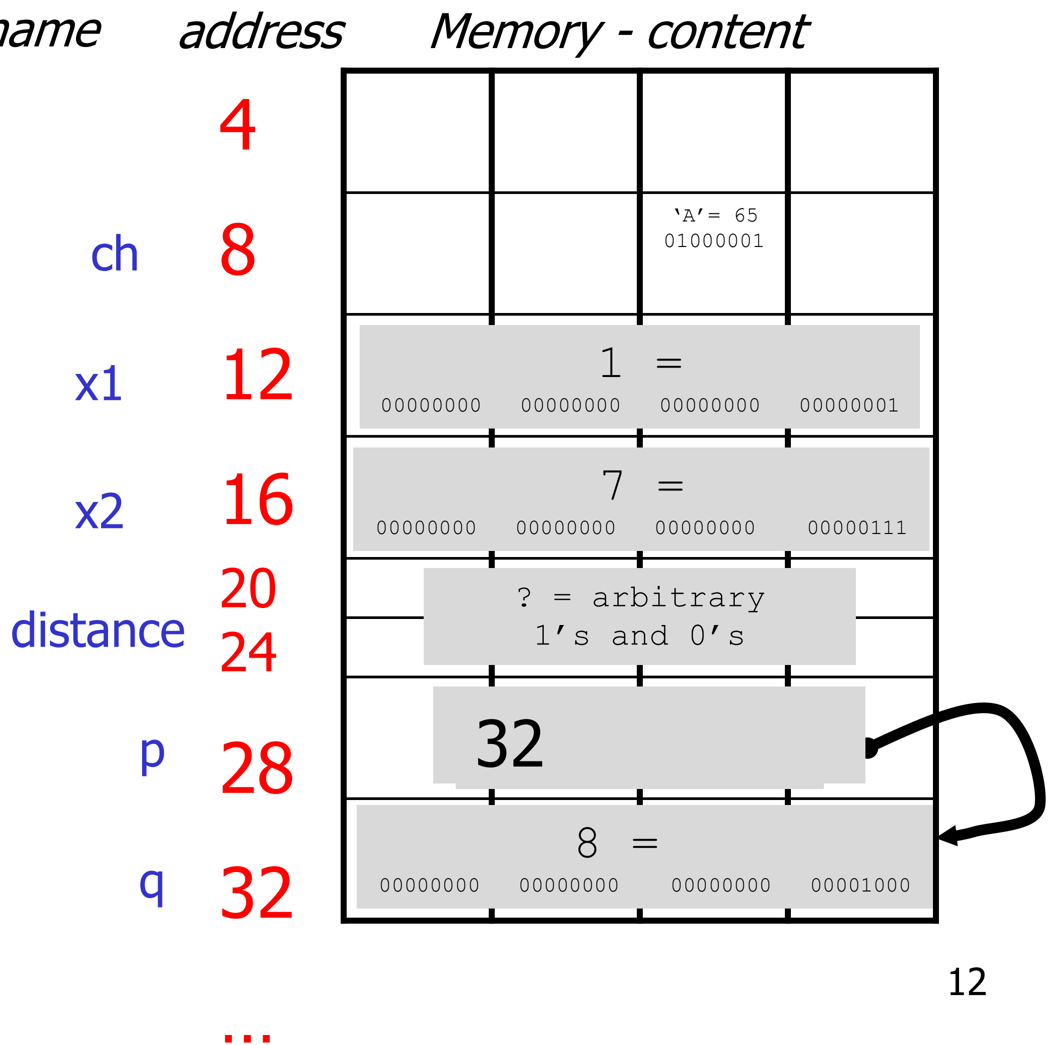
Addresses and Pointers

Recall memory concepts

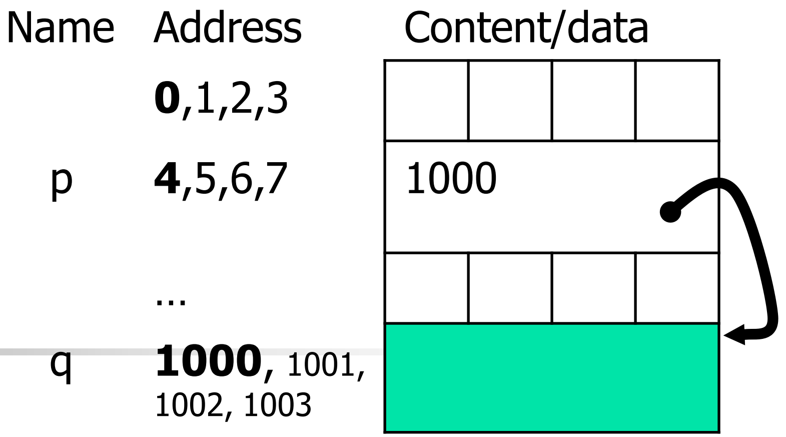
```
char ch=‘A’;

int x1=1, x2=7;

double distance;

int *p;

int q=8;

p = &q;
```

How about

```
char *pc;
double *pd;
```



What is a Pointer?



Name	Address	Content/data
	0,1,2,3	
p	**4**,5,6,7	1000
	…	
q	**1000**, 1001, 1002, 1003	

- A **pointer** is a variable that holds the *address* of a variable (memory location)
- If a variable ***p*** holds the address of variable ***q***, then ***p*** is said to point to ***q***
- If ***q*** is a variable at location 1000 in memory, then ***p*** would have the value 1000 (***q***’s address)

How to declare a pointer variable

- Pointer variables are declared using an asterisk * before the pointer name: `int a, b, *ptr;`
- **`a`** and **`b`** are integer variables
- **`ptr`** is a pointer that can store the address (integer value) of another integer variable

```
double c, *p1;
```

Declaring pointers

- When using the form `int *p, q;` the * operator does not distribute.
- In the above example
  - p is declared to be a pointer to an int var
  - q is declared to be an int var
- If you want both to be pointers, then use the form `int *p, *q;`

Address Operator: **&**

- A variable can be referenced using the **address** operator **&**
- For the example in slide 12:

  &x1 is the **address of x1**

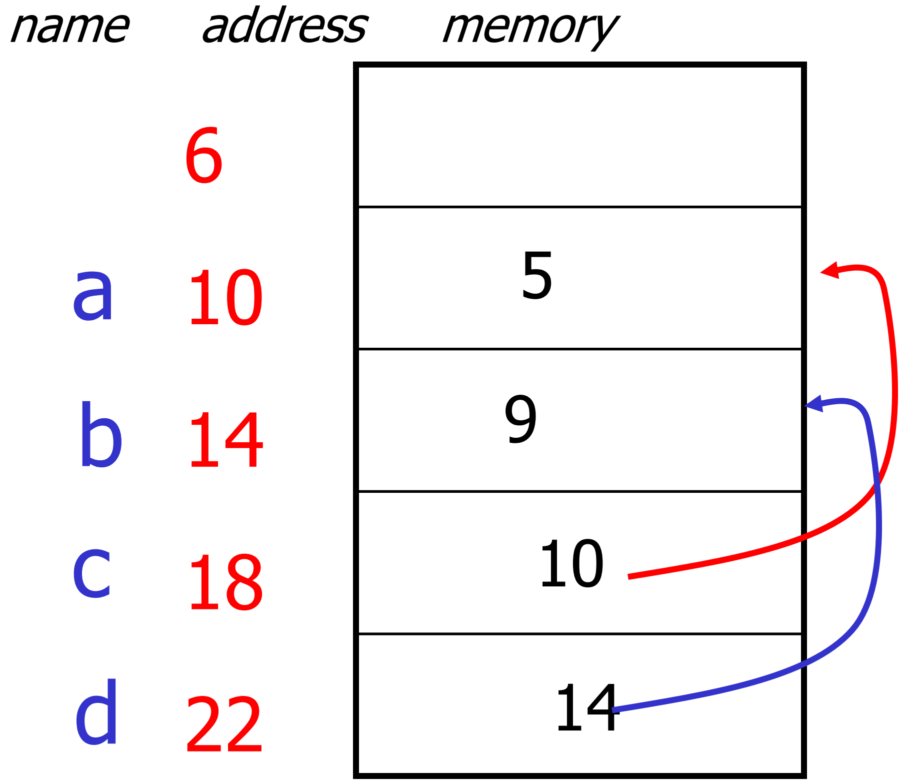
```
printf(“%d”, &x1); → will print 12 (address)
printf(“%d”, x1);  → will print 1 (content)
```

* Value-pointed-to

- * has different meanings in different contexts
  - `int a, b, c;`
  - `a = x * y;` → Multiplication
  - `int *ptr;` → Declare a pointer
  - `ptr = &y;`
  - `a = x *  *ptr;` → Value-pointed-to
- ***** before pointer name in C statements is used as **indirection** or **de-referencing** operator

Example: Pointers, address, indirection

```
int a, b;
int *c, *d;
a = 5;
c = &a;
d = &b;
*d = 9;
print c, *c, &c
print a, b
```

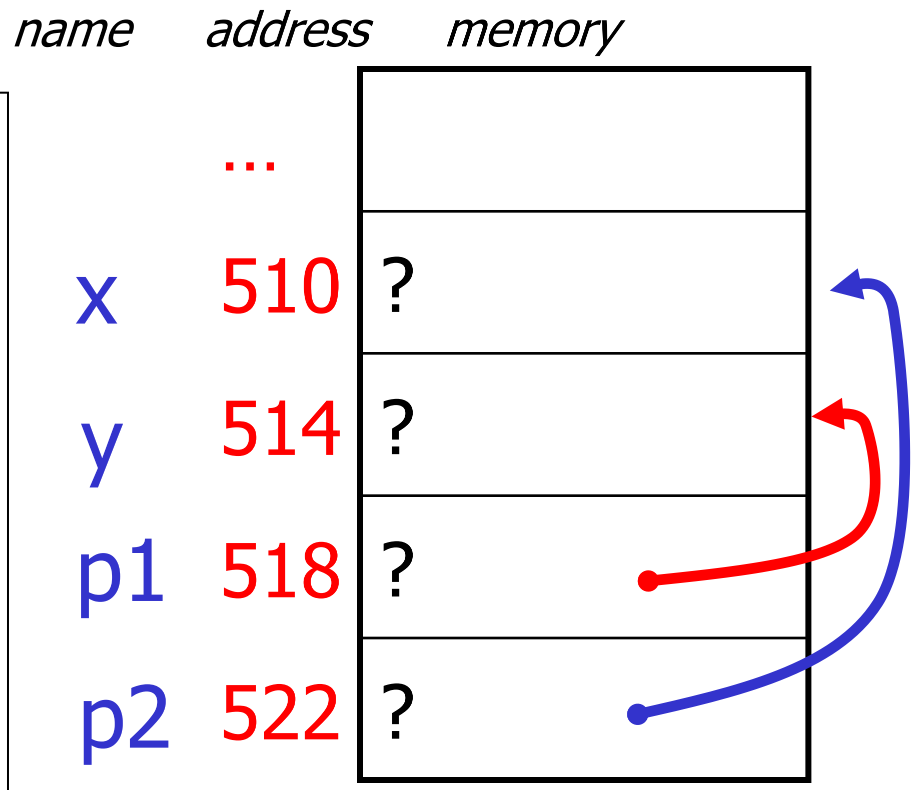


name	*address*	*memory*
	6	
a	10	5
b	14	9
c	18	10
d	22	14

```
c=10   *c=5    &c=18

a=5 b=9
```

Exercise:
Trace the following code

```
int x, y;
int *p1, *p2;
x = 3 + 4;
y = x / 2 + 5;
p1 = &y;
p2 = &x;
*p1 = x + *p2;
*p2 = *p1 + y;
print p1, *p1, &p1
print x, &x, y, &y
```

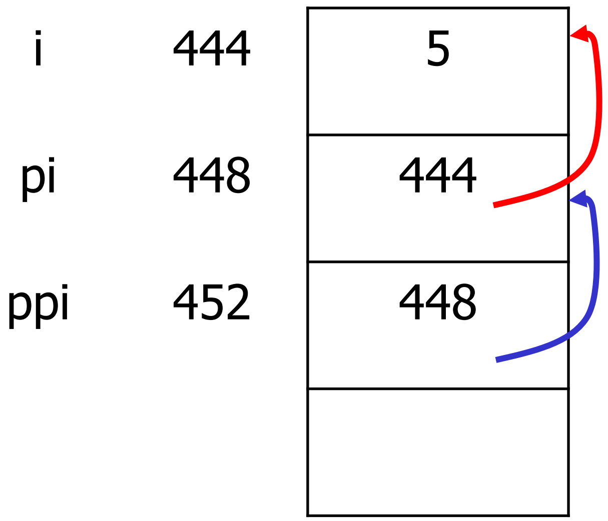


name	*address*	*memory*
	…	
x	510	?
y	514	?
p1	518	?
p2	522	?

Pointers to Pointers

```
int    i;
int    *pi;
int    **ppi;
i=5;
ppi=&pi;
*ppi = &i;
```

Can we have
int ***p;



i	444	5
pi	448	444
ppi	452	448

What will happen now

```
i=10;
*pi=20;
**ppi = 30;
```

```
main()
{
  int *pi;
  …
  f(&pi);
}

int f(int **p)
{
 *pp=New(int);
  …
}
```

MORE ABOUT POINTERS

NULL
ASSIGNMENT,
COMPARISON,
TYPE,
POINTER ARITHMETIC,
FUNCTION PARAMETERS,
POINTERS TO POINTERS,
POINTERS TO FUNCTIONS

NULL pointer

- A pointer can be assigned or compared to the integer zero, or, equivalently, to the symbolic constant **NULL**, which is defined in **<stdio.h>**.
- A pointer variable whose value is NULL is not pointing to anything that can be accessed

Assigning values to a pointer

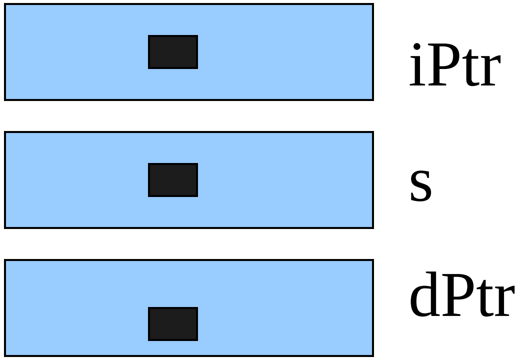
```
int a, b=2;
int *iPtr1, **iPtr2;
```

- the assignment operator (=) is defined for pointers
- the right hand side can be any expression as long as it evaluates to the same type as the left

```
iPtr1  = &a;
iPtr2  = &iPtr1;  // iptr2 = &a;
*iPtr1 = b+3;
*iPtr2 = iPtr1;
```

Pointer Initialization

```
int *iPtr=0;
char *s=0;
double *dPtr=NULL;
```



! When we assign a value to a pointer during it is declaration, we mean to put that value into pointer variable (no indirection)! *Same when calling functions with ptr parameters*

```
int *iPtr=0; //  is same as
     int *iPtr;
     iPtr=0;   //  not like *iPtr = 0;
```

Exercise

Give a memory snapshot after each set of assignment statements

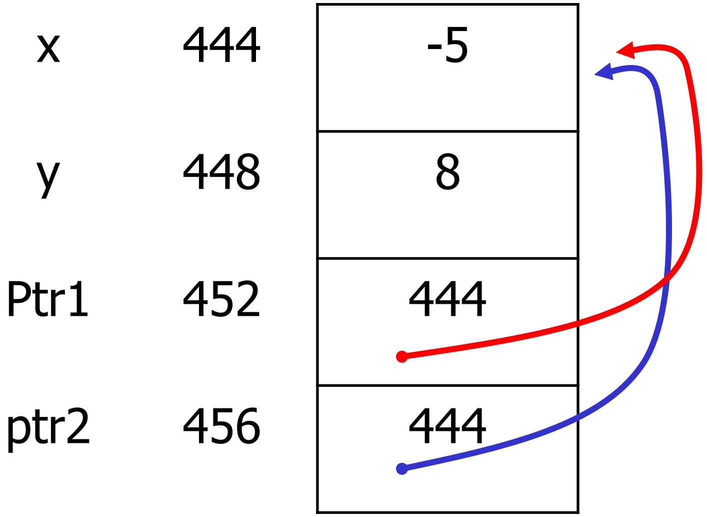
```
int a=3, b=2;
int *ptr1=0, *ptr2=&b;
ptr1   = &a;
a      = *ptr1 * *ptr2;
ptr1   = ptr2;
*ptr1  = a+b;
```

a	100	
b	104	
ptr1	108	
ptr2	112	

Many-to-One Pointing

A pointer can point to only one location at a time, but several pointers can point to the same location.

```
/* Declare and
   initialize variables. */
int x = -5, y = 8;
int *ptr1, *ptr2;

/*  Assign both pointers
    to point to x.  */
ptr1 = &x;
ptr2 = ptr1;
```



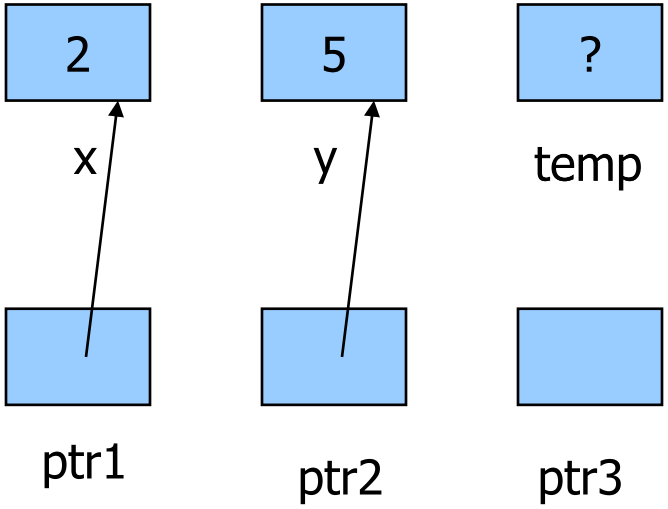
x	444	-5
y	448	8
Ptr1	452	444
ptr2	456	444

The memory snapshot after these statements are executed

Exercise

Show the memory snapshot after the following operations

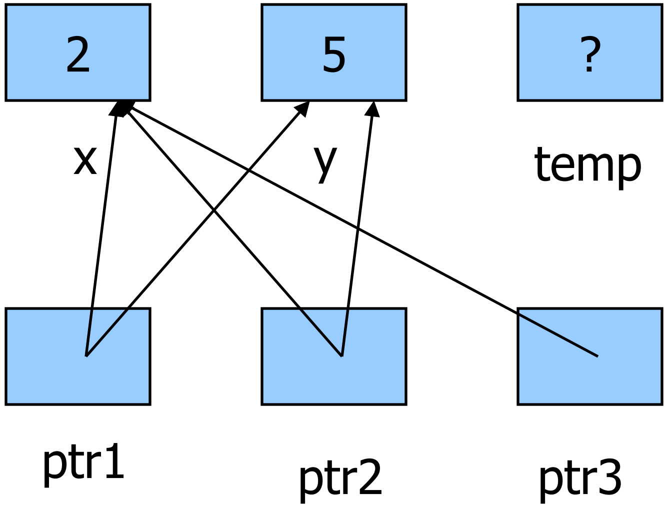
```
int x=2, y=5, temp;
int *ptr1, *ptr2, *ptr3;

// make ptr1 point to x
  ptr1 = &x;

// make ptr2 point to y
  ptr2 = &y;
```



Exercise

Show the memory snapshot after the following operations

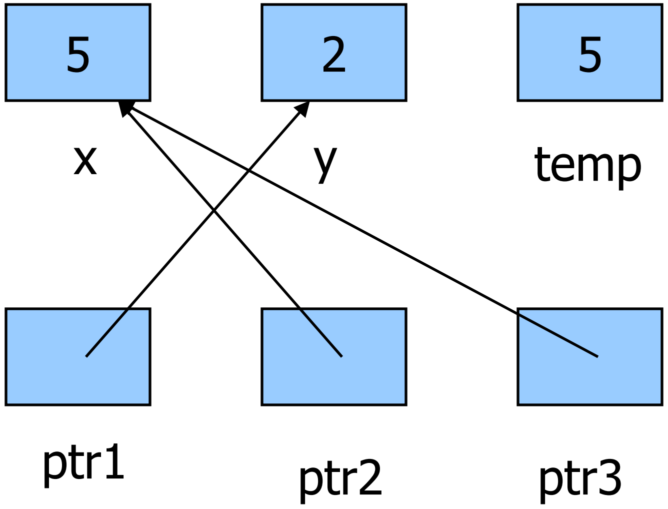
```
// swap the contents of
// ptr1 and ptr2

  ptr3 = ptr1;
  ptr1 = ptr2;
  ptr2 = ptr3;
```



Exercise

Show the memory snapshot after the following operations

```
// swap the values pointed
// by ptr1 and ptr2

  temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
```



Comparing Pointers

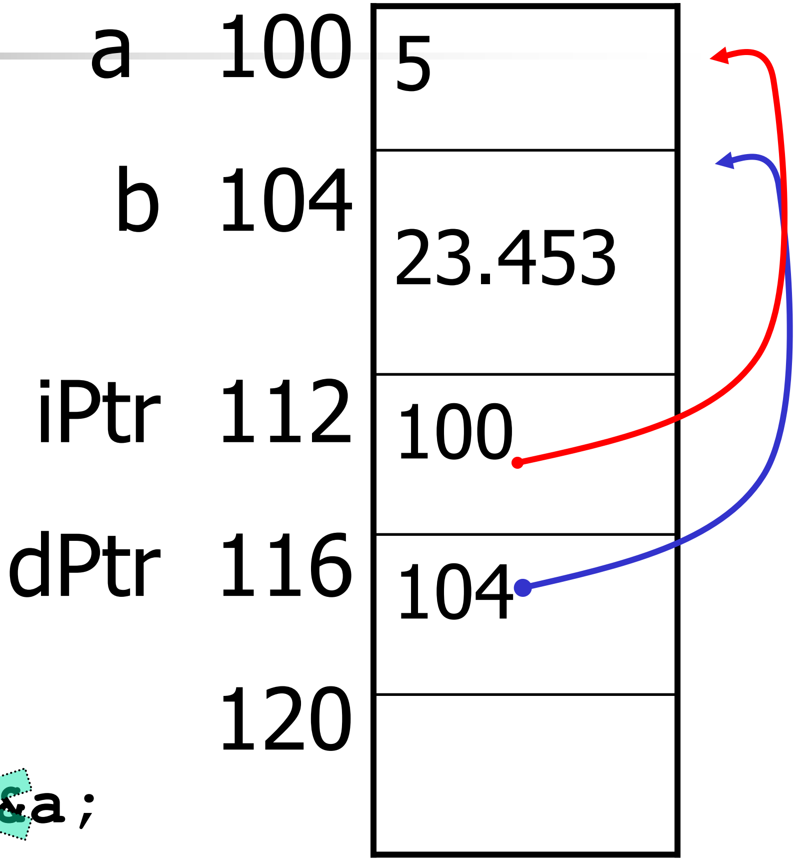
- You may compare pointers using >,<,== etc.
- Common comparisons are:
  - check for null pointer `if (p == NULL)` …
  - check if two pointers are pointing to the same location
    - `if (p == q)` … Is this equivalent to
    - `if (*p == *q)` …
  - Then what is `if (*p == *q)` …
    - compare two values pointed by p and q

Pointer types

- Declaring a pointer creates a variable that is capable of holding an address
- Addresses are integers!
- But, the **base type** we specify in the declaration of a pointer is the type of variable to which this pointer points
  - *!!! a pointer defined to point to an integer variable cannot also point to a float/double variable even though both holds integer address values !!! WHY?*

Example: pointers with different types

```
int a=5;
double b=23.452;
int *iPtr;
double *dPtr;
iPtr = &a;
dPtr = &b; // dP~~tr=&a~~;
```



- the variable iPtr is declared to point to an int
- the variable dPtr is declared to point to a double

Pointer Arithmetic

- Four arithmetic operations are supported
  - +, -, ++, --
  - only integers may be used in these operations
  - Arithmetic is performed relative to the variable type being pointed to

    Example: p++;

    - if p is defined as int *p, p will be incremented by 4 (system dependent)
    - if p is defined as double *p, p will be incremented by 8(system dependent)
    - when applied to pointers, ++ means increment pointer to point to next value in memory
  - MOSTLY USED WITH ARRAYS (as we C later)

Pointers in Function References (!IMPORTANT!)

- In C, function references are call-by-value except when an array name is used as an argument.
  - An array name is the address of the first element
  - Values in an array can be modified by statements within a function
- **To modify a function argument, a pointer to the argument must be passed (call-by-reference)**

```
scanf("%f", &X);
```

This statement specifies that the value read is to be stored at the address of X

- The actual parameter that corresponds to a pointer argument must be an address or pointer.

Call by Value

```
void swap(int a,int b)
{
   int temp;

   temp = a;
   a = b;
   b = temp;

   return;
}
```

```
main()
{
  int x = 2, y = 3;

  printf("%d %d\n“,x,y);
   swap(x,y);
  printf("%d %d\n“,x,y);
}
```

Changes made in function swap are lost when the function execution is over

Call by reference

```
void swap2(int *aptr,
           int *bptr)
{
  int temp;

  temp = *aptr;
  *aptr = *bptr;
  *bptr = temp;

  return;
}
```

```
main()
{
 int x = 2, y = 3;


 printf("%d %d\n",x,y);
  swap2(&x, &y);
 printf("%d %d\n",x,y);
}
```

Changes made in function swap are done on original x and y.
So they do not get lost when the function execution is over

Pointers allow us to get more than one value from a function

- Write a function to compute the roots of quadratic equation $ax^2+bx+c=0$. How to return two roots?

```
void comproots(int a,int b,int c,double *dptr1, double *dptr2)
{
 *dptr1 = (-b - sqrt(b*b-4*a*c))/(2.0*a);
 *dptr2 = (-b + sqrt(b*b-4*a*c))/(2.0*a);
 return;
}
main()
{
     int a,b,c;
     double root1,root2;

     printf("Enter Coefficients:\n");
     scanf("%d %d %d",&a,&b,&c);

     computeroots(a,b,c,&root1,&root2);

     printf("First Root = %lf\n",root1);
     printf("Second Root = %lf\n",root2);
}
```

For complete program, See quadeq.c
Figure 2-1 in the textbook

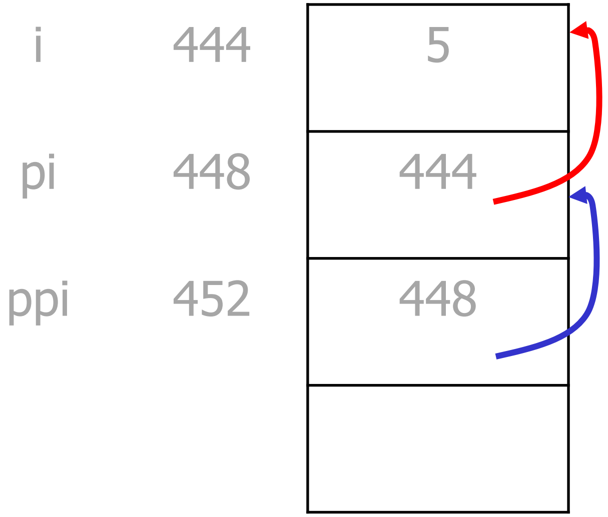
Trace a program

```
main()
{
   int x, y;
   max_min(4, 3, 5, &x, &y);
   printf(“ First: %d  %d”, x, y);
   max_min(x, y, 2, &x, &y);
   printf(“Second: %d  %d”, x, y);
}
void max_min(int a, int b, int c,
             int *max, int *min)
{
    *max = a;
    *min = a;
    if (b > *max) *max = b;
    if (c > *max) *max = c;
    if (b < *min) *min = b;
    if (c < *min) *min = c;
    printf(“F: %d  %d\n”, max, *max);
}
```

name	Addr	Value
x	100	
y	104	
	108	
	112	
	…	
a	400	
b	404	
c	408	
max	412	
min	416	

Pointers to Pointers

```
int    i;
int    *pi;
int    **ppi;
i=5;
ppi=&pi;
*ppi = &i;
```



i	444	5
pi	448	444
ppi	452	448

Can we have
int ***p;

What will happen now

```
i=10;
*pi=20;
**ppi = 30;
```

```
main()
{
  int *pi;
  …
  f(&pi);
}

int f(int **p)
{
 *pp=New(int);
  …
}
```

Exercise: swap pointer variables

- Implement and call a function that can exchange two pointers such that if p1 is pointing v1 and p2 is pointing v2 then (after calling your function) p1 must point to v2 and p2 must point to v1.

```
/* give a prototype for your function here */
 ………………………………….
void main(){
   int v1 = 10, v2 = 25;
   int *p1 = &v1, *p2 = &v2;
   /* call your function here */
         …………………………………………
}
/* implement your function here */
```

Pointers to functions

- int f(); // f is a func returning an int
- int *f(); // f is a func returning ptr to an int
- int (*f)(); // f is a ptr to a func that returns int
- int *(*f)(); // f is a ptr to a func that returns ptr to int
- int *f[]; // f is an array of ptr to int
- int (*f[])(); // f is an array of ptr to functions that return int

```
int f(int) {…}
int (*pf)(int) = &f;
ans = f( 25); ans =(*pf)(25); ans=pf(25);
```

There is a program called `cdecl` that explains an existing C declarations (you may find it on Internet)

Arrays

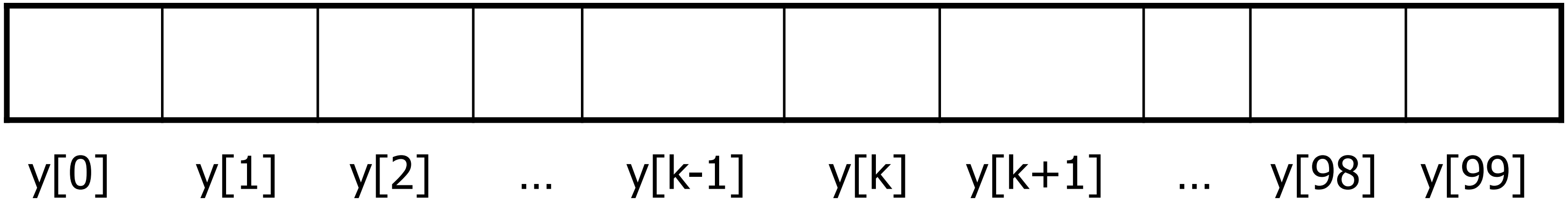
- Collection of individual data values
  - Ordered (first, second…)
  - Homogenous (same type)

One-Dimensional Arrays

- Suppose, you need to store the years of 100 cars. Will you define 100 variables?

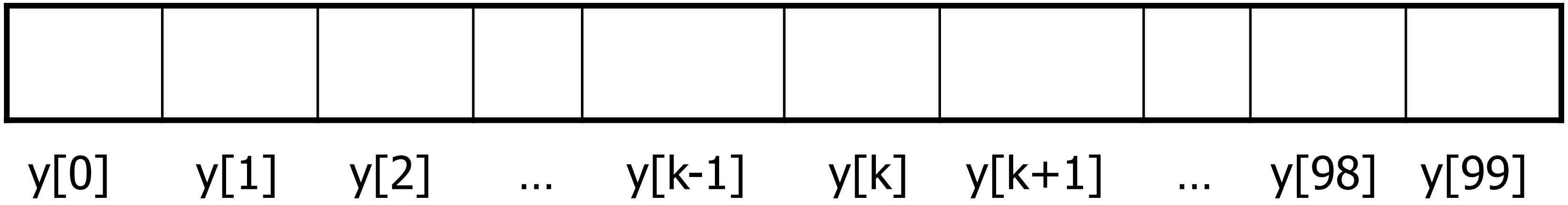
```
int y1, y2,…, y100;
```

- An array is an indexed data structure to represent several variables having the same data type: `int y[100];`

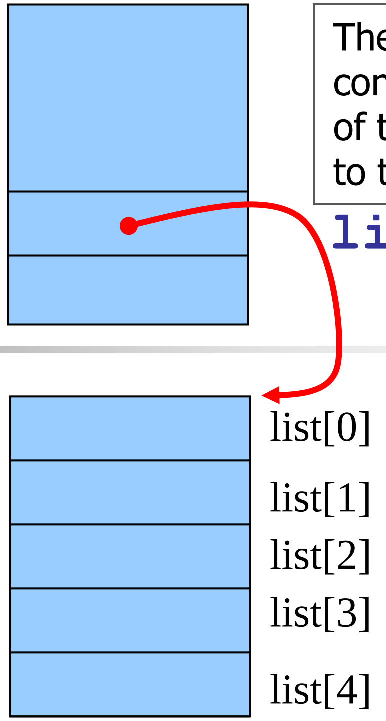


One-Dimensional Arrays (cont’d)

- An *element* of an array is accessed using the **array name** and an **index or subscript**, *for example:* `y[5]` which can be used like a variable
- In C, the subscripts always start with 0 and increment by 1, so `y[5]` is the sixth element
- The name of the array is the **address** of the first element and the subscript is the **offset**



Example



The name of the array is constant showing the **address** of the first element, Or **pointer** to the first element

`list` is a constant

```
#define NE   5
int list[NE];
```

- allocates memory for 5 integer variables
  - subscript of first element is 0
  - subscript of last element is 4

Definition and Initialization

- An array is defined using a declaration statement.

```
data_type array_name[size];
```

  - allocates memory for `size` elements
  - subscript of first element is 0
  - subscript of last element is `size-1`
  - `size` **must be a constant**

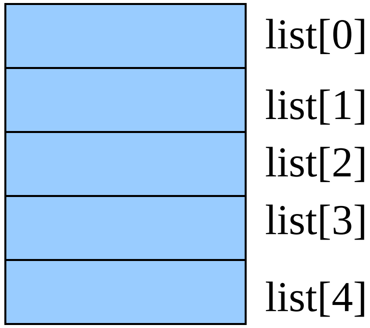
Initializing Arrays

- Arrays can be initialized at the time they are declared.
- Examples:

```
double taxrate[3] ={0.15, 0.25, 0.3};
char list[5] = {‘h’, ’e’, ’l’, ’l’, ’o’};

double vector[100] = {0.0};    /* assigns
                zero to all 100 elements */
int s[] = {5,0,-5}; /*the size of s is 3*/
```

Assigning values to an array

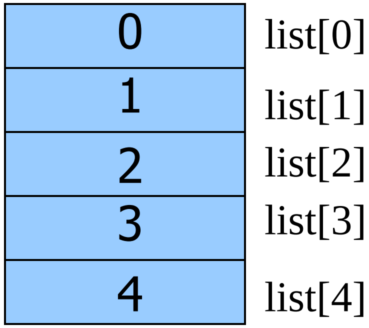
For loops are often used to assign values to an array



list[0]
list[1]
list[2]
list[3]
list[4]

Example:

```
int list[5], i;
for(i=0; i<5; i++){
      list[i] = i;
}
OR
for(i=0; i<=4; i++){
    list[i] = i;
}
```



0 list[0]
1 list[1]
2 list[2]
3 list[3]
4 list[4]

Assigning values to an array

Give a for loop to assign the below values to list

4	list[0]
3	list[1]
2	list[2]
1	list[3]
0	list[4]

```
int list[5], i;
for(i=0; i < 5; i++){
      list[i] = 4-i;
}
```

- ***C does not check bounds on arrays***
- list[6] = 8; /* may give segmentation fault or overwrite other memory locations*/

Exercise

```
int rand_int(int a,int b)
{
   return rand()%(b-a+1) + a;
}
```

- `int data[100], i;`
- Store random numbers [0,99] in data

```
 for (i=0; i<100; i++)
      data[i] = rand() % 100;
```

- Store random numbers [10,109] in data

```
for (i=0; i<100; i++)
        data[i] = (rand() % 100) + 10;
```

OR

```
for (i=0; i<100; i++)
        data[i] = rand_int(10,109);
```

Computations on one-D arrays

Go to Arrays as Function Arguments (62)

Find Maximum

- Find maximum value in data array

```
int data[100], max, i;
for (i=0; i<100; i++)
  data[i] = rand_int(10,109);
max = data[0];
for (i=1; i<100; i++){
  if (data[i] > max)
    max = data[i];
}
printf("Max = %d\n",max);
```

Find average

- Find average of values in data array

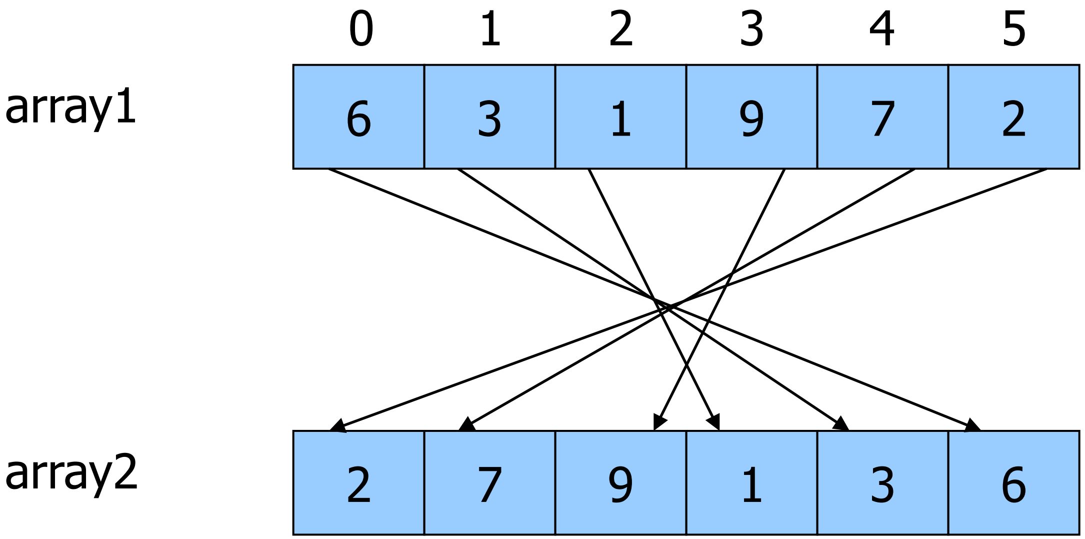
```
int data[100], sum, i, avg;
for (i=0; i<100; i++)
    data[i] = rand_int(10,109);
sum = 0;
for (i=0; i<100; i++){
    sum = sum + data[i];
}
avg = (double)sum/100;
printf(“Avg = %lf\n", avg);
```

Number of elements greater than average

- After finding the average as shown in previous slide, use the following code

```
count = 0;
for (i=0; i<100; i++){
   if (data[i] > avg)
     count++;
}
printf(“%d elements are “
        “greater than avg”, count);
```

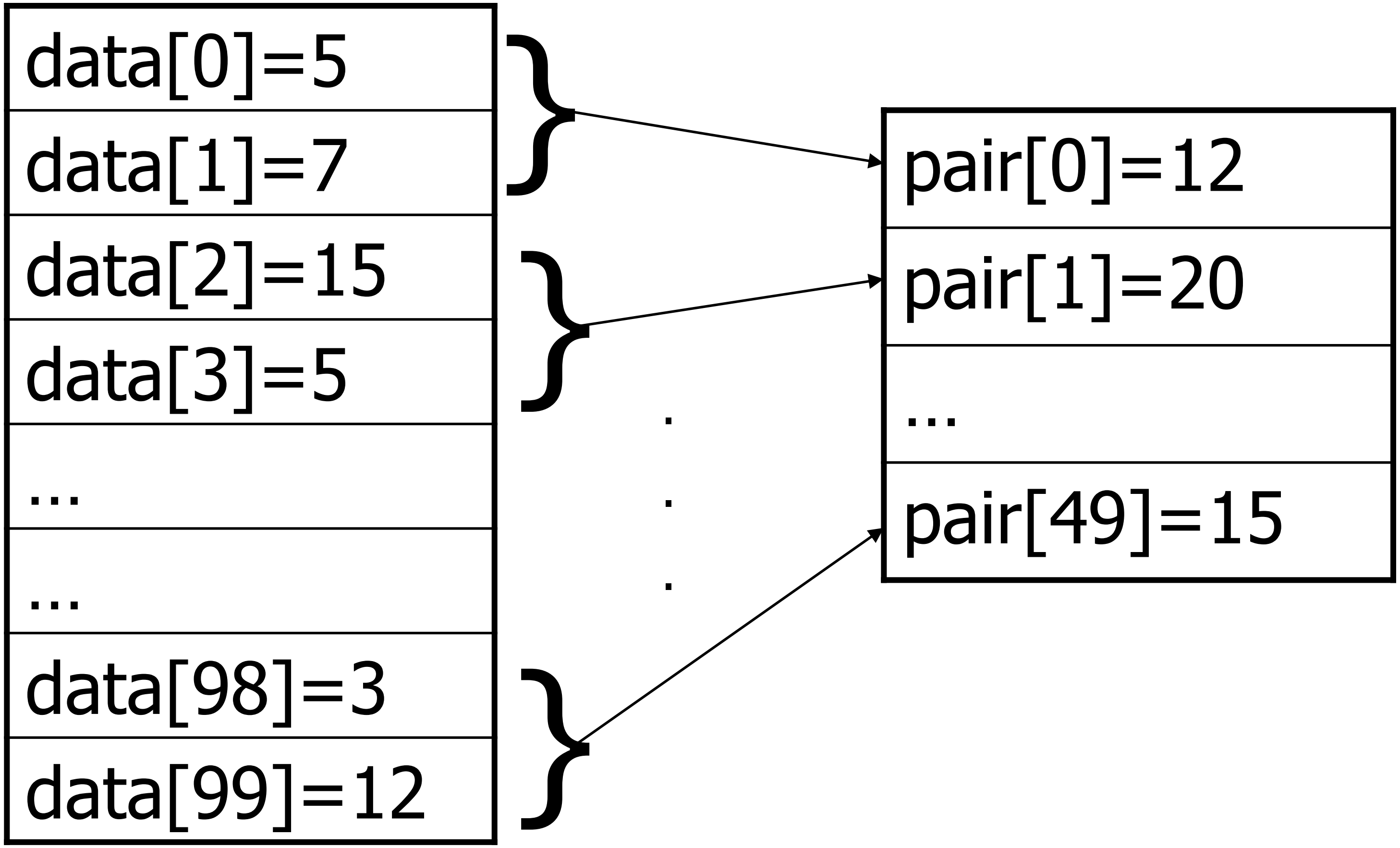
Copy array1 to array2 in reverse order



	0	1	2	3	4	5
array1	6	3	1	9	7	2
array2	2	7	9	1	3	6

Find pair sum

- Find sum of every pair in data and write into pair array



data[0]=5

data[1]=7

data[2]=15

data[3]=5

…

…

data[98]=3

data[99]=12

pair[0]=12

pair[1]=20

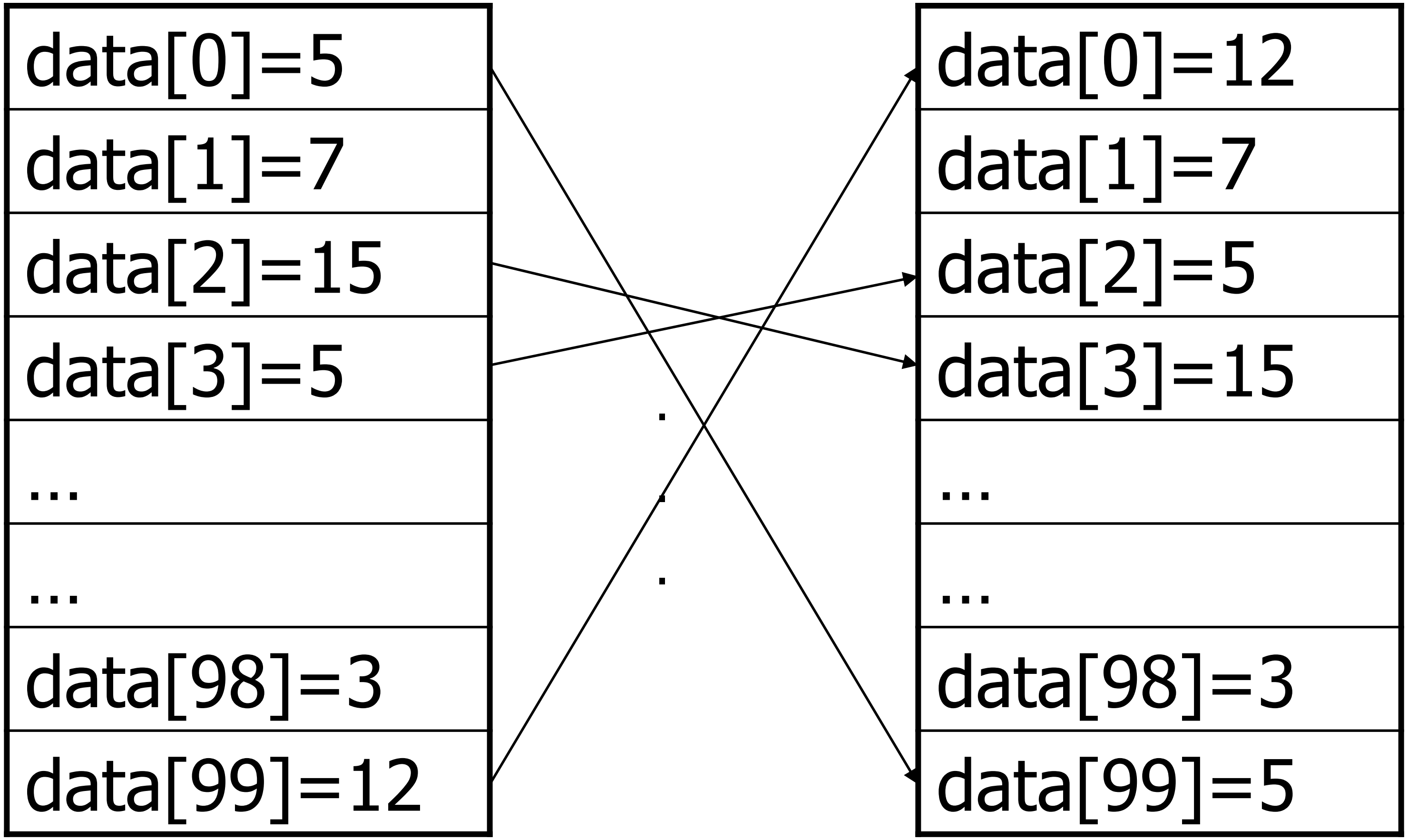
…

pair[49]=15

solution

```
int data[100], pair[50], i;

for (i=0; i<100; i++)
   data[i] = rand_int(1,100);

for (i=0; i<50; i++){
   pair[i]= data[2*i] + data[2*i+1];
}
```

Randomly re-shuffle numbers 30 times



data[0]=5
data[1]=7
data[2]=15
data[3]=5
…
…
data[98]=3
data[99]=12

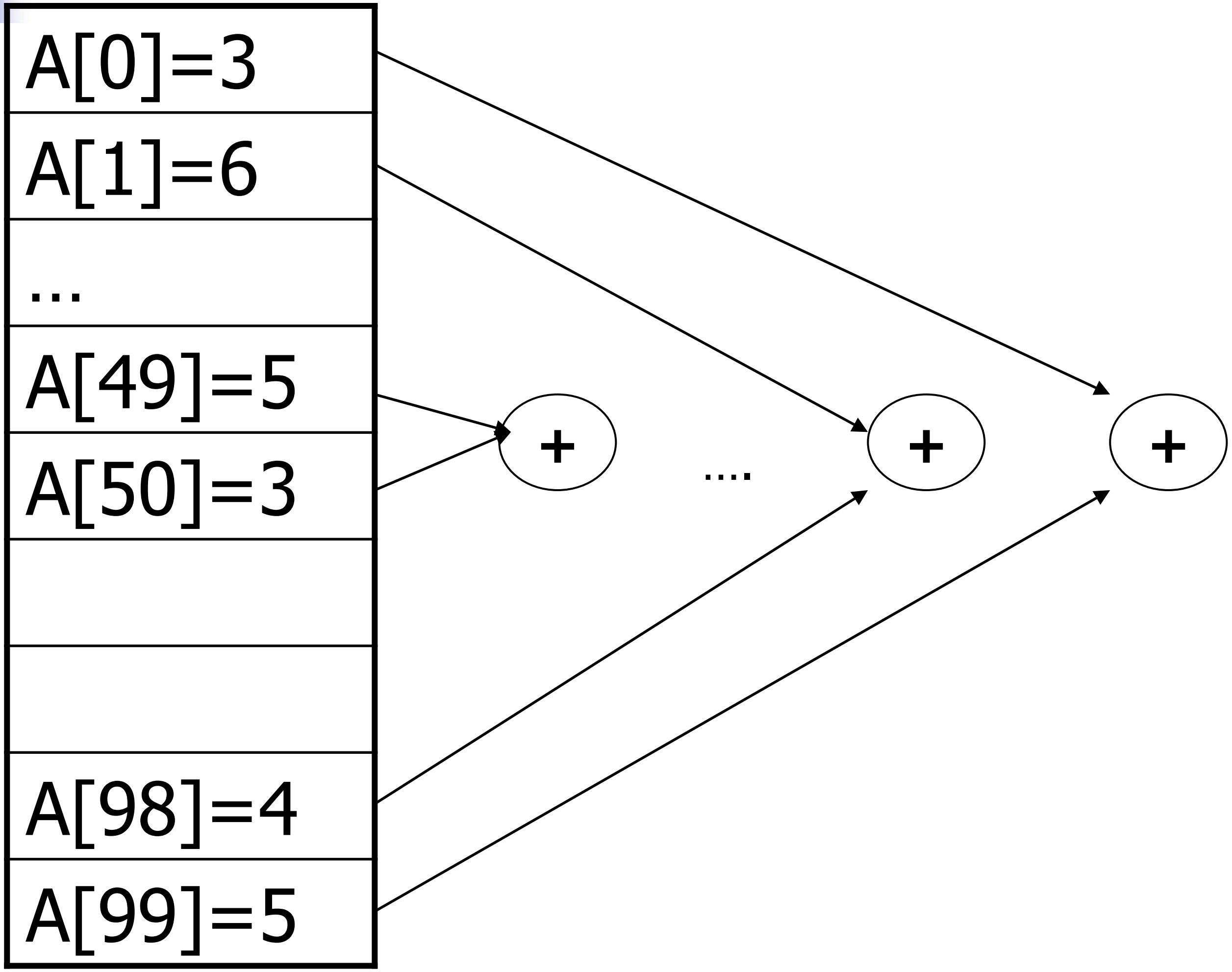
.
.
.

data[0]=12
data[1]=7
data[2]=5
data[3]=15
…
…
data[98]=3
data[99]=5

solution

```
int data[100], i, j, k, tmp;
for (i=0; i<100; i++)
    data[i] = rand_int(1,109);
for (n=0; n<30; n++){
    i=rand_int(0,99);
    j=rand_int(0,99);
    tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```

Print sum of top-bottom pairs



A[0]=3

A[1]=6

…

A[49]=5

A[50]=3

A[98]=4

A[99]=5

+

….

+

+

Random numbers from an irregular range

- Suppose you want to generate 50 random numbers, but you want to chose them uniformly from a set of given numbers like 52 67 80 87 90 95
- Can you do this using arrays?

Arrays as Function Arguments

Function Arguments

Passing individual array elements

- Like a regular variable, we can pass an individual array element (**call-by-value**) or
- We can pass the address of the individual array element (**call-by-reference**) to a function

```
void donothing(int a, int *b)
{ … }

int main(void)
{
  int x=3, y=5, array[5] = {1,2,3,4,5};
  donothing(x, &y);
  donothing(array[2], &array[4]);
                              // Calls donothing(3, address of array +12);
  donothing(x, array); // see next slide
}
```

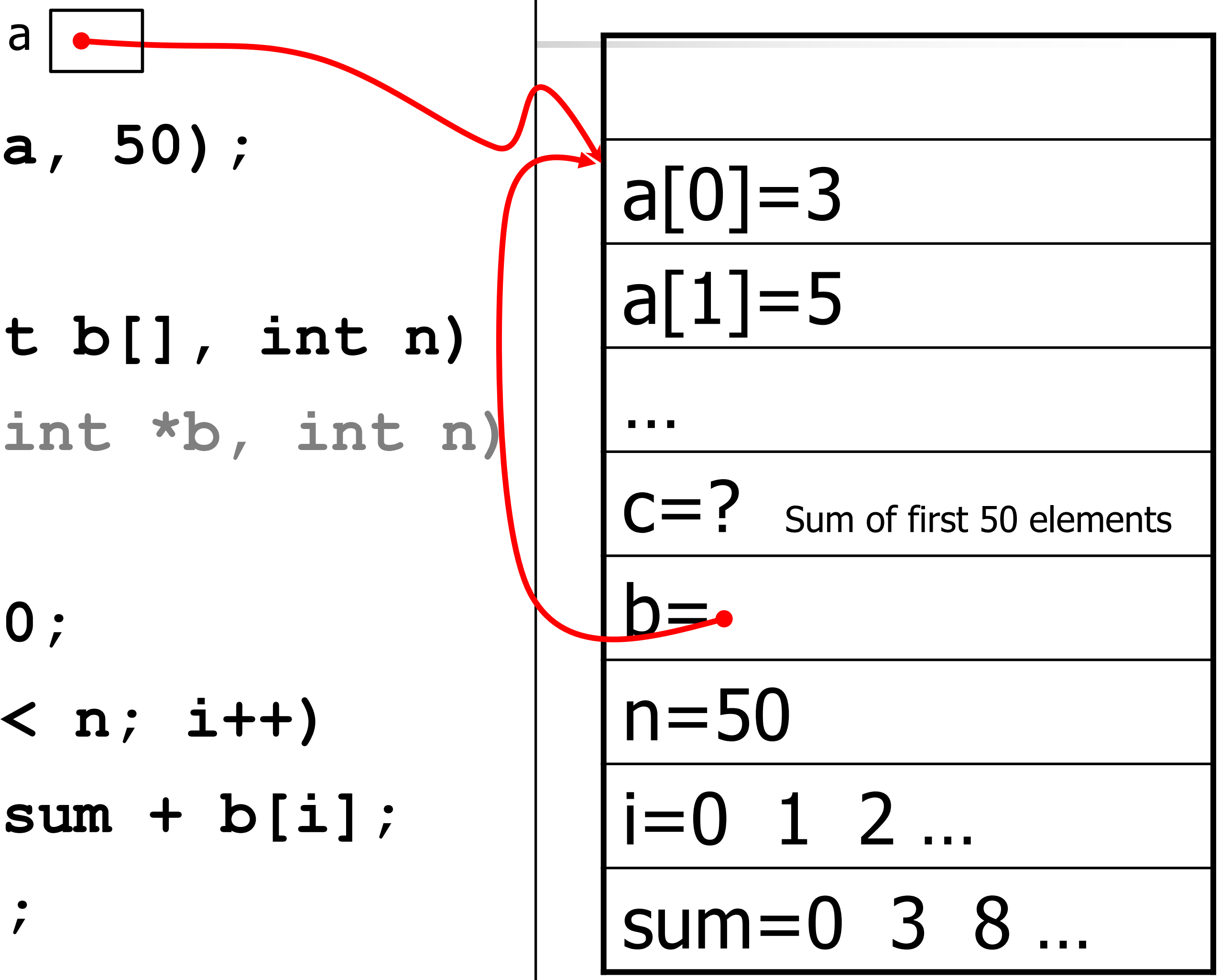
Function Arguments

Passing whole array

- Arrays are always pass by **reference**
  - Modifications to the array are reflected to main program
- The array name is the *address* of the first element (POINTER) but like a constant so it cannot be changed to point another location
- The maximum *size* of the array must be specified at the time the array is declared.
- The *actual number* of array elements that are used will vary, so the *actual number* of the elements in the array is usually passed as another (call-by-value) argument to the function

Exercise

```
main()
{
   int a[100]={3, 5,…};
   int c;
   c = sum_arr(a, 50);
}
int sum_arr(int b[], int n)
//int sum_arr(int *b, int n)
{
    int i, sum=0;
    for(i=0; i < n; i++)
         sum = sum + b[i];
    return(sum);
}
```



a[0]=3

a[1]=5

…

c=? Sum of first 50 elements

b=

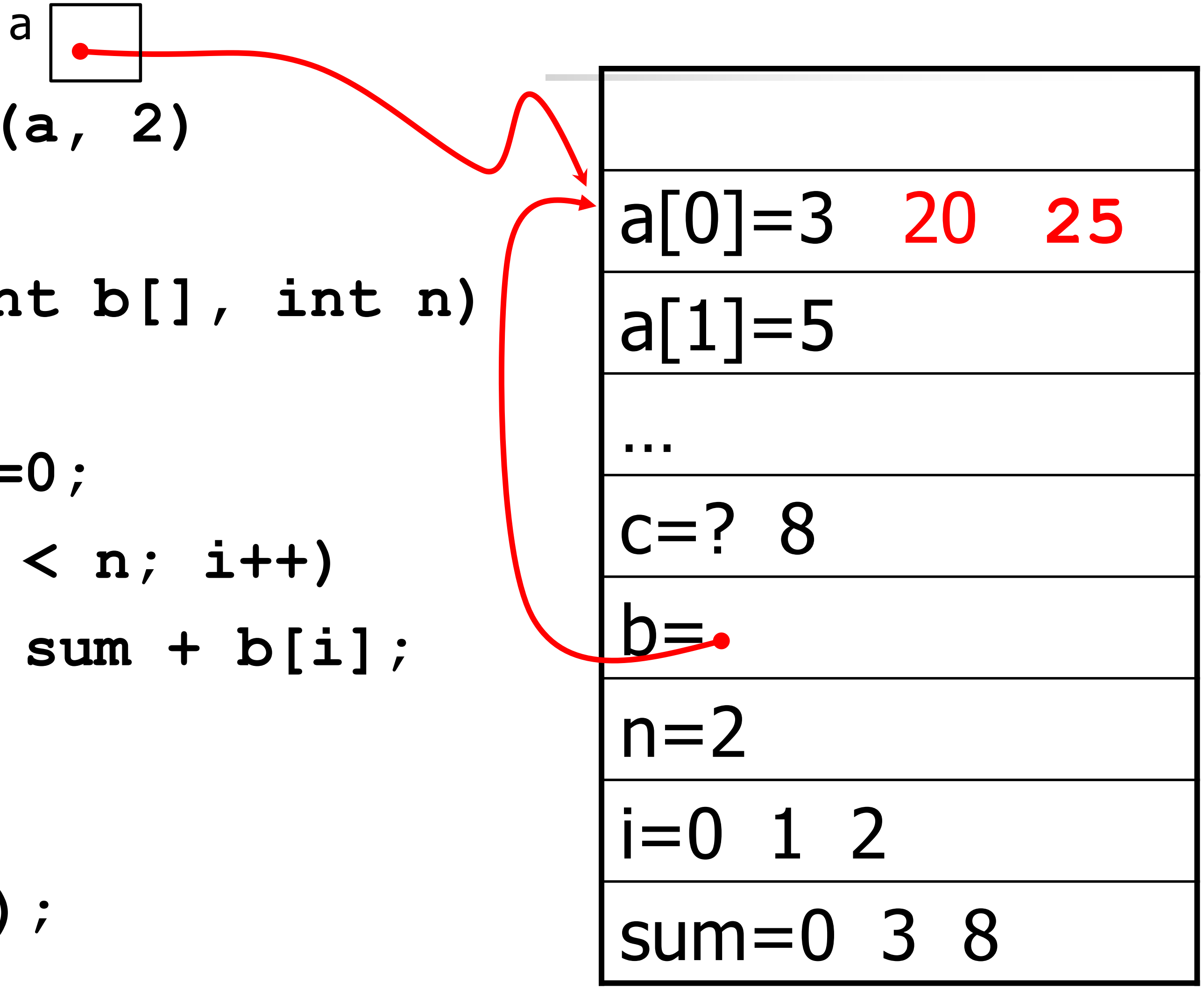
n=50

i=0 1 2 …

sum=0 3 8 …

Exercise

```
main()
{
   int a[100]={3, 5, …};
   int c;
   c = sum_arr(a, 2)
}
int sum_arr(int b[], int n)
{
    int i, sum=0;
    for(i=0; i < n; i++)
         sum = sum + b[i];
    b[0] = 20;
    *b=25;
    return(sum);
}
```



a

a[0]=3 20 25

a[1]=5

…

c=? 8

b=

n=2

i=0 1 2

sum=0 3 8

Exercise

Write a function to find maximum value in the array data

```
int main()
{
  int data[100],i, max;

  for (i=0; i<100; i++)
    data[i] = rand() % 100;
  max = maximum(data,100);
  printf("Max = %d\n",max);
  return(0);
}
```

```
int maximum(int fdata[],
            int n)
{
  int i, fmax;
  fmax = fdata[0];
  for (i=0; i<n; i++)
    if(fdata[i] > fmax)
       fmax = fdata[i];
  return(fmax);
}
```

Exercise

What is the output of the following program?

```
int main()
{
  int data[10];

  for (i=0; i<10; i++)
    data[i] = rand() % 100;

  print(data,10);
  modify(data,10);
  print(data,10);

  return(0);
}
```

```
void print(int pdata[], int n)
{
  int i;
  for (i=0; i<n; i++)
   printf("data[%d]=%d\n",
           i,pdata[i]);
   return;
}
void modify(int fdata[], int n)
{
  int i;
  for (i=0; i<n; i++)
    fdata[i] = 1;
  return;
}
```

More Examples of one dimensional arrays

Go to 2-D Arrays (89)

Trace a program

- Trace the following program and show how variables change in memory.

```
int main()
{
   int x[5]={3, 5, 3, 4, 5};
   int i, j, count;
   for(i = 0; i < 5;  i++){
      count = 1;
      for(j = i+1; j < 5;  j++){
         if (x[i] == x[j]) count++;
      }
      printf(“%d %d \n”, x[i], count);
   }
}
```

x[0]	3
x[1]	5
x[2]	3
x[3]	4
x[4]	5
i	?
j	?
count	?

Search Algorithms

- Unordered list
  - Linear search
  - In a loop compare each element in array with the value you are looking for
- Ordered list
  - Linear search
  - A better solution is known as Binary search (but we will skip it this time, it is like looking for a word in a dictionary)

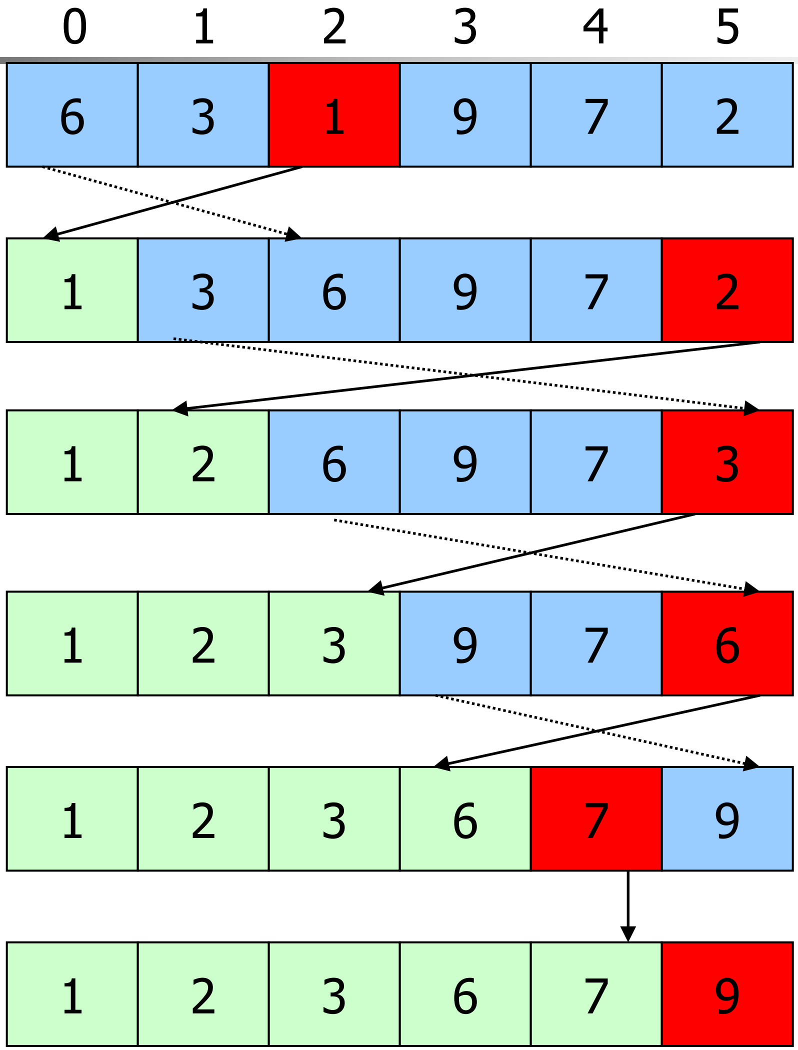
Unordered list – linear search

```
int search1(int x[], int n, int value)
{
   int i;
   for(i=0; i < n; i++) {
       if (x[i]== value)
            return i;
   }
   return(-1);
}
```

Ordered list – linear search

```
int search2(int x[], int n, int value)
{
  int i;
  for(i=0; i < n; i++) {
     if (x[i]== value)
        return i;
     else if (x[i] > value)
        break;
  }
  return(-1);
}
```

Sorting an array



0	1	2	3	4	5
6	3	1	9	7	2
1	3	6	9	7	2
1	2	6	9	7	3
1	2	3	9	7	6
1	2	3	6	7	9
1	2	3	6	7	9

Selection Sort (solution 1)

```
void selection_sort(double x[], int n)
{
  int k,j,m;
  double temp;

  for(k=0; k<=n-2; k++)  {
    m = k;
    for(j=m+1; j<=n-1; j++)
        if(x[j] < x[m])
          m = j;
    temp = x[k];
    x[k] = x[m];
    x[m] = temp
  }
}
```

```
m = find_min_pos(x, n, k);
```

```
swap(x, k, m);
```

Selection Sort (solution 2)

```
void selection_sort(double x[], int n)
{
  int k, m;

  for(k=0; k<=n-2; k++){
     m = find_min_pos(x, n, k);
     swap(x, k, m);
  }
}
```

Selection Sort cont'd

```
int find_min_pos(double fx[], int fn, int fk)
{
   int j;
   int m=fk;
   for (j=m+1; i<=fn-1; j++)
      if (fx[j] < fx[m])
         m = j;
   return(m);
}
```

Selection Sort cont’d

```
void swap(double sx[], int sk, int sm)
{
  double temp;
  temp = sx[sk];
  sx[sk] = sx[sm];
  sx[sm] = temp;
  return;
}
```

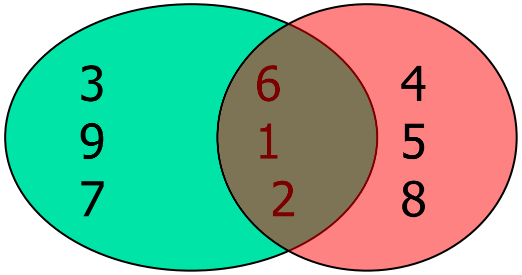
Merge two sorted array

- Assume we have A and B arrays containing sorted numbers
- For example
  - A = { 3, 5, 7, 9, 12}
  - B = {4, 5, 10}
- Merge these two arrays as a single sorted array C, for example
  - C = {3, 4, 5, 5, 7, 9, 10, 12}

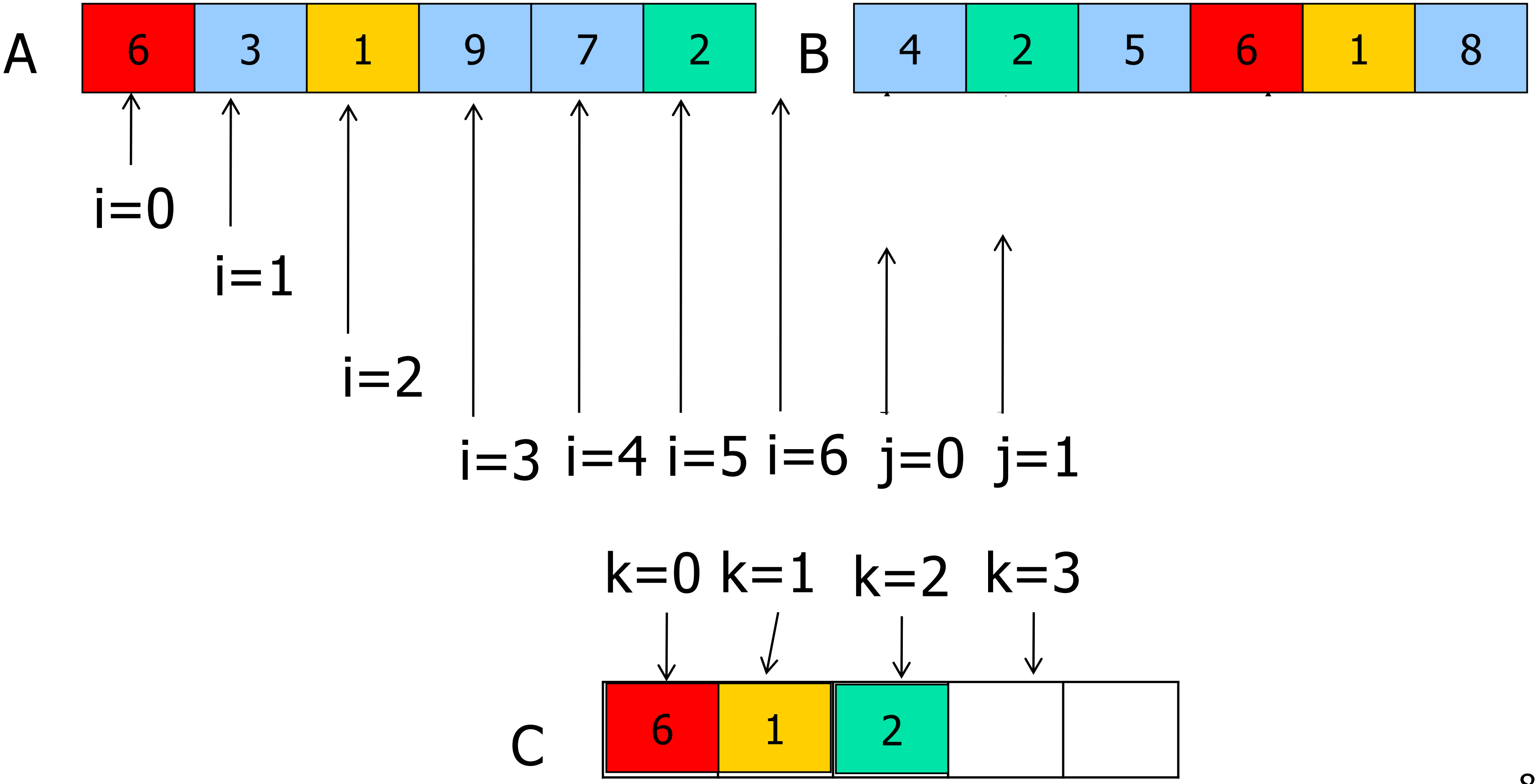
Intersection Set

- Suppose we have two sets (groups) represented by A and B
- E.g., A is the set of students taking Math,
  B is the set of students taking Science.
- Find set C, the intersection of A and B, i.e., students taking both Math and Science

```
For each element ID in A
      Search that ID in B
      if found, put ID into C
```



Use arrays to represent A and B
Hand example



A					
6	3	1	9	7	2

B					
4	2	5	6	1	8

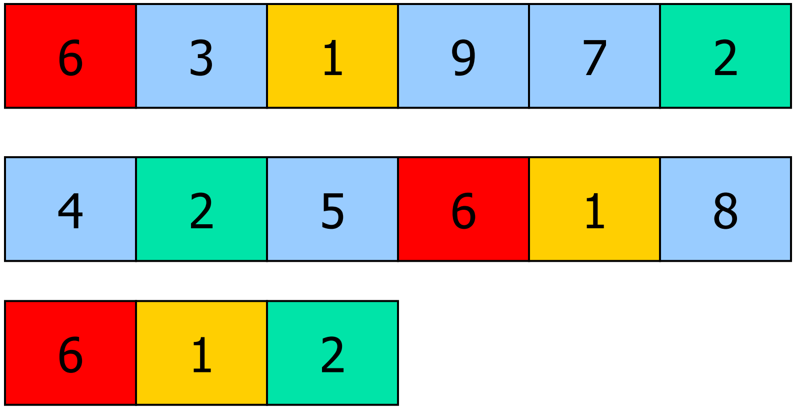
i=0 i=1 i=2 i=3 i=4 i=5 i=6 j=0 j=1

k=0 k=1 k=2 k=3

C				
6	1	2		

Solution

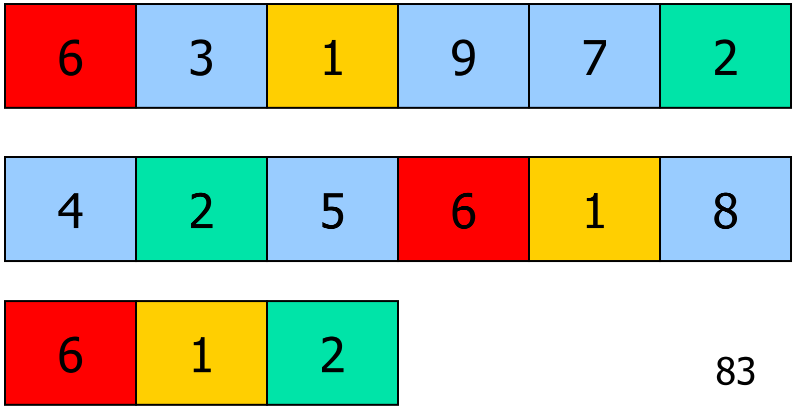
```
int intersection(int A[],int B[],
                 int C[], int n)
{
  int i=0, j=0, k=0;
  for(i=0; i < n; i++) {
     for(j=0; j < n; j++) {
        if (A[i]==B[j]){
           C[k]=A[i];
           k++;
           break;
        }
     }
  }
  return(k);
}
```



6	3	1	9	7	2
4	2	5	6	1	8
6	1	2			

Another Solution

```
int intersection(int A[], int B[],
                 int C[], int n)
{
  int i=0, k=0, elem;
  while (i < n) {
    elem = A[i];
    if(find_count(B,n,elem) == 1) {
         C[k] = elem;
         k = k + 1;
     }
    i = i + 1;
  }
  return(k);
}
```



6 3 1 9 7 2

4 2 5 6 1 8

6 1 2

What if A and B were sorted?

1	2	3	6	7	9

1	2	4	5	6	8

1	2	6

- Will the previous solution work?
- Yes, but we could find intersection set faster!
- How?
- See next slide

```
int sorted_intersection(int A[],int B[],
                        int C[], int n)
{
  int i=0, j=0, k=0;
  while( i < n && j < n ) {
    if (A[i]==B[j]){
           C[k]=A[i];
           k++; i++; j++;
    } else if (A[i] < B[j]) {
           i++;
    } else {               /* A[i] > B[j] */
           j++;
    }
  }
  return(k);
}
```

Exercise: union or difference

- As in previous example suppose two sets are given as arrays.
- Find union and difference
- For example
  - A={3,4,5} and B={2,3,5,7}
  - A U B = {2,3,4,5,7}
  - A – B = {4}
  - B – A = {2,7}

Exercise: Histogram

- Suppose somehow we read **npts** integer numbers from a file into an array declared by **int x[100]**. We know that all the values are integers between 0 and 10. Now suppose we want to find how many 0’s ,1’s, …, 10’s exist in the array **x**.
- Write a function that will take **x** and **npts** as the parameters and prints out the number of 0’s ,1’s, …, 10’s that exist in the array **x**

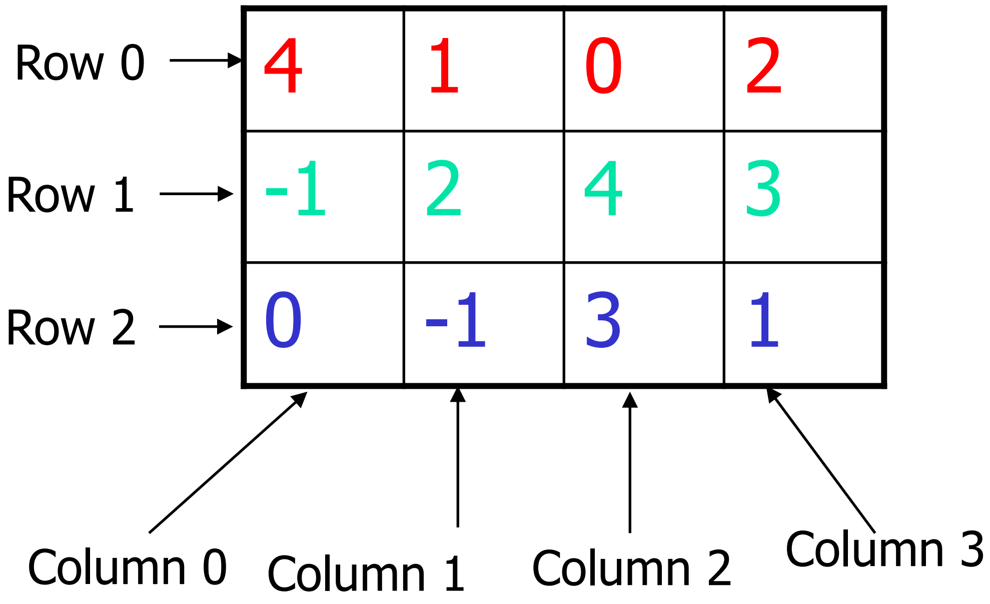
solution

```
void my_function(int x[], int npt)
{
    int i, hist[11]={0};

    for(i=0; i < npt; i++)
        hist[x[i]]++;

    for(i=0; i < 11; i++)
        printf(“%d appears %d times\n”,
                    i, hist[i]);
    return;
}
```

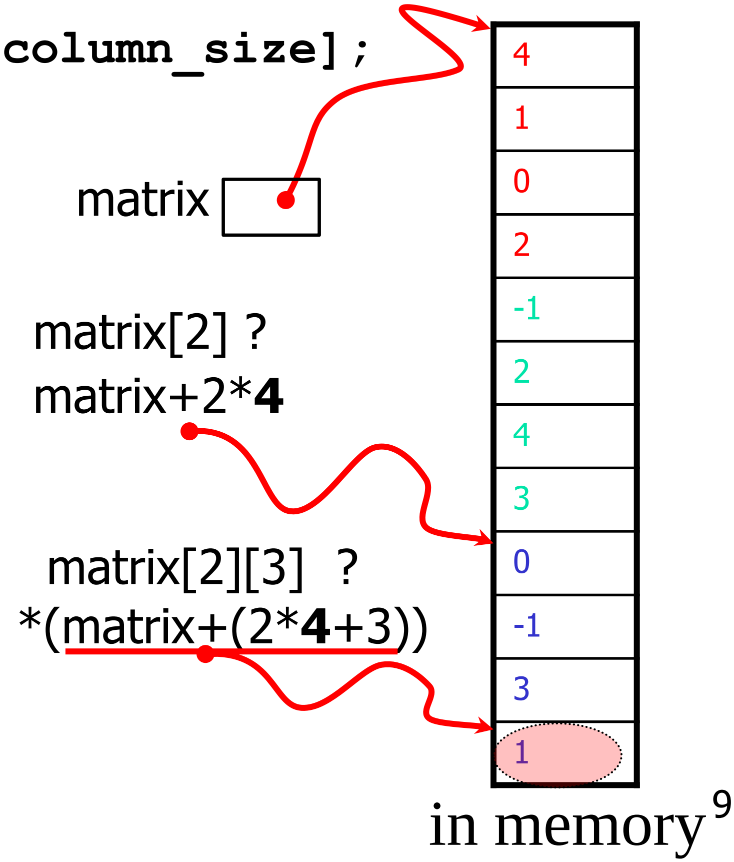
Matrices (2D-array)

Matrices (2D-array)

- A matrix is a set of numbers arranged in a grid with rows and columns.
- A matrix is defined using a type declaration statement.
  - `datatype array_name[row_size][column_size];`
  - `int matrix[3][4];`



	Column 0	Column 1	Column 2	Column 3
Row 0	4	1	0	2
Row 1	-1	2	4	3
Row 2	0	-1	3	1



matrix

matrix[2] ?

matrix+2***4**

matrix[2][3] ?

*(matrix+(2***4**+3))

Accessing Array Elements

```
int matrix[3][4];
```

- `matrix` has 12 integer elements
- `matrix[0][0]` element in first row, first column
- `matrix[2][3]` element in last row, last column
- `matrix` is the address of the first element
- `matrix[1]` is the address of the Row 1
- `matrix[1]` is a one dimensional array (Row 1)

Initialization

```
int x[4][4] = {  {2, 3, 7, 2},
                 {7, 4, 5, 9},
                 {5, 1, 6, -3},
                 {2, 5, -1, 3}};
int x[][4] = {   {2, 3, 7, 2},
                 {7, 4, 5, 9},
                 {5, 1, 6, -3},
                 {2, 5, -1, 3}};
```

Initialization

```
int i, j, matrix[3][4];
for (i=0; i<3; i++)
   for (j=0; j<4; j++)
      matrix[i][j] = i;
```

j

i

	0	1	2	3
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2

```
matrix[i][j] = j;
```

j

i

	0	1	2	3
0	0	1	2	3
1	0	1	2	3
2	0	1	2	3

Exercise

- Write the nested loop to initialize a 2D array as follow

0	1	2
1	2	3
2	3	4
3	4	5

```
int i, j, x[4][3];
for(i=0; i<4; i++)
   for(j=0; j<3; j++)
      x[i][j] = i+j;
```

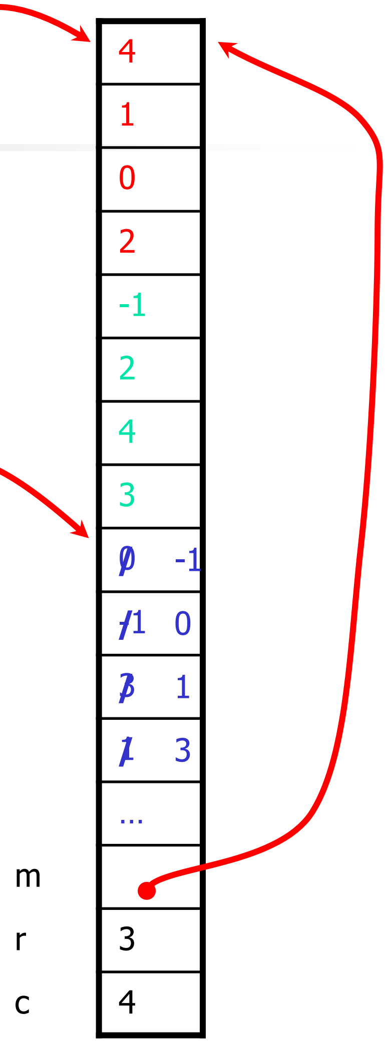
2-Dim Arrays as Arguments to Functions

```
int i, j, matrix[3][4];
for (i=0; i<3; i++)
   for (j=0; j<4; j++)
       matrix[i][j] = …;
print_m(matrix, 3, 4);

selection_sort_int(matrix[2], 4);
```

```
void print_m(int m[][4],
             int r, int c)
{
  int i,j;
  for (i=0; i < r; i++) {
    for (j=0; j < c; j++)
      printf("%2d ", m[i][j]);
    printf("\n");
  }
  printf("\n");
}
```

```
*(m+i*4+j);
```

```
 4  1  0  2
-1  2  4  3
 0 -1  3  1
```



Computations on 2D arrays

Go to Strings:
Array of Characters (115)

Max in 2D

- Find the maximum of *int matrix[3][4]*

```
int max = matrix[0][0];
for (i=0; i<3; i++)
  for (j=0; j<4; j++)
    if (matrix[i][j] > max)
        max = matrix[i][j];
```

	0	1	2	3
0	0	1	0	2
1	-1	2	4	3
2	0	-1	3	1

Find a value in 2D

- Find the number of times *x* appears in *int matrix[3][4]*

```
int count = 0;
for (i=0; i<3; i++)
   for (j=0; j<4; j++)
      if (matrix[i][j] == x)
          count = count + 1;
```

	0	1	2	3
0	0	1	0	2
1	-1	2	4	3
2	0	-1	3	1

Matrix sum

- Compute the addition of two matrices

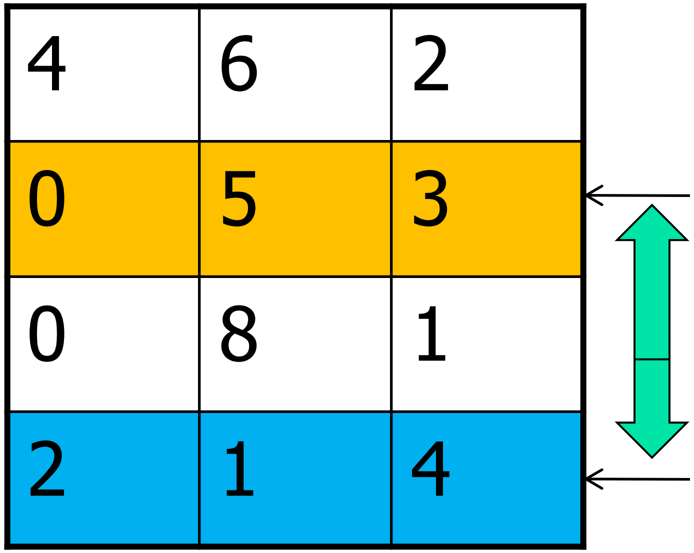
$$
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 1 & 0 & 2 \\
1 & -1 & 2 & 4 & 3 \\
2 & 0 & -1 & 3 & 1
\end{array}
\;+\;
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 3 & -1 & 3 & 1 \\
1 & 1 & 4 & 2 & 0 \\
2 & 2 & 1 & 1 & 3
\end{array}
\;=\;
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 \\
\hline
0 & 3 & 0 & 3 & 3 \\
1 & 0 & 6 & 6 & 3 \\
2 & 2 & 0 & 4 & 4
\end{array}
$$

solution

```
int matrix1[3][4],
    matrix2[3][4],
    sum[3][4];
// initialize matrix1 and matrix2

for (i=0; i<3; i++)
   for (j=0; j<4; j++)
        sum[i][j]= matrix1[i][j]+matrix2[i][j];
```

Exchange Two Rows

```
int  A[4][3]={…};
int i, j, k, temp;
i = …;
j = …;
/* excahnge ith and jth rows of A */
```



4	6	2
0	5	3
0	8	1
2	1	4

4	6	2
2	1	4
0	8	1
0	5	3

Transpose

a

1	5	3
4	2	6

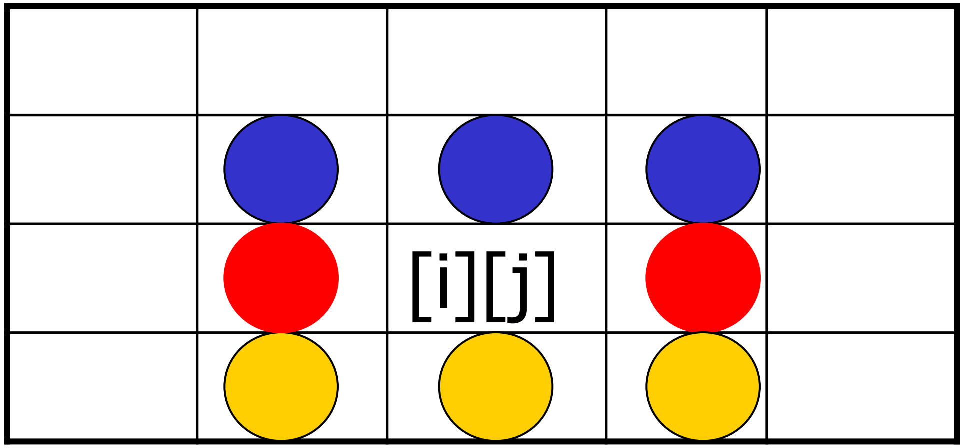
b

1	4
5	2
3	6

```
void transpose(int a[NROWS][NCOLS],
               int b[NCOLS][NROWS])
{
  /*  Declare Variables.   */
  int i, j;

  /*  Transfer values to the
      transpose matrix.   */
  for(i=0; i<NROWS; i++) {
     for(j=0; j<NCOLS; j++) {
          b[j][i] = a[i][j];
     }
  }
  return;
}
```

mine sweeper

- int m[4][5] = {{…}, …};
- If m[i][j] is 0, there is no mine in cell m[i][j]
- If m[i][j] is 1, there is a mine in cell m[i][j]
- Print the number of mines around cell m[i][j]



Solution (1) - incomplete

```
count=0;
if(  m[i-1][j-1]      )     count++;
if(  m[i-1][j]        )     count++;
if(  m[i-1][j+1]      )     count++;
if(  m[i][j-1]        )     count++;
if(  m[i][j+1]        )     count++;
if(  m[i+1][j-1]      )     count++;
if(  m[i+1][j]        )     count++;
if(  m[i+1][j+1]      )     count++;
```

What if [i][j] is not in the middle?

[i][j]				
		[i][j]		[i][j]

Solution (1) – complete

NR: is number of rows

NC: number of columns

```
count=0;
if( i>0 && j>0 &&        m[i-1][j-1] ) count++;
if( i>0 &&               m[i-1][j]   ) count++;
if( i>0 && j<NC-1 &&     m[i-1][j+1] ) count++;
if( j>0 &&               m[i][j-1]   ) count++;
if( j<NC-1 &&            m[i][j+1]   ) count++;
if( i<NR-1 && j>0 &&     m[i+1][j-1] ) count++;
if( i<NR-1 &&            m[i+1][j]   ) count++;
if( i<NR-1 && j<NC-1 &&  m[i+1][j+1] ) count++;
```

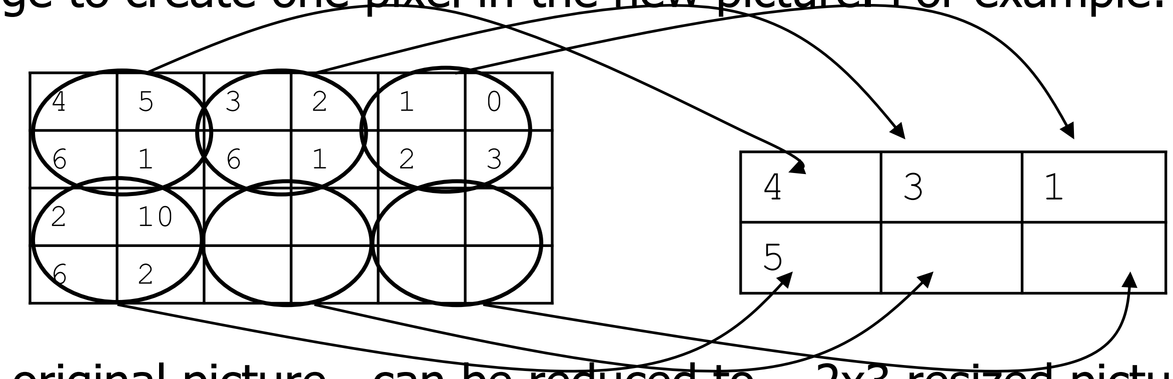
Solution (2)

NR: is number of rows, NC: number of columns

```
int r, c, count=0;
for(r=i-1; r <= i+1; r++) {
  if (r < 0 || r >= NR) continue;
  for(c=j-1; c <= j+1; c++) {
    if (c < 0 || c >= NR) continue;
    if (c == c) continue;
    if( m[r][c]) count++;
    }
}
```

Example: Resize a picture

- A b&w picture is usually represented using a two-dimensional array, where each element (called pixel) of the array is an integer number denoting the intensity of the light at a given pixel. Suppose we have a b&w picture with the size of 100x200 pixels and we want to reduce its size to 50x100 pixels.
- For this, we may consider 4 pixels from the original picture and take their average to create one pixel in the new picture. For example:



4	5	3	2	1	0
6	1	6	1	2	3
2	10				
6	2				

4	3	1
5		

4x6 original picture   can be reduced to   2x3 resized picture

- Write a function that takes **orig[100][200]** and **resized[50][100]** as parameters and determines the values in resized picture as described above.

Matrix multiplication

double a[3][2], b[2][4], c[3][4];

- Find c = a * b;

$$\begin{bmatrix} 3 & 4 \\ 5 & 2 \\ 1 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 7 & 1 \\ 4 & 5 & 6 & 8 \end{bmatrix} = \begin{bmatrix} 22 & 29 & 45 & 35 \\ 18 & 40 & 47 & 21 \\ 26 & 33 & 43 & 49 \end{bmatrix}$$

3*2 + 4*4=22

3*3 + 4*5=29

3*7 + 4*6=45

3*1 + 4*8=35

5*2 + 2*4=18

5*3 + 2*5=40

5*7 + 2*6=47

5*1 + 2*8=21

1*2 + 6*4=26

1*3 + 6*5=33

1*7 + 6*6=43

1*1 + 6*8=49

Matrix Multiplication cont’d

$$\begin{array}{c|cc} i & & \\ \hline 0 & 3 & 4 \\ 1 & 5 & 2 \\ 2 & 1 & 6 \end{array} \times \begin{array}{cccc} & j & & \\ 0 & 1 & 2 & 3 \\ \hline 2 & 3 & 7 & 1 \\ 4 & 5 & 6 & 8 \end{array} = \begin{array}{c|cccc} & & j & & \\ i & 0 & 1 & 2 & 3 \\ \hline 0 & 22 & 29 & 45 & 35 \\ 1 & 18 & 40 & 47 & 21 \\ 2 & 26 & 33 & 43 & 49 \end{array}$$

$$i=0 \; \begin{array}{cc} 3 & 4 \\ \hline \multicolumn{2}{c}{k} \end{array} \times \begin{array}{c|c} j=0 & \\ \hline 2 & k \\ 4 & \end{array} =$$

```
c[i][j] =
  a[i][k=0]*b[k=0][j] +
  a[i][k=1]*b[k=1][j]
```

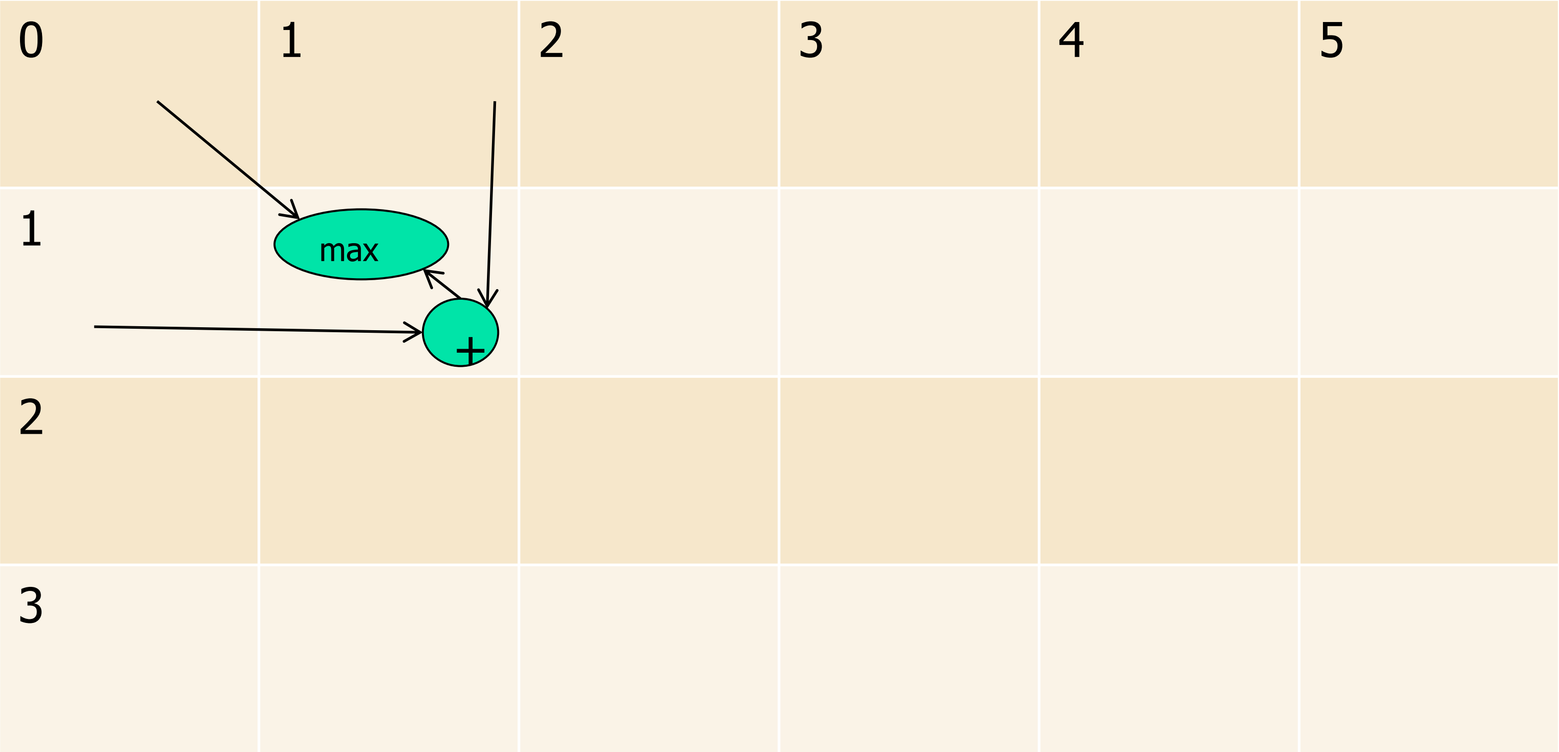
Matrix Multiplication cont'd

```
#define N  3
#define M  2
#define L  4
void matrix_mul(a[N][M], int b[M][L], int c[N][L])
{
  int i, j, k;
  for(i=0; i < N; i++) {
     for(j=0; j < L; j++) {
         c[i][j] = 0;
         for(k=0; k < M; k++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
         }
     }
  }
  return;
}
```

Exercise: Find possible values for cell s[i][j] in Sudoku

4			1	7		6	8	
		8		4		3		7
	5		3				4	
7					4		1	
3		1	9	[i][j]	7	4		6
	6		8					9
	3				5		7	
8		2		6		9		
	4	5		1	8			3

Exercise: Dynamic programming

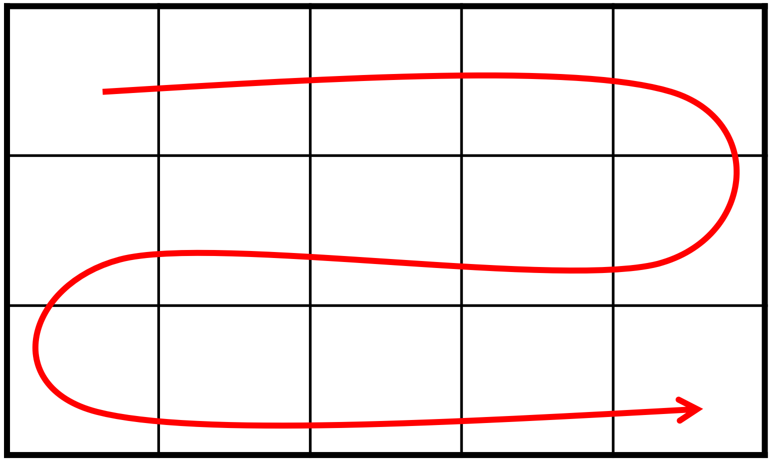


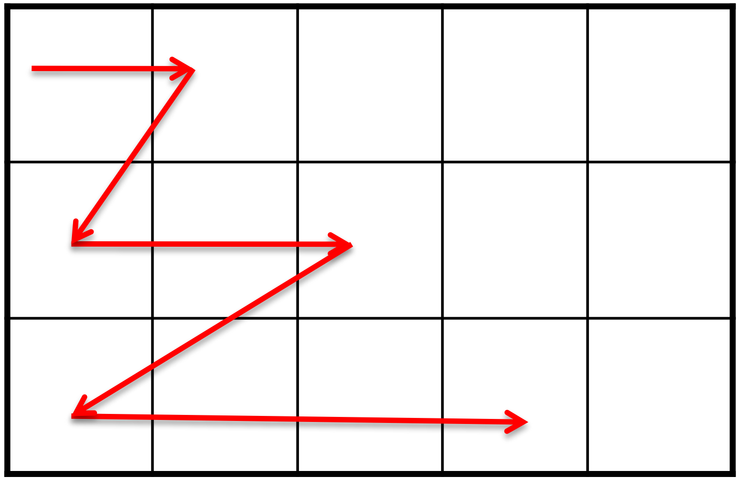
0	1	2	3	4	5
1					
2					
3					

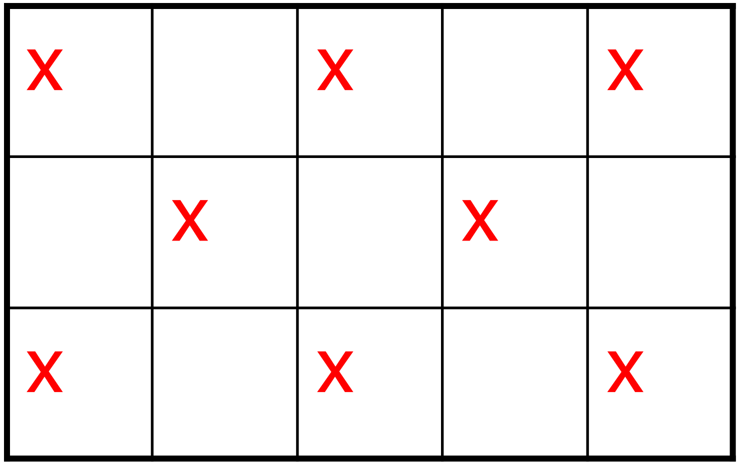
A[i][j] = max{A[i-1][j-1], A[i-1][j]+A[i][j-1]}

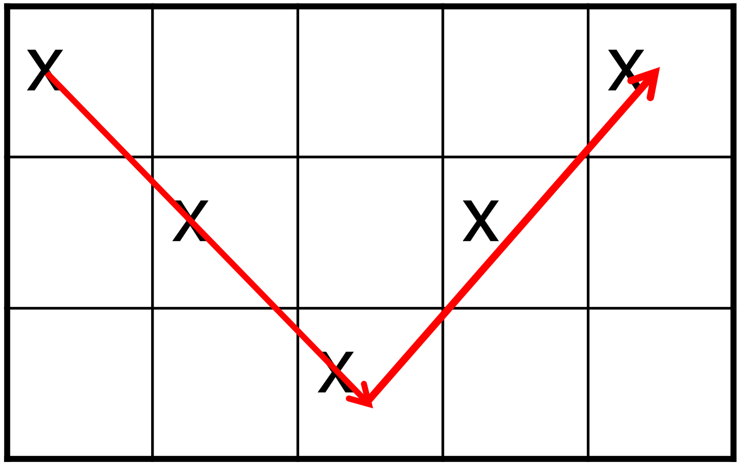
Exercise: Walks on 2d Arrays

write a code to print the values in the given order









STRINGS: ARRAY OF CHARACTERS
(ALSO DISCUSSED LATER IN CH3)

Character representation

- Each character has an integer representation

'a'	'b'	'c'	'd'	'e'	…	…	…	…	'z'

97 98 99 100 101 112

'A'	'B'	'C'	'D'	'E'	…	…	…	…	'Z'

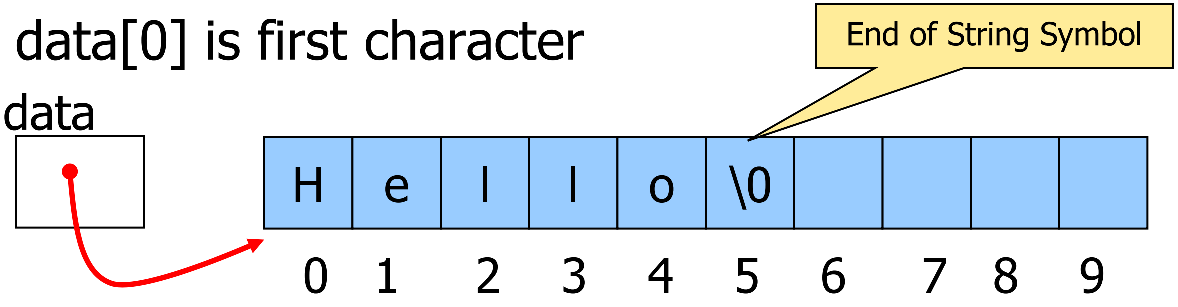
65 66 67 68 69 90

'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
48	49	50	51	52	53	54	55	56	57

'\0'	'\n'
0	10

Strings: Array of Characters

- A string is an *array* of characters
  - `char data[10] = “Hello”;`
  - `char data[10] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};`
- Can be accessed char by char
  - data[0] is first character



End of String Symbol

data

H	e	l	l	o	\0				
0	1	2	3	4	5	6	7	8	9

  - String ends with `‘\0’`
- Use printf/scanf to print/input strings
  - `printf(“%s”, data);        scanf(“%s”, data);`
  - `sprintf(data, “%d ”, X); sscanf(data, “%d ”, &X);`

Strings: Array of Characters vs. Character pointer

Give memory layout

- A string is an *array* of characters
  - `char data1[10] = "Hello";  // data1 is a constant !`
  - `char data1[10] = {'H', 'e', 'l', 'l', 'o', '\0'};`

  vs
  - `char data2[] = "Hello";`
  - `char data2[] = {'H', 'e', 'l', 'l', 'o', '\0'};`
- Character pointer
  - `char *data3 = "Hello";    // "Hello" is a constant !`
  - `char *data5;`
  - `data5 = "Hello";`
  - **string** data4 = "Hello"; /* if we use genlib.h
  - data4 = GetLine();   typedef **char ***string; */

Difference between char s1[5] = "ABCD" and char *s1="ABCD"

```
char s1[5]="ABCD";
…

printf(“%s\n”, s1);
printf(“%c\n”,s1[2]);
s1[2] = 'E';
printf(“%s\n”, s1);
s1="XYZ";      ✗
```

```
char *s1;
…
s1="ABCD";
printf(“%s\n”, s1);
printf(“%c\n”,s1[2]);
s1[2] = ‘E’; ✗
printf(“%s\n”, s1);
s1="xyz";
```

char s1[5]=“ABCD”; // is the same as
char s1[5]={‘A’, ‘B’, ‘C’, ‘D’, ‘\0;};

As we see later, we can fix this
s1= malloc(5); strcpy(s1,”ABCD”);

Exercise

- Write a function to count the number of characters in a string.
- Idea: count the number of characters before \0

H	e	l	l	o	\0				

Solution

```
int count_letters(char cdata[])
{
  int i=0;

  while (cdata[i] != '\0')
    i = i + 1;

  return(i);
}
```

Exercise

- Write a function that prints a string in reverse
- Idea: find the end of the string and print the characters backwards.

H	e	l	l	o	\0				

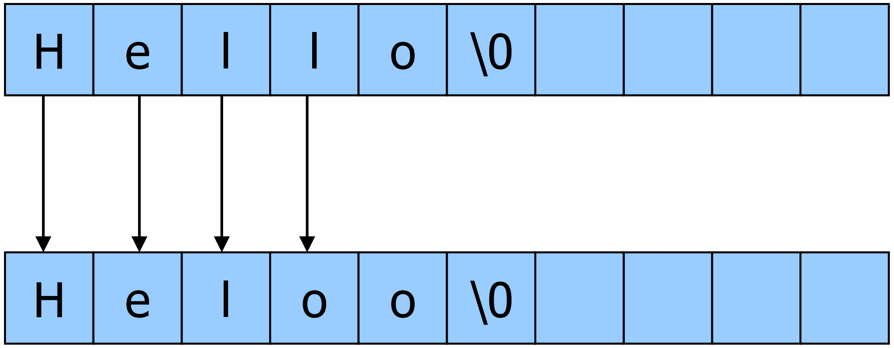
Output: olleH

Solution

```
void print_reverse(char pdata[])
{
  int size,position;
  size = count_letters(pdata);
  position = size - 1;
  while (position >= 0) {
      printf("%c",pdata[position]);
      position = position -1;
  }
  printf("\n");
  return;
}
```

Exercise

- Write a function that compares 2 strings S1 and S2 using lexicographic order.
- Idea: compare character by character
- Return
  - a neg value if S1 < S2,
  - 0 if S1 == S2
  - a pos value if S1 > S2



H	e	l	l	o	\0				

H	e	l	o	o	\0				

l < o in lexicographic order

Solution (incomplete)

```
int compare(char cdata1[], char cdata2[])
{
  int i= 0;

  while (cdata1[i] == cdata2[i]){
    i = i + 1;
  }
  return (cdata1[i] - cdata2[i]);
}
```

Solution (complete)

```
int compare(char cdata1[], char cdata2[])
{
  int i= 0;
  while (cdata1[i] && cdata2[i]
          && cdata1[i] == cdata2[i]){
     i = i + 1;
   }
  return (cdata1[i] - cdata2[i]);
}
```

Exercise: strings (char array)

- Write a function to check if string s1 appears in string s2? If so return 1, otherwise return 0;
- For example,

  If s1=“**abcd**” and s2=“xyzaabb**abcd**xyz”, then yes return 1.

BACK TO POINTERS:

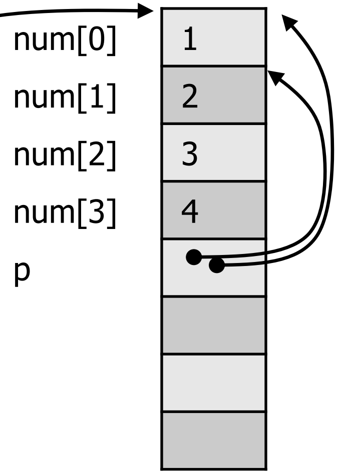
Pointers and Arrays, and Pointer Arithmetic

Pointers and Arrays

- The name of an array is the address of the first elements (i.e., a pointer to the first element in the array)
- The array name is a *constant* that always points to the first element of the array and its value **cannot** be changed.
- Array names and pointers may often be used interchangeably.

Example

```
int num[4] = {1,2,3,4}, *p, ~~q[];~~

p = num;          /* same as p = &num[0]; */

printf("%i", *p);             // print num[0]
printf("%i", *(p+2));         // print num[2]

p++;
printf("%i", *p);             // print num[1]
```



num[0]	1
num[1]	2
num[2]	3
num[3]	4
p	●

Recall: Pointer Arithmetic

- Four arithmetic operations are supported
  - +, -, ++, --
  - only integers may be used in these operations
  - Arithmetic is performed relative to the variable type being pointed to

    Example: p++;

    - if p is defined as int *p, p will be incremented by 4 (system dependent)
    - if p is defined as double *p, p will be incremented by 8 (system dependent
    - when applied to pointers, ++ means increment pointer to point to next element in memory
  - MOSTLY USED WITH ARRAYS (as we C now)

Pointer Arithmetic (con’t)

```
int  arr[10];
int *p;
p = &arr[0]; // or p = arr;
```

- Suppose `p` is a pointer to `arr[0]` and `k` is an integer
  - `p` and `arr` have the same value (`p` is a variable but `arr` is a ?)
  - `p+k` is defined to be `&arr[k]`
  - In other words, `p+k` computes the address of an array element located `k` elements **after** the address currently indicated by `p`
  - Similarly, `p-k` computes the address of an array element located k elements **before** the address currently indicated by `p`
- Suppose `p=&arr[5];`

  `p++` and `p--` would be the address of `arr[?]` and `arr[?]`

  `p+2` and `p-3` would be the address of `arr[?]` and `arr[?]`
- Suppose `p1=&arr[5]; p2=&arr[8];`

  `p1 - p2` expression has the `value ?`

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	*Postfix increment/decrement (see Note 2)*	
++ --	P*refix increment/decrement (see Note 2)*	**right-to-left**
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (change type)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	**right-to-left**
=	Assignment	**right-to-left**
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Note 1: Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer. **Note 2**: Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement y = x * z++; the current value of z is used to evaluate the expression (i.e., z++ evaluates to z) and z only incremented after all else is done.

Compiler dependent side effects: printf("%d %d\n", ++n, pow(2,n)); or A[i] = i++;

Avoid side effects! If you are not sure about side effects, you wont take advantage of idiomatic expressions of C.

One of the most common idiomatic expression: *p++

- * and ++ are **unary operators** competing for operant p (Recall: **Right-to-Left Rule**)
- So, ++ takes precedence over *
- *p++ is the same as *(p++)

lvalue: internal memory location capable of storing Data. The *l* at the beginning shows that *lvalues* can appear on the **l**eft side of assignment statement in C.

- Recall: Postfix ++ increments the value of p but returns the value that p had prior to the increment operation
  - Suppose a is 5; → X = 3 + a++; vs X = 3 + ++a;
- So, *p++ means "Dereference the pointer p as an **lvalue** the object to which it currently points. As a side effect, increment the value of p so that , if the original lvalue was an element in an array, the new value of p points to the next element in that array"
- How about *++p , ++*p
- How about *p+1, *(p+1)

Relationship between Pointers and Arrays

- index a pointer using array notation

```
char mystring[] = “This is a string”;
char *str;
int i;
str = mystring;
for(i =0; str[i]!=‘\0’; i++)
  printf(“%c”, str[i]);
```

Previous example with `pointer arithmetic`

```
char mystring[] = “This is a string”;
char *str;
for(str = mystring; *str!=‘\0’; str++)
  printf(“%c”, *str);
```

- Will the following do the same thing?

```
for(str = mystring; *str; str++)
  printf(“%c”, *str);
```

- How about the following?

```
str = mystring;
while(*str++) printf(“%c”, *str);
```

Relationship between Pointers and Arrays (cont'd)

```
int Sum(int a[], int n)
{
   int i, sum;
   sum = 0;
   for(i=0; i<n; i++){
        sum += a[i];
   }


   return(sum)
}
```

```
int Sum(int *ip, int n)
{
   int i, sum;
   sum = 0;
   for(i=0; i<n; i++){
        sum += *(ip+i);
        sum += *ip; ip++;
        sum += *ip++;
     }
   return(sum)
}
```

Then what is the difference?

- When the variables are originally declared…

`char a[5];`// memory is allocated for 5 elements

`char *p;`   // no memory is allocated yet

- We can use `p` after `p=a`;

  (What is the point? I can simply use `a`, right?)

- As we C later, pointers are needed for "dynamic memory allocation"
  - p = GetLine();   // vs. ~~scanf("%s", p);~~
  - scanf("%s", a);   // vs. ~~a=GetLine();~~

Exercise

- Using **pointer arithmetic**, re-write the previously discussed string functions
  - a function that counts the number of characters in a string.
  - a function that reverses the characters in a string.
  - a function that compares 2 strings S1 and S2 using lexicographic order.
  - function that searches string S1 in S2 and returns 1 if S1 is in S2; otherwise, it returns 0.

2D arrays and Pointers

Two Dimensional Arrays and Pointers

A two-dimensional array is stored in sequential memory locations, in row order.

```
int s[2][3] = {{2,4,6}, {1,5,3}};

int *sptr = &s[0][0]; // int *sptr = s; //is this OK?
```

```
// How about
int **ssptr = s;
// or
int *ssptr[] = s;
// or
int ssptr[][] = s;
…ssptr[i][j] …
```

```
Memory allocation:

s[0][0]   2
s[0][1]   4
s[0][2]   6
s[1][0]   1
s[1][1]   5
s[1][2]   3
```

```
A pointer reference to s[0][1] would be *(sptr+1)
A pointer reference to s[1][2] would be *(sptr+1*3+2)
```

*row offset * number of columns + column offset*

```
s[i][j] <-> *(sptr + i*3+j)
```

Array of pointers Dynamic 2d arrays

- int *s[5];

- int **s;

2D-char array vs. Array of strings (char pointers)

What is the key difference?

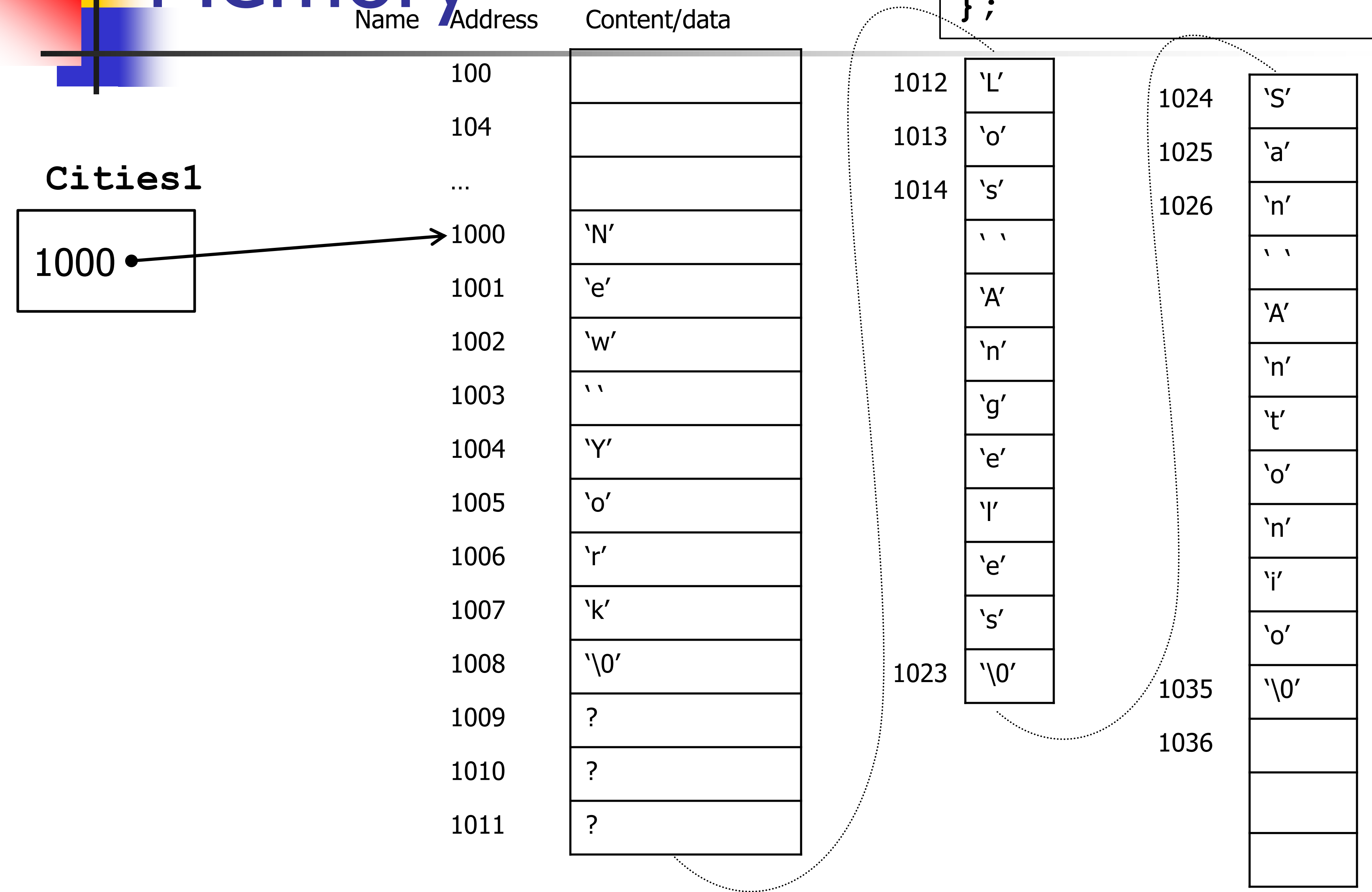
2D-char array

3

```
char Cities1[][12] = {
   “New York”,
   “Los Angeles”,
   “San Antonio”,
};
```

How does it look in Memory

```
char Cities1[][12] = {
   “New York”,
   “Los Angeles”,
   “San Antonio”,
};
```



Name

Cities1

1000

Address	Content/data
100	
104	
…	
1000	‘N’
1001	‘e’
1002	‘w’
1003	‘ ‘
1004	‘Y’
1005	‘o’
1006	‘r’
1007	‘k’
1008	‘\0’
1009	?
1010	?
1011	?

1012	‘L’
1013	‘o’
1014	‘s’
	‘ ‘
	‘A’
	‘n’
	‘g’
	‘e’
	‘l’
	‘e’
	‘s’
1023	‘\0’

1024	‘S’
1025	‘a’
1026	‘n’
	‘ ‘
	‘A’
	‘n’
	‘t’
	‘o’
	‘n’
	‘i’
	‘o’
1035	‘\0’
1036	

sizeof …

```
int n = sizeof Cities1;     // 36
int m = sizeof Cities1[0]; // 12
int nCities1 = n/m;
```

```
printf(“%d %s\n”,nCities1,Cities1[1]);
```

Can we print `Cities1[2][4]` // yes ‘A’

`Cities1[2][4] = ‘X’;` // yes we can

Array of strings (char pointers)

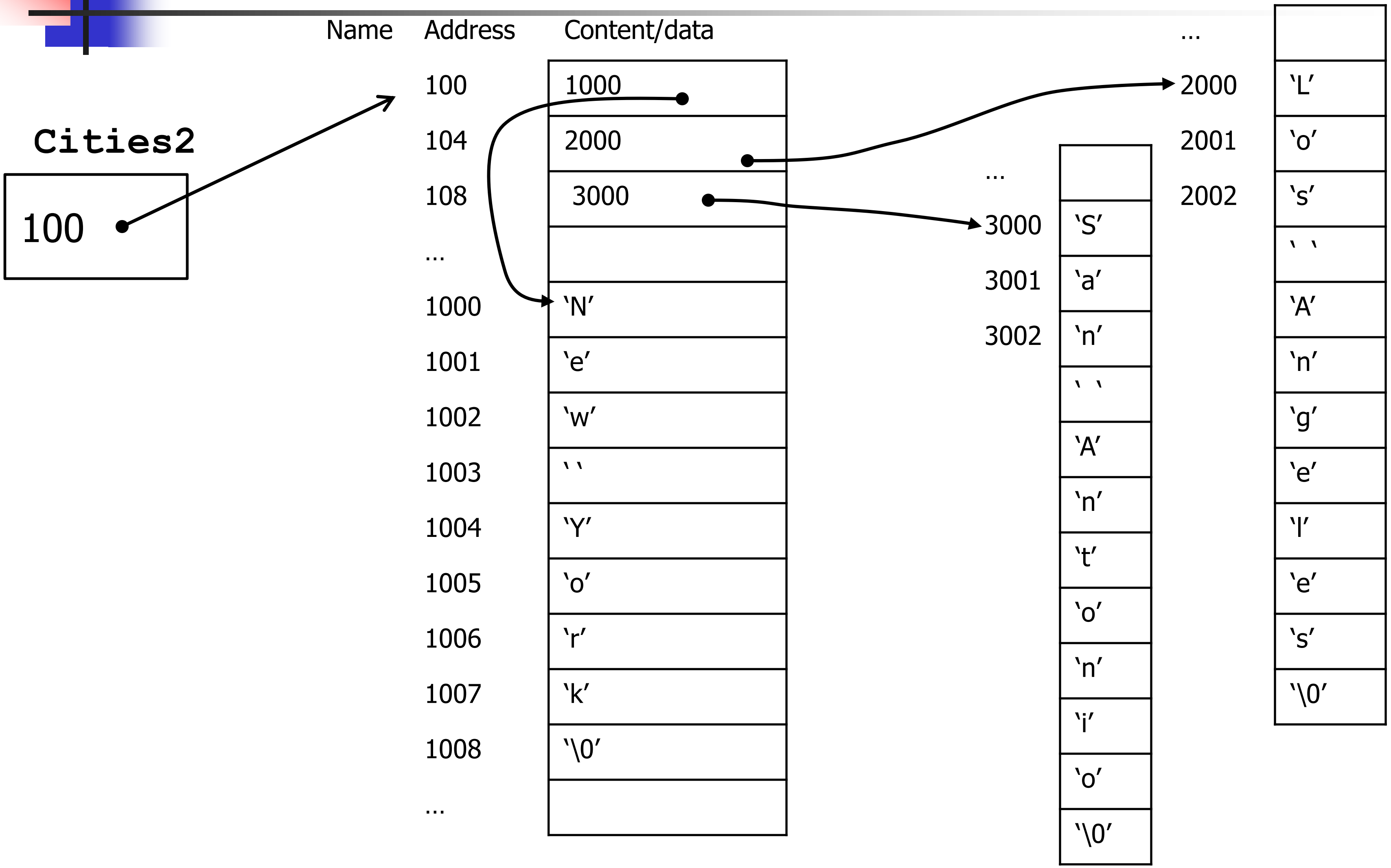
3

```
/* instead of
   char Cities1[][12] = {…} */

char *Cities2[] = {
  "New York",
  "Los Angeles",
  "San Antonio",
};
```

```
// textbook's version
typedef char *string;

string Cities2[] = {
  "New York",
  "Los Angeles",
  "San Antonio",
};
```

How does it look in Memory

```
char *Cities2[] = {
   "New York",
   "Los Angeles",
   "San Antonio",
};
```



Name	Address	Content/data
Cities2		100
	100	1000
	104	2000
	108	3000
	…	
	1000	‘N’
	1001	‘e’
	1002	‘w’
	1003	‘ ’
	1004	‘Y’
	1005	‘o’
	1006	‘r’
	1007	‘k’
	1008	‘\0’
	…	

Address	Content/data
…	
3000	‘S’
3001	‘a’
3002	‘n’
	‘ ’
	‘A’
	‘n’
	‘t’
	‘o’
	‘n’
	‘i’
	‘o’
	‘\0’

Address	Content/data
…	
2000	‘L’
2001	‘o’
2002	‘s’
	‘ ’
	‘A’
	‘n’
	‘g’
	‘e’
	‘l’
	‘e’
	‘s’
	‘\0’

sizeof …

```
int n = sizeof Cities2;      // 12
int m = sizeof Cities2[0];   // 4
int nCities2 = n/m;

printf(“%d %s\n”,nCities2,Cities2[1]);
```

Can we print `Cities2[2][4]` // yes ‘A’

`Cities2[2][4] = ‘X’;` // NO why?

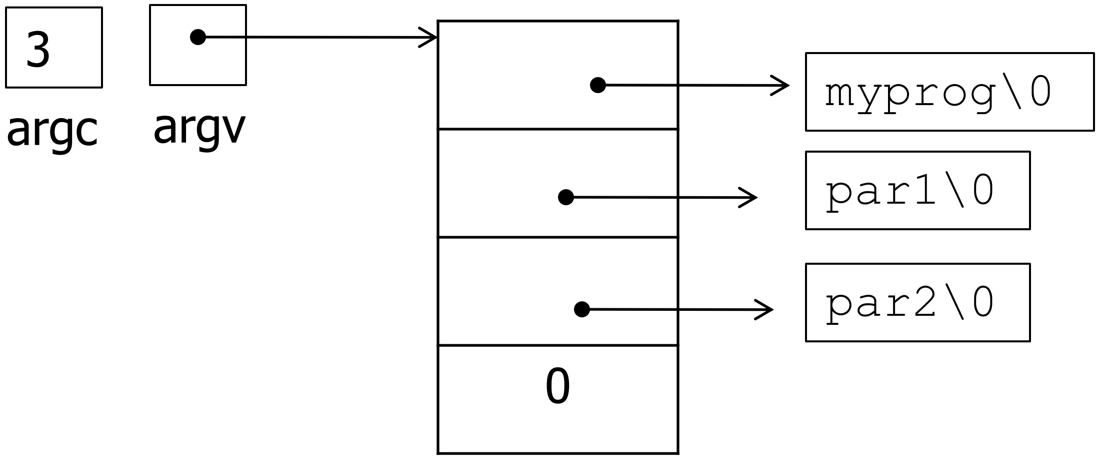
Command-line Arguments

- We can pass parameters to main program when it begins executing

```
> myprog par1 par2
```

- `main(int argc, char *argv[])`
  - `argc` is the number of parameters including program name (so it is at least 1)
    - in the above example argc is …
  - `argv` is the pointer to the array of pointers representing parameters as strings

Command-line Arguments (cont'd)



- Let's write a program that echoes command-line arguments (2 versions)
  - argv is an array of character pointers, so we can index the array
  - argv is also a pointer to an array of pointers, so we can manipulate the pointer

Command-line Arguments (cont’d)

```
#include <stdio.h>

main(int argc, char *argv[])
{
  int i;
  for(i=1; i < argc; i++){
    printf(“%s\n”, argv[i]);
  }
  return 0;
}
```

```
#include <stdio.h>

main(int argc, char *argv[])
{
  while(--argc > 0){
    printf(“%s\n”, *++argv);
  }
  return 0;
}
```

Can we print `argv[2][3]` // yes ‘2’

`argv[2][3] = ‘X’;` // yes we can. why?

Wildcards, Quotes, Back Quotes and Apostrophes

in shell commands (* ? [] " ` ')

- prog this is first second
- prog “this is first” second
- prog *
- prog *.c
- prog ??.c
- prog [a-d,AX]
- prog [abc]*[123]?.c

```
/* prog.c */
#include <stdio.h>
main(int argc, char *argv[])
{
   int i;
   for(i=1; i < argc; i++){
     printf(“%s\n”, argv[i]);
   }
   return 0;
}
```

` (Back Quotes) : Command Substitution
' (Apostrophe Marks) : No Change
$HOME

Exercise

- Write a program that takes two values and an operator between them as command-line arguments and prints the result.

  You will need `atoi(str)` which returns the numeric value for the string str. For example, `atoi("435")` returns 435

- For example,

```
 > mycal 23 + 45
 68
 > mycal 10 * 45
 450
```

Records (Structures)

- A collection of one or more variables, data members, grouped under a single name.
- Unlike arrays, the data members can be of different types.
- For example, consider a Payroll system
  - Name, title, ssn, salary, withholding …

Defining a new structure type (basic style)

- Define a new structure type
- Declare variables of new type

```
struct name {
   field-declarations
};

struct name var_name;
```

```
struct employeeRec {
        char *name;
        char title[20];
        char ssnum[11];
        double salary;
        int withholding;
 };

 struct employeeRec e1, e2;
```

How about memory layout?
&e1?

sizeof(struct employeeRec)? sizeof e1?

Defining a new structure type (textbook’s style)

- Typedef a new structure type
- Declare variables of new type

```
                a_name
typedef struct {
   field-declarations
} nameT;


nameT var_name;
```

sizeof e1?

```
                        employeeRec
typedef struct {
        char *name;
        char title[20];
        char ssnum[11];
        double salary;
        int withholding;
} employeeRecT;

employeeRecT e1, e2;
```

How about memory layout?

Record Selection
Accessing Data Members

- Suppose we have declared `e1` and `e2`
- We can refer the record as a whole by using its name: `e1 e2`
- To refer specific field, we write

`record_name` **`dot`** `field_name`

```
e1.name
e2.salary
```

Initializing Structures

```
struct employeeRec {
        char *name;
        char title[20];
        char ssnum[11];
        double salary;
        int withholding;
 };
```

we can initialize when we define a variable

```
struct employeeRec manager = {       // OK
   “name last”, “partner”, “xxx-xx-xxx”, 250.0, 1
};

employeeRecT manager = { /* book’s style  */
   “name last”, “partner”, “xxx-xx-xxx”, 250.0, 1
};
```

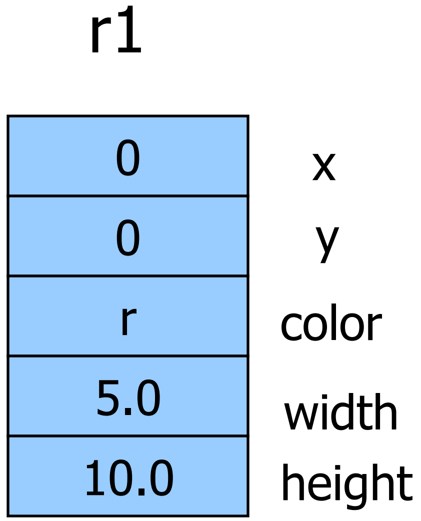
Or we can initialize in the program

```
e1.name = “Turgay Korkmaz”;     // OK
e1.title = “Associate Prof”;    // !!! NOT OK why? !!!
e1.ssnum = “xxx-xx-xxx”;        // !!! NOT OK why? !!!
e1.salary = 999999.99;
e1.withholding = 2;
```

```
strcpy(e1.title, “Associate Prof”);
strcpy(e1.ssnum,”xxx-xx-xxx”);
```

Initializing Structures another example

```
struct Rect
{
   double x;
   double y;
   char color;
   double width;
   double height;
};

struct Rect r1 = {0,0,'r',5,10};
```



r1

0	x
0	y
r	color
5.0	width
10.0	height

Can you do the same using **typedef**?

Assignment operator

- Assignment operator is defined for structure of the *same type*.

```
struct Rect
{
  double x;
  double y;
  char color;
  double width;
  double height;
};
struct Rect r1, r2;
```

```
 r1.x = 10.5;
 r1.y = 18.99;
 r1.width = 24.2;
 r1.height = 15.9;
 r1.color = 'r';

/* Copy all data
 * from r1 to r2.
 */
 r2 = r1;
```

Exercise: how about `e2=e1;` from previous slides!

Scope of a Structure

- Member variables are local to the structure.
- Member names are not known outside the structure.

```
struct Rect
{
  double x;
  double y;
  char color;
  double width;
  double height;
};
struct Rect r1, r2;
int x;
r1.x = 3.5
x = 4;
y = 2;   // compiler will give an error
```

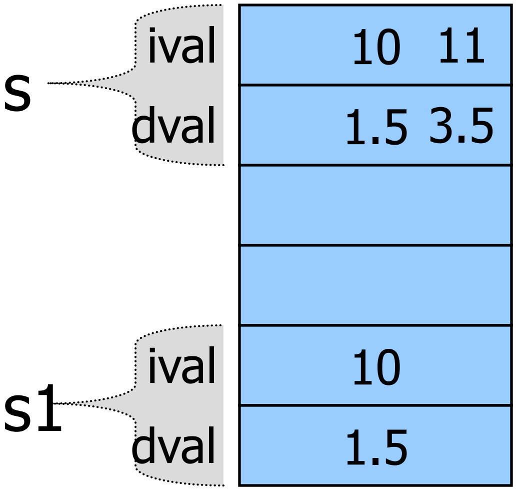
Structures as Arguments to Functions

- A structure is passed as an argument to a function (**call-by-value**).
  - Changes made to the formal parameters do not change the argument. (see next exercise)
- A pointer to a structure may also be passed as an argument to a function (**call-by-reference**).
  - Changes made to the formal parameters also change the argument.
- Functions return type could be `struct` too… or pointer to `struct`

How can you pass a whole array to a function using call-by-value?

Call by Value Example

```
struct simple
{
        int ival;
        double dval;
};
void fun1(struct simple s)
{
        s.ival = s.ival + 1;
        s.dval = s.dval + 2;
}

int main(void)
{
        struct simple s1 = {10, 1.5};
        fun1(s1);
        printf(“%i %lf\n”,
                s1.ival , s1.dval );
        return 0;
}
```



We will see 10 1.500000

Exercise

- Write a program using structures that manipulates pairs.
- Write functions for Addition and multiplication of pairs that are defined as

$$(a,b)+(c,d)=(a+c,b+d)$$

$$(a,b)*(c,d)=(a*c,b*d)$$

Exercise (cont'd): Pair Structure

- Store two integers to represent the first and second number of pair

```
struct pair
{
   int first;
   int second;
};
```

```
typedef struct pair
{
   int first;
   int second;
} pairT;
```

Exercise (cont’d): Addition

```
struct pair add(struct pair p1,
                struct pair p2)
{
  struct pair temp;
  temp.first = p1.first + p2.first;
  temp.second = p1.second + p2.second;
  return temp;
}
```

Exercise (cont'd): Multiplication

```
struct pair multiply(struct pair p1,
                     struct pair p2)
{
  struct pair temp;
  temp.first = p1.first * p2.first;
  temp.second = p1.second * p2.second;
  return temp;
}
```

Exercise (cont'd): How to use the functions

```
struct pair mp1,mp2,mp3,mp4;

printf("Enter first pair\n");
scanf("%d %d",&mp1.first, &mp1.second);

printf("Enter second pair\n");
scanf("%d %d",&mp2.first, &mp2.second);

mp3 = add(mp1, mp2);
printf("Addition result =
   (%d,%d)\n",mp3.first,mp3.second);

mp4 = multiply(mp1,mp2);
printf("Multiplication result =
   (%d,%d)\n",mp4.first,mp4.second);
```

Exercise (cont'd): add two new functions

- Update the program to support the following on pairs

```
c*(a,b) = (c*a,c*b)
(a,b)^c = (a^c,b^c)
```

Pointers to Records

Why do we need pointers to records…

- Passing a pointer to a record, functions can manipulate the fields of that record
- Dynamic memory allocations can be done
- A pointer to a record is usually smaller and more easily manipulated than record itself

Pointers to Records: declaration

```
struct employeeRec {
      char *name;
      char title[20];
      char ssnum[11];
      double salary;
      int withholding;
 };

 struct employeeRec e1, e2;

 struct employeeRec *ePtr;

 ePtr = &e1;
```

```
typedef struct {
   // same fields
} employeeRecT;
employeeRecT e1, e2;

employeeRecT *ePtr;
ePtr = &e1;
```

```
typedef struct {
    // same fields
} *employeeRecTptr;
//employeeRecT e1, e2;
employeeRecTptr ePtr;
// ePtr = &e1;
ePtr=New(employeeRecTptr);
```

What happens in memory?

Pointers to Records: Record selection

- How about `*ePtr.salary`
- Which means `*(ePtr.salary)` Why?
- We want `(*ePtr).salary` Right?
- But this notation is much too cumbersome, so C defines **->** operator
- `ePtr->salary` has the same effect as `(*ePtr).salary`

Operator	Description		Associativity
()	Parentheses (function call) (see Note 1)		left-to-right
[]	Brackets (array subscript)		
.	Member selection via object name		
->	Member selection via pointer		
++ --	Postfix increment/decrement (see Note 2)		
++ --	Prefix increment/decrement (see Note 2)	←←←←←←←	**right-to-left**
+ -	Unary plus/minus		
! ~	Logical negation/bitwise complement		
(type)	Cast (change type)		
*****	Dereference		
&	Address		
sizeof	Determine size in bytes		
*** / %**	Multiplication/division/modulus		left-to-right
+ -	Addition/subtraction		left-to-right
<< >>	Bitwise shift left, Bitwise shift right		left-to-right
< <=	Relational less than/less than or equal to		left-to-right
> >=	Relational greater than/greater than or equal to		
== !=	Relational is equal to/is not equal to		left-to-right
&	Bitwise AND		left-to-right
^	Bitwise exclusive OR		left-to-right
|	Bitwise inclusive OR		left-to-right
&&	Logical AND		left-to-right
||	Logical OR		left-to-right
?:	Ternary conditional	←←←←←←←	**right-to-left**
=	Assignment	←←←←←←←	**right-to-left**
+= -=	Addition/subtraction assignment		
***= /=**	Multiplication/division assignment		
%= &=	Modulus/bitwise AND assignment		
^= |=	Bitwise exclusive/inclusive OR assignment		
<<= >>=	Bitwise shift left/right assignment		
,	Comma (separate expressions)		left-to-right

Note 1: Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer. **Note 2**: Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement y = x * z++; the current value of z is used to evaluate the expression (i.e., z++ evaluates to z) and z only incremented after all else is done.

Compiler dependent side effects: printf("%d %d\n", ++n, pow(2,n)); or A[i] = i++;

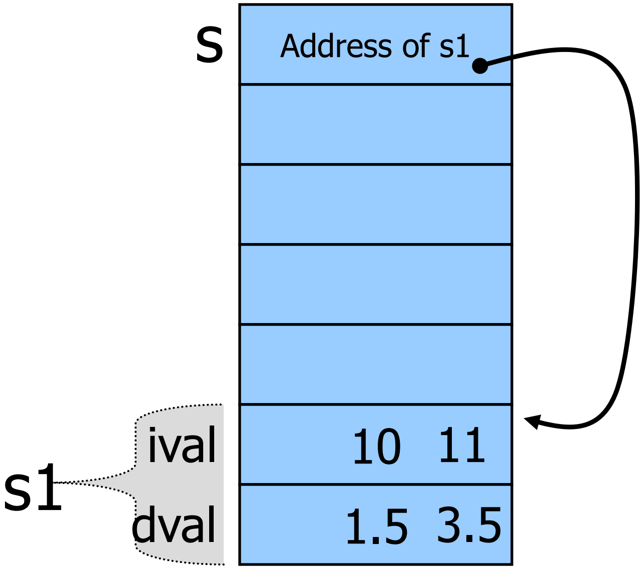
Avoid side effects! If you are not sure about side effects, you wont take advantage of idiomatic expressions of C.

Structures as Arguments to Functions

- A structure is passed as an argument to a function (**call-by-value**).
  - Changes made to the formal parameters do not change the argument. (see next exercise)
- A pointer to a structure may also be passed as an argument to a function (**call-by-reference**).
  - Changes made to the formal parameters also change the argument.
- Functions return type could be `struct` too… or pointer to `struct`

Call by Reference Example

```
struct simple
{
        int ival;
        double dval;
};
void fun1(struct simple *s)
{
        s->ival = s->ival + 1;
        s->dval = s->dval + 2;
}

int main(void)
{
        struct simple s1 = {10, 1.5};
        fun1(&s1);
        printf("%i %lf\n",
               s1.ival , s1.dval );
        return 0;
}
```



We will see 11 3.500000

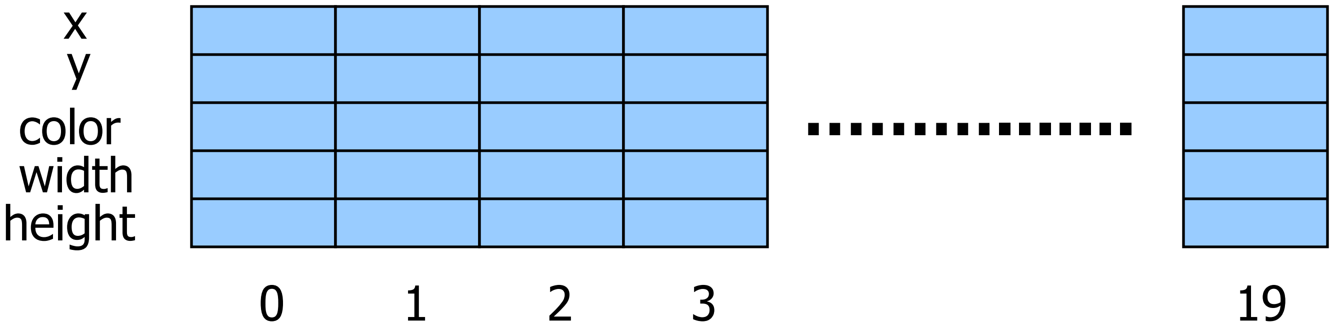
Arrays of Structures

- Arrays of structures may be declared in the same way as other C data types.

```
struct rect rectangles[20];
```

- `rectangles[0]` references first structure of rectangles array.

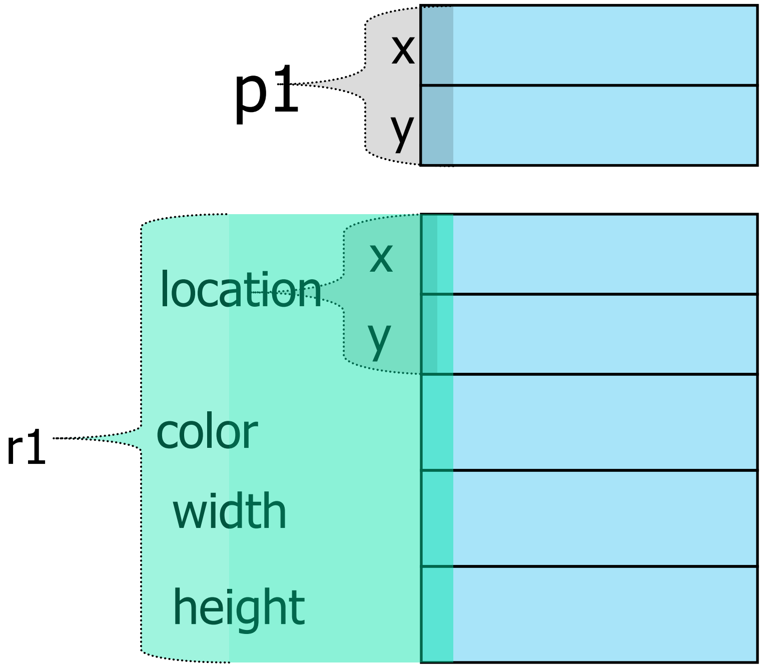
```
rectangles[0].color = ‘r’;
```



Nested Structures

Structure definitions may contain data members that are other structures:

```
struct Point{
   double x;
   double y;
};
struct Rect2
{
   struct Point location;
   char color;
   double width;
   double height;
};
Struct Point p1;
struct Rect2 r1;
r1.location.x= 2.3
r1.location.y=4.5;
r1.color = 'r';
```



Nested Structures with pointers

```
struct Point{
   double x;
   double y;
};
```

```
struct poly5 {
   struct Point *p[5];
   char color;
   struct Point center;
};
```

```
struct Point t, *tp;

struct poly5 p, *pp;
```

Explain the problems (if any) in the following statements and try to fix them…

```
p.p[4].x=5;

pp = &p;

p->p->x=3;

p->center.y=5;
```

Self-referential Structures

```
struct Point{
  double x;
  double y;
 struct Point *next;
};
```

- Why do we need something like that? *(discussed later when we talk about data structures such as linked lists, trees, graphs etc.)*

Unions

- A union is a variable that may hold (at different times) values of different types and sizes in a **single area of storage**…

```
union u_two {
  int ival;
  double dvalue;
} u, *uPtr;
```

- Variable u will be large enough to hold largest of two types `u.ival` or `u.dval`
- In case of pointer declaration, `uPtr->ival`

Bit-fields

```
struct flags{
  unsigned int sync: 1;
  unsigned int fin:  1;
  unsigned int type: 2;
} f;

f.sync=1;
if (f.fin == 0 && …)
```

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement (see Note 2)	**right-to-left**
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (change type)	
*****	Dereference	
&	Address	
sizeof	Determine size in bytes	
*** / %**	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	**right-to-left**
=	Assignment	**right-to-left**
+= -=	Addition/subtraction assignment	
***= /=**	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Note 1: Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer. **Note 2**: Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement y = x * z++; the current value of z is used to evaluate the expression (i.e., z++ evaluates to z) and z only incremented after all else is done.

Compiler dependent side effects: printf("%d %d\n", ++n, pow(2,n)); or A[i] = i++;

Avoid side effects! If you are not sure about side effects, you wont take advantage of idiomatic expressions of C.

Dynamic Memory Allocation

Dynamic Memory Allocation

- At runtime, memory can be dynamically allocated from unused storage called **heap**
- A program may create as many or as few variables as required, offering greater **flexibility**
- Dynamic allocation is often used to **support data structures** such as stacks, queues, linked lists and trees.

Dynamic Memory Allocation

`#include <stdlib.h>`

- **void *malloc(size_t *size*);**
  - allocates `size` bytes of memory

```
int *ip;
char *cp;
void *vp;
ip = malloc(?);
ip = malloc(sizeof(int));
cp = malloc(10);
cp = (char *) malloc(10);
```

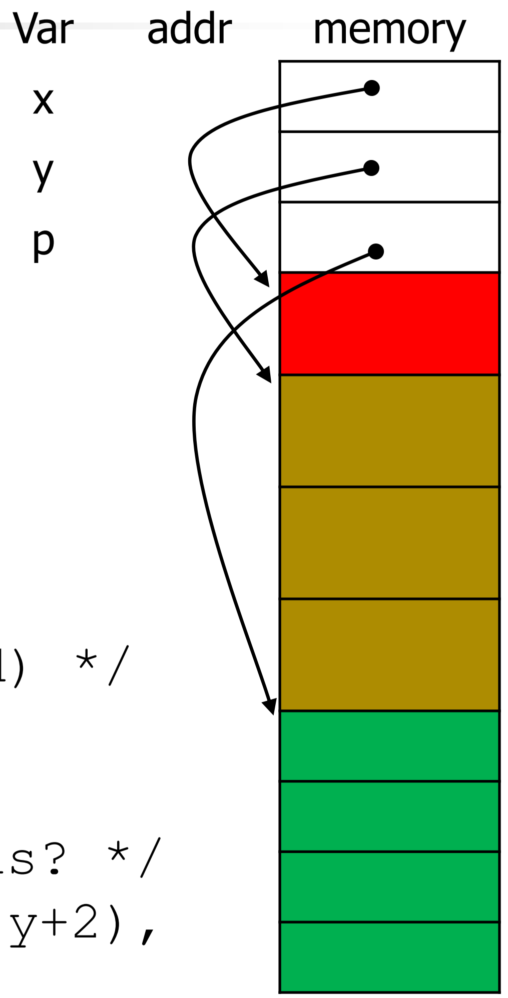
- **void *calloc(size_t *nitems*, size_t *size*);**
  - allocates `nitems*size` bytes of cleared memory
- **void free(void **ptr*);**
- **void *realloc(void **ptr*, size_t *size*);**
  - The size of memory requested by malloc or calloc can be changed using realloc

malloc and calloc

- Both functions return a **pointer** to the newly allocated memory
- If memory can not be allocated, the value returned will be a **NULL** value
- The pointers returned by these functions are declared to be a **void** pointer, WHY?
- A cast operator should be used with the returned pointer value to **coerce** it to the proper pointer type (otherwise, compiler gives warning)

Example of malloc and calloc

```
double *x, *y;
int *p;

/* Allocate memory for 1 double.  */
x = (double *) malloc( sizeof(double) );

/* Allocate memory for 3 double.  */
y = (double *) malloc(3*sizeof(double));

/* Allocate memory for 4 integers.(cleared) */
p = (int *) calloc(4,sizeof(int));

/* how can we access these memory locations? */
*x, x[0], y[0], y[1], y[2], *y, *(y+1), *(y+2),
*p, *(p+1), … p[0], p[1], …
```



Memory limitation

- Dynamic memory is finite. So,
  - We need to check if we really get the memory we want!

```
char *a;
a = (char *) malloc(10);
if (a==NULL){
      printf("No memory available"); exit(-1);
}
```

  - We need to free memory if we don't need it anymore

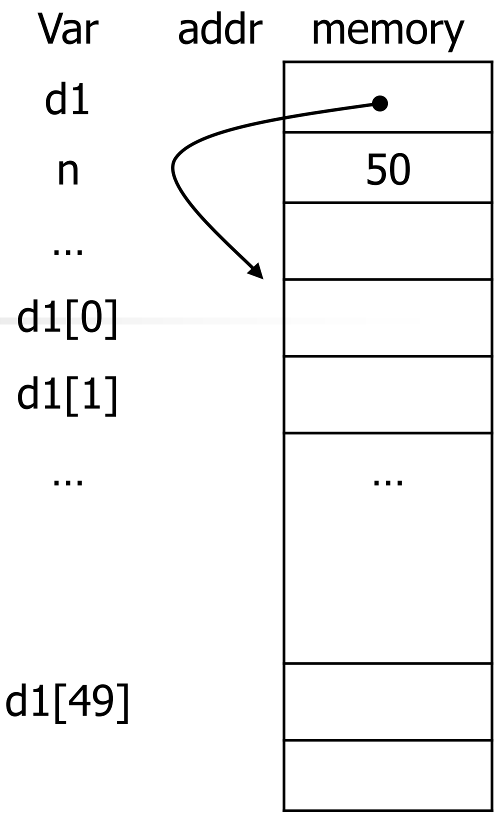
```
free(a);
```

No garbage collection in C

- `GetBlock` function provided by the textbook's genlib library checks out-of-memory condition

```
a = (char *) GetBlock(10); /* … */  FreeBlock(a);
```

Dynamic arrays (1D)



Var	addr	memory
d1		●
n		50
…		
d1[0]		
d1[1]		
…		…
d1[49]		

- As we saw before,
  we use malloc or calloc (how?)

```
double *d1;
int n = 50;

d1 = (double *)  malloc(n*sizeof(double));
if (d1==NULL){ printf("No memory available"); exit(-1);}
```

- Textbook's genlib library provides

```
d1 = NewArray(n, double);
/* NewArray is a macro expanded to */
d1 = (double *) GetBlock( n * sizeof (double));
```

Dynamic arrays (2D) - I

we can use a dynamic 1D array of data items

```
double dd[10][50];
double *d2;
int r = 10, c = 50;
d2 = (double *)
     malloc(r*c*sizeof(double));
if (d2==NULL) Error("No memory available");
```

How will you access data item [i][j]?

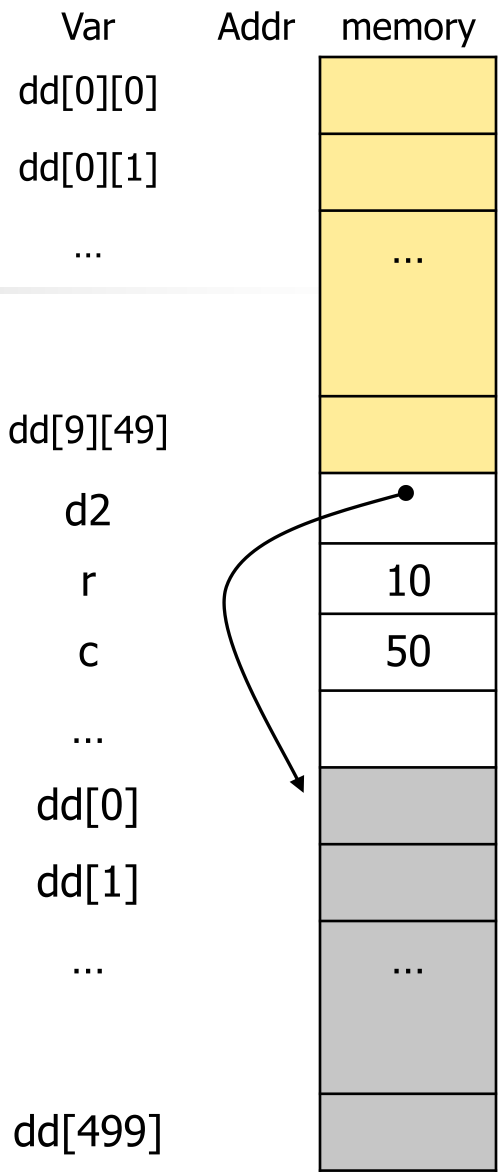
`dd[i][j]` or `*(dd + i*50+j)`

Will ~~`d2[i][j]`~~ work? why/why not?

How about

`d2[i*c+j]` or `*(d2 + i*c+j)`

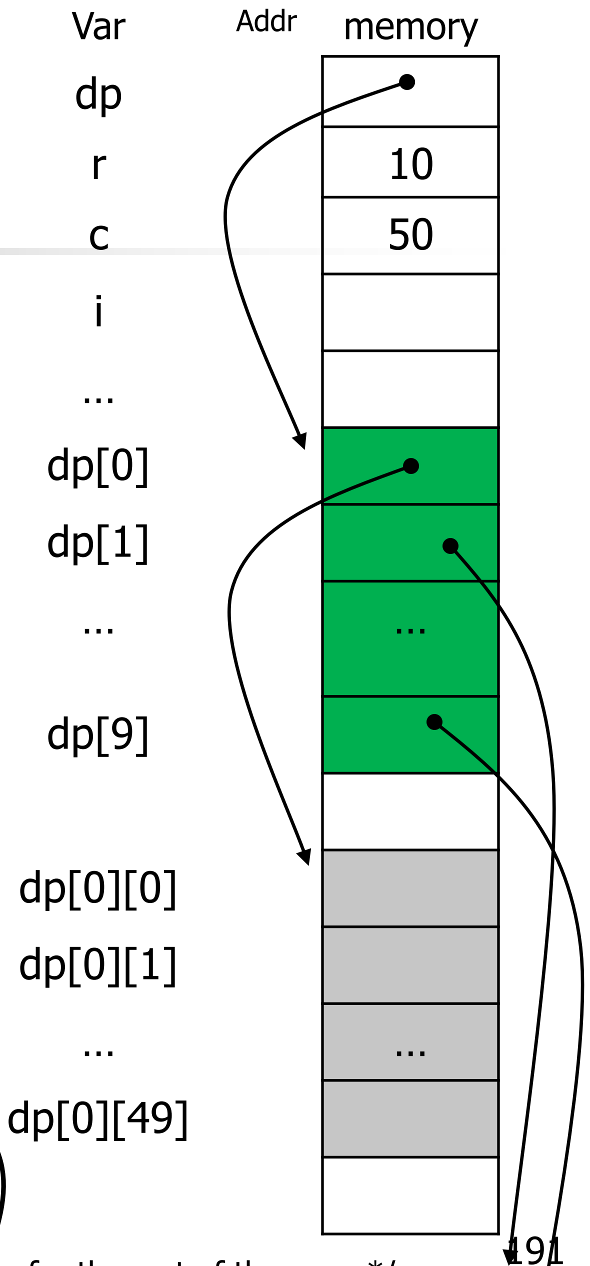
/* how will you free all the memory allocated */



Dynamic arrays (2D) – II

we can use 1D array of pointers and 1D array of data items

```
double **dp;
int r = 10, c = 50, i;
dp = (double **)
        malloc(r*sizeof(double *));
if (dp==NULL) Error("No memory available");
for(i=0; i<r; i++){
 dp[i]=(double *)
             malloc(c*sizeof(double));
 if (dp[i]==NULL) Error("No memory available");
}
```



How will you access data item [i][j]?

`dp[i][j]` or ~~`*(dp + i*50+j)`~~

/* how will you free all the memory allocated */

/* also we should free partially allocated memory in for loop if there is not enough memory for the rest of the rows */

DYNAMIC RECORDS

Dynamic Records (basic style)

```
struct employeeRec {
      char *name;
      char title[20];
      char ssnum[11];
      double salary;
      int withholding;
};
struct employeeRec e1, e2;
struct employeeRec *ePtr;

ePtr = (struct employeeRec *) malloc( sizeof(struct employeeRec) );
if (ePtr==NULL) Error(“No memory available”);

      /* OR use Textbook’s genlib library  */
ePtr = GetBlock( sizeof(struct employeeRec) );
```

Dynamic Records (typedef style)

```
typedef struct {
        char *name;
        char title[20];
        char ssnum[11];
        double salary;
        int withholding;
} employeeRecT;
employeeRecT e1, e2;
employeeRecT *ePtr;

ePtr = (employeeRecT *) malloc( sizeof(employeeRecT) );
if (ePtr==NULL) Error(“No memory available”);

          /* OR use Textbook’s genlib library  */
ePtr = GetBlock( sizeof(employeeRecT) );
```

Pointers to Records: declaration

```
struct employeeRec {
      char *name;
      char title[20];
      char ssnum[11];
      double salary;
      int withholding;
 };

 struct employeeRec e1, e2;

 struct employeeRec *ePtr;

 ePtr = &e1;
```

```
typedef struct {
   // same fields
} employeeRecT;
employeeRecT e1, e2;

employeeRecT *ePtr;
ePtr = &e1;
```

```
typedef struct {
    // same fields
} *employeeRecTptr;
//employeeRecT e1, e2;
employeeRecTptr ePtr;
// ePtr = &e1;
ePtr=New(employeeRecTptr);
```

Dynamic Records (3)
Use New() in Textbook’s genlib

```
 typedef struct {
        char *name;
        char title[20];
        char ssnum[11];
        double salary;
        int withholding;
 } *employeeT;
employeeT ePtr;

ePtr = New(employeeT);
/* New is a macro expanded to */
ePtr =(employeeT)GetBlock( sizeof *((employeeT) NULL) );

/* so, if you want, you can use malloc too as follows */
ePtr = (employeeT) malloc( sizeof *((employeeT) NULL) );
if (ePtr==NULL) Error(“No memory available”);
```

```
typedef struct {
   // same fields
} employeeRecT;
employeeRecT e1, e2;

employeeRecT *ePtr;
ePtr = &e1
ePtr = New(employeeRecT *);
```

Summary

stdlib.h vs. genlib.h

```
char *a;
double *d;
int   *i;
int n = 50;
```

- #inclide <stdlib.h>

```
a = (char *) malloc(10);
if (a==NULL){/* Err msg, Quit */}
---------------------------
d=(double *)
   malloc(n*sizeof(double));
---------------------------
i=(int *) malloc(sizeof(int));
---------------------------
free(a); free(d);
---------------------------
```

- #include “genlib.h”

```
a = (char *)GetBlock(10);

-----------------------------
d=NewArray(n,double); //macro

-----------------------------
i = New(int *); //macro
-----------------------------
FreeBlock(a); FreeBlock(d);
-----------------------------
```

```
sizeof variable
sizeof(type)
```

DYNAMIC ARRAY OF RECORDS AND RECORD POINTERS

Recall dynamic records from previous slides

Dynamic Array of Records (1D)

```
struct employeeRec {
       char *name;
       char title[20];
       char ssnum[11];
       double salary;
       int withholding;
};
struct employeeRec e1, e2;
struct employeeRec *ePtr, *eArray;
int n=50;
ePtr =  (struct employeeRec *)
              malloc( sizeof(struct employeeRec) );
if (ePtr==NULL) Error(“No memory available”);
eArray = (struct employeeRec *)
              malloc( n * sizeof(struct employeeRec) );
if (eArray==NULL) Error(“No memory available”);
// How can we access array elements? *(eArray + 5) vs. eArray[5]
```

Dynamic Array of Record Pointers (1D) to create 2D array of records

How can we access the records and the fields in them?

eArray[1] **.** or **->** salary?
eArrayPtr[2][3] **.** or **->** salary?

```
struct employeeRec {
         char *name;
         char title[20];
         char ssnum[11];
         double salary;
         int withholding;
};
struct employeeRec e1, e2;
struct employeeRec *ePtr, *eArray, **eArrayPtr;
int n=50, col=40;
ePtr = (struct employeeRec *) malloc(sizeof(struct employeeRec));
if (ePtr==NULL) Error("No memory available");
eArray=(struct employeeRec *) malloc(n*sizeof(struct employeeRec));
if (eArray==NULL) Error("No memory available");

eArrayPtr =(struct employeeRec **)
                   malloc(n*sizeof(struct employeeRec *));
if (eArrayPtr==NULL) Error("No memory available");
for(i=0; i< n; i++){
  eArrayPtr[i] = (struct employeeRec *) malloc(sizeof(struct employeeRec) );
  if (eArrayPtr[i]==NULL) Error("No memory available");
}
```

col *

Exercise

Suppose you are given `int row, col;`

- Create a dynamic 2D array of records?
- Create a dynamic 2D array of record pointers and records?

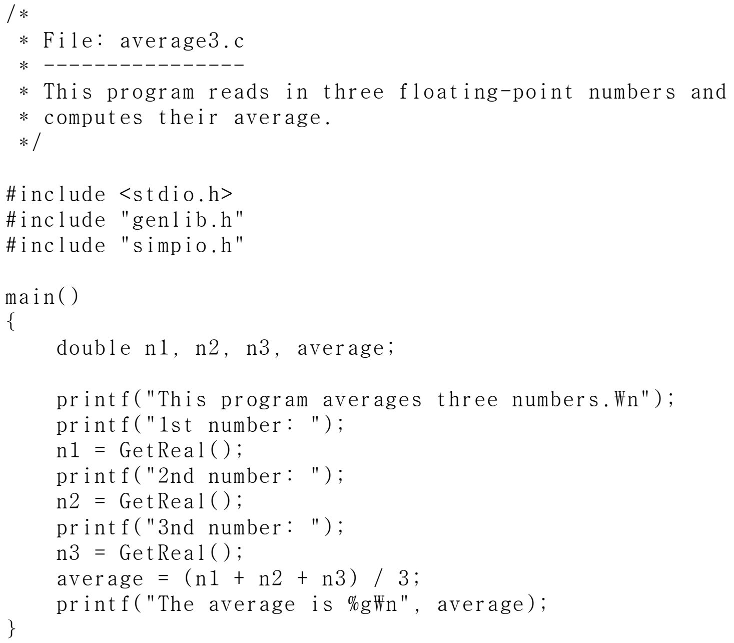
Suppose you are given `int x, y, z;`

- Create a dynamic 3D array of integers

To compile the programs using Book’s Libraries on our Linux machines: use **gccx** instead of **gcc**

- Login to linux, type/save your program

```
main212:> pico average3.c
```



```
/*
 * File: average3.c
 * ----------------
 * This program reads in three floating-point numbers and
 * computes their average.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double n1, n2, n3, average;

    printf("This program averages three numbers.₩n");
    printf("1st number: ");
    n1 = GetReal();
    printf("2nd number: ");
    n2 = GetReal();
    printf("3nd number: ");
    n3 = GetReal();
    average = (n1 + n2 + n3) / 3;
    printf("The average is %g₩n", average);
}
```

```
main212:> gccx average3.c -o average3
```