PDS LAB – 7 (Section-5) Date: 20th March 2017;
Pointers & Dynamic Memory Allocation
Tutorial Problems

1. Write a C function *int* factor_compute (int*)* to return all factors of a given positive integer in increasing order. Use pointers for passing the arguments and return the result. Demonstrate the utility of the above C function using main () function. Input the positive number through key board and print the number and its factors as output. Use malloc () and free () functions to allocate and release the memory.

2. Using pointers write C functions to (i) *char* enter-text (char*)* enter input text terminated by enter key and return the entered text through a pointer and (ii) *int string_lenght (*char)* to determine the length of the string and return it to the main function. Through main () function, call the above 2 functions and print the input string and its length.

3. Write a C program (using pointers) to count the number of positive, negative and zero values in an $M \times N$ matrix. Use appropriate dynamic memory allocation functions to allocate the memory. Enter the values of M (rows), N (columns) and elements of the matrix through key-board, and print the elements of a matrix row-column form and mention the number of positive, negative and zero values present in the matrix.

Assignment Problems (For All Students)

1. Write a C function (using pointers) to compute and return the common factors for the given sequence of numbers. Use the solution of Tut-1 question to solve this problem.

2. Write a C function (using pointers) to reverse the input string and then check whether the given string is palindrome or not? Use the solution of Tut-2 question for solving this question.

3. Write C functions to compute (i) addition of 2 matrices and (ii) product of 2 matrices using pointers. Through main (), input the 2 matrices using keyboard and print their sum and product.

4. Write a C function (using pointers) to merge the 2 sorted arrays into a single sorted array and return the sorted array to the main function. Input 2 sorted arrays from keyboard through main function and print the returned merged array.

5. Solve the above problem using recursive calls and pointer manipulation.

Assignment Problems (For those who completed 5 assignment problems)

1. Write C functions to (i) Enter the list of names (use the solution of Tut-2 question) and (ii) sort them based on alphabetical order. Through main () specify the number of names in the list, and call the above functions to enter the list of names and for sorting. Print both the input list of names and sorted list of names.

2. Write a C function to compute the determinant of a given matrix. Use pointers and appropriate memory allocation functions to solve the above problem.

Programming and Data Structures

Debasis Samanta

Computer Science & Engineering

Indian Institute of Technology Kharagpur

Spring-2017

Lecture #7

Pointers in C

Concept of Pointer

Concept of Pointer

Who lives in their?

London SW1A 1AA, UK

What is the address?

Today's Discussion…

* Concept of Pointer
* Pointers and its Applications
* Pointer Manipulation in C
* Pointers and Array
* Pointer Arrays
* Command Line Arguments
* Pointers to Functions

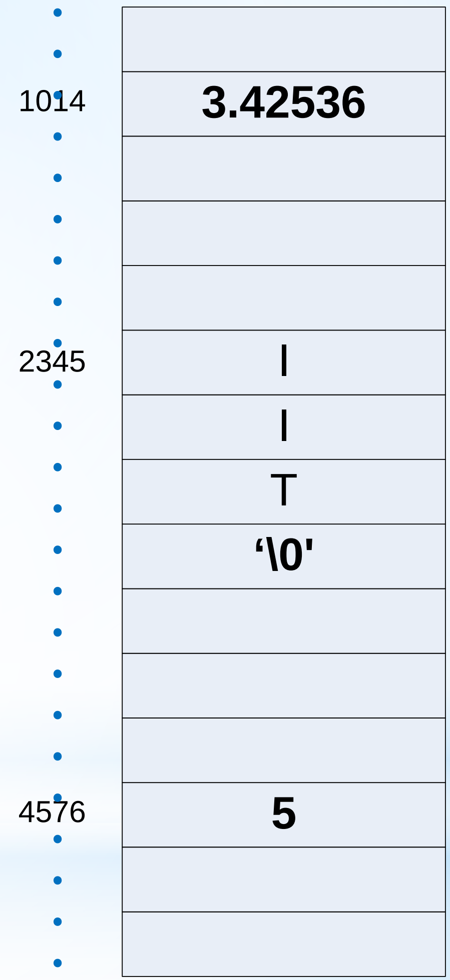
Concept of Pointer

- In memory, every stored data item occupies one or more contiguous memory cells.
  - The number of memory cells required to store a data item depends on its type (char, int, float, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Concept of Pointer

```
int y = 5;
char name[5]="IIT";
float x;
```

```
Memory locations
&y = 4576
&x = 1014
name[5] = 2345
```



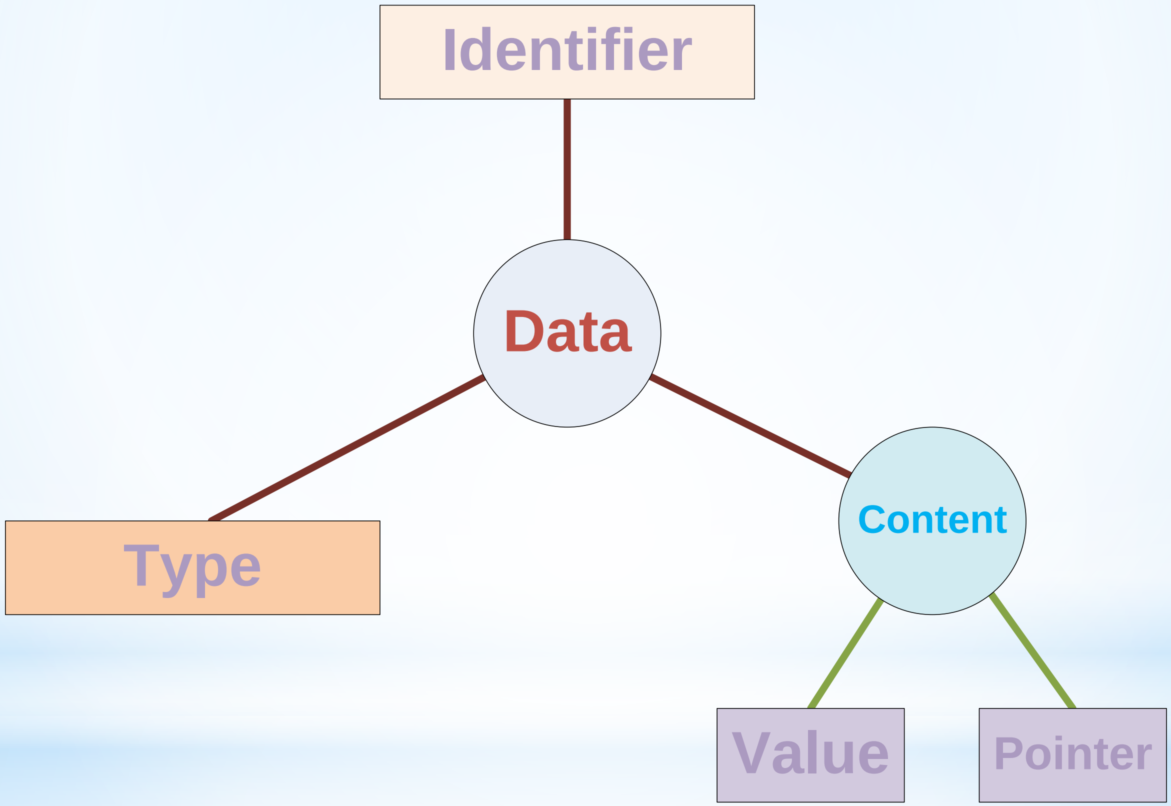
Concept of Pointer

- A pointer is a variable that represents the location (rather than the value) of a data item.

```
int x = 5;
```

Identifier	:	x
Value	:	5
Memory location	:	4576

Concept of Pointer



Concept of Pointer

- In memory, every stored data item occupies one or more contiguous memory cells.
    - The number of memory cells required to store a data item depends on its type (char, int, float, double, etc.).

- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
    - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Concept of Pointer

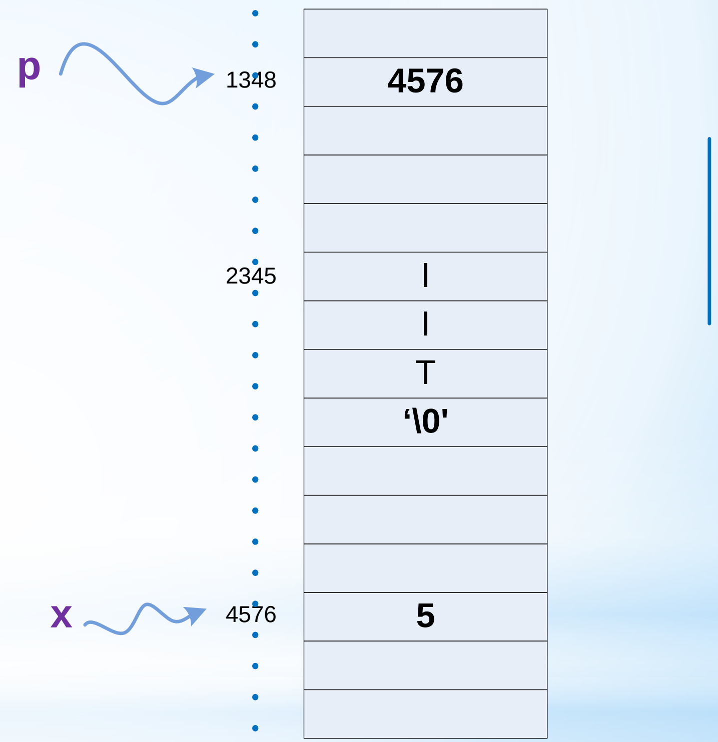
- In many programs, it is required to know the memory locations of some identifiers.
  - That is, we need to store memory addresses.

- Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.
  - Such variables that hold memory addresses are called pointers.
  - Since a pointer is a variable, its value is also stored in some memory location.
  - Such a variable is called pointer variable.

Concept of Pointer

- Suppose, we store the pointer of x in p

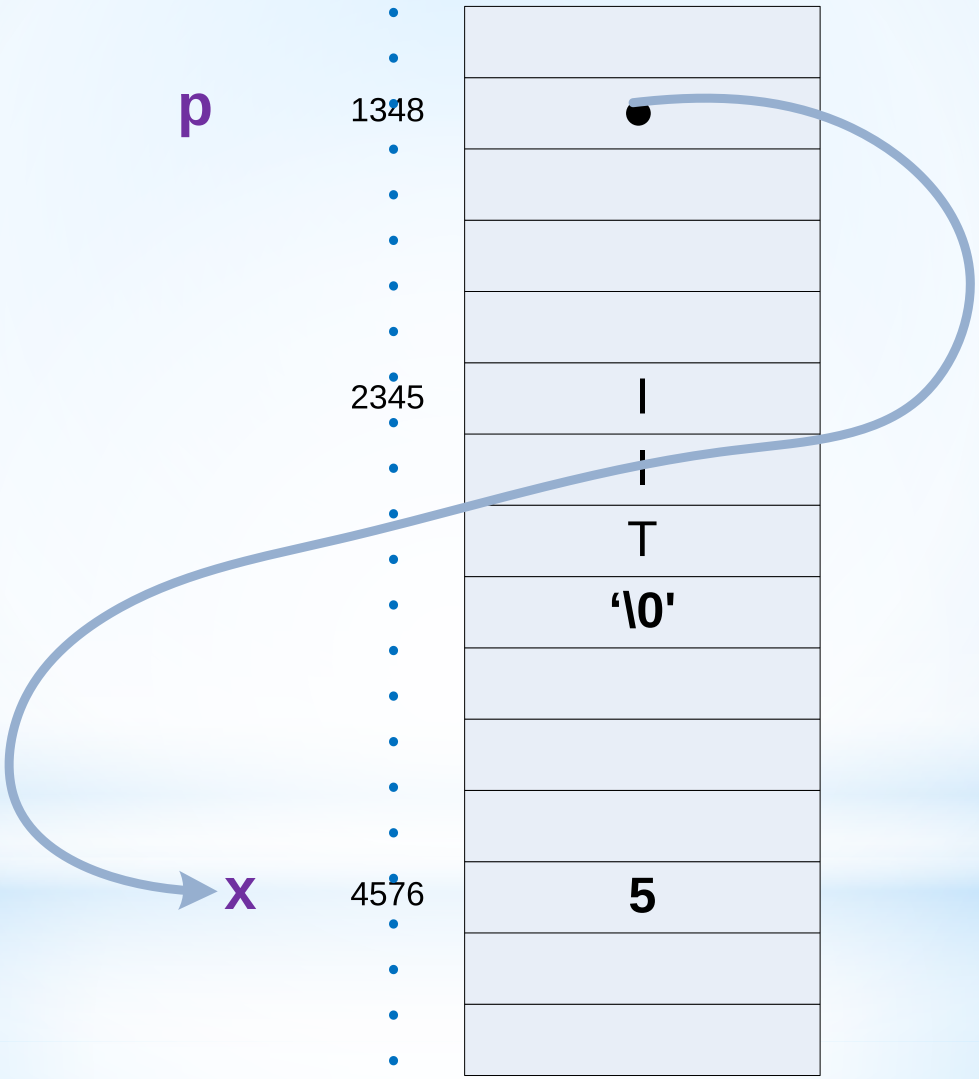
Variable	Address	Value
x	4576	5
p	1348	4576



In C, we write it as

```
p = &x;
```

Concept of Pointer



Applications of Pointer

- They have a number of useful applications.
  - Enables us to access a variable that is defined outside the function.
  - Can be used to pass information back and forth between a function and its reference point.
  - More efficient in handling data tables.
  - Reduces the length and complexity of a program.
  - Sometimes also increases the execution speed.

Pointer Manipulation in C

Declaration of a Pointer Variable

- Syntax in C

```
data_type  *pt-var_name;
```

- This tells the compiler three things about the variable `pt_var_name`
  - The asterisk * tells that the variable `pt_var_name` is a pointer variable
  - `pt_var_name` needs a memory location
  - `pt_var_name` points to a variable of type data_type

- Examples

```
int *x;         // A variable to store address of an integer variable
char *name;     // A variable to store address of a string variable
float *y;       // A variable to store address of a float variable
```

Initialization of a Pointer Variable

- Once a pointer variable `pt_var_name` is declared, it can be made to point to a variable `var_name` using an assignment statement such as

```
pt_var_name = & var_name
```

// Here, & is a unary operator

```
Example:
  int x = 179, *p;
  p = &x;
```

- A pointer variable can be initialized in its declaration itself.

Example:

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

Note:

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

is invalid!

Pointer Variables

```
#include <stdio.h>

main(){
   int a;
   float b, c;
   double d;
   char ch;
   a = 10; b = 2.5;
   c = 12.36; d = 12345.66;
   ch= ‘A’;
   printf(“a is stored in location %u \n”, &a);
printf(“b is stored in location %u \n”, &b);
printf(“c is stored in location %u \n”, &c);
printf(“d is stored in location %u \n”, &d);
printf(“ch is stored in location %u \n”, &ch) ;
}
```

Accessing a Variable through Pointer

- A pointer variable `pt_var_name` points a variable `var_name`. We can access the value stored in using the pointer.

- This is done by using another unary operator * (asterisk), usually known as the indirection operator.

```
value = * pt_var_name;
```

```
Example:
  int x = 179, y, *p;
  p = &x;
  y = *p;
```

Note:

`y = *(&x);` is a short-cut!

Pointer Arithmetic

- Like other variables, pointer variables can be used in arithmetic expressions. Suppose, `p1` is a pointer to `x` and `p2` is a pointer to `y`.

```
Examples:

  *p1 = *p2 + 10;        //   Same as x = y + 10;

   y = *p2 + 1;          //   Same as y = ++y;

  *p1 += 1;              //   Same as x = x + 1;

  ++(*p1);               //   Same as *p1 = *p1+1;

  *p1 = *p2;             //   Same as x = y;

   p1 = p2;              //   p1 and p2 both points to  y;
```

Pointer Arithmetic

- Following usage are illegal.

```
&235        // Pointing at constant is meaningless

int arr[20];
. . .
&arr        // Pointing at array name is not done.
            // arr itself points to the starting location of the array

&(a+b)      // Pointing at expression is illegal

p = 1024;   // Assigning an absolute address to a pointer variable
```

Pointers and Arrays

Pointers and Arrays

- When an arrays is declared, the complier allocates a base address and sufficient amount of memory to store all the elements of the array in a contagious memory locations.

- The base address is the location of the first element of the array (index = 0).

- In C, there is a strong relationship between pointers and arrays.
  - Any operation that can be achieved by array subscripting can also be done with pointers
  - The pointer version, in general, is faster.

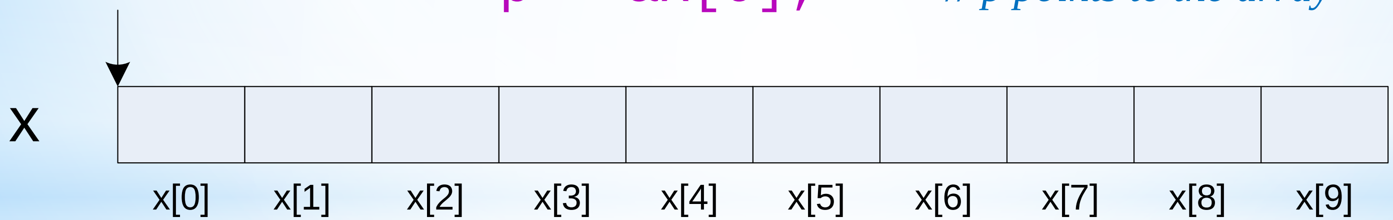
Pointers and Arrays

- Suppose, the is a declaration of an array of integers

```
int x[10];
```

- Name of the array, that is, x itself points to the starting location of the array. In other words

```
int *p;          // p is a pointer variable
p = &x[0];       // p points to the array
```



- `p` and `x` points to same memory location; however, `x` is not a pointer variable!
- `(p+i)` is the address of `x[i]`.

Pointers and Arrays

- If `p` points to an array `x`, then

  `p = x;` implies the current value of the pointer

  `*(p+i)` implies the content in `x[i]`

- Few things to be noted

  `x+i` is also identical to `p+i`

  `x[i]` can also be written as `*(x+i)`

  `&x[i]` and `x+i` are identical

  `p[i]` is identical to `*(p+i)`

  `p = x` and `p++` are legal *// Because, p is a pointer variable*

  `x = p` and `x++` are illegal *// name of an array is not a*

Pointers and Arrays

- When an array name is passed to a function, what is passed is the location of initial element.

- Within the called function, this argument is a local variable, and so an array name parameter is a pointer.

```
#include  <stdio.h>

int  arrayLen (char *a)
{
    int length;
    for (i=0; *s != ‘\0’; s++)
         length++;
    return (length);
}
```

Pointers and Arrays

- In the example, since `s` is a pointer, incrementing it is perfectly legal.
- `s++` has no effect on the content in the array, but just increments the private copy of the pointer.
- Following are all valid calls

```
#include  <stdio.h>

void  main()
{
    …..
    arrayLen(x);              // When, there exist say, char  x[100];

    arrayLen(“Debasis Samanta”);          // Passed as a string

    arrayLen(ptr);                   // When, there exist say, char  *ptr;
}
```

Pointers and Arrays

- Passing an entire array to a function

```
#include <stdio.h>

float findMean (int x[], int size)
{
      int i, sum = 0;
      for (i=0; i<size; i++)
         sum += x[i];
      return ((float)sum/size);
}

void main()
{
      int a[100], i, n;

      scanf (“%d”, &n);
      for (i=0; i<n; i++)
         scanf (“%d”, &a[i]);

      printf (“\n Mean is %d”,findMean(a,n);
}
```

Pointers and Arrays

- It is possible to pass a part of an array to a function.

Example:

Suppose, a function is defined as

`foo(int x[]);` or `foo(int *x);`

Then

```
foo(&a[i]);
```

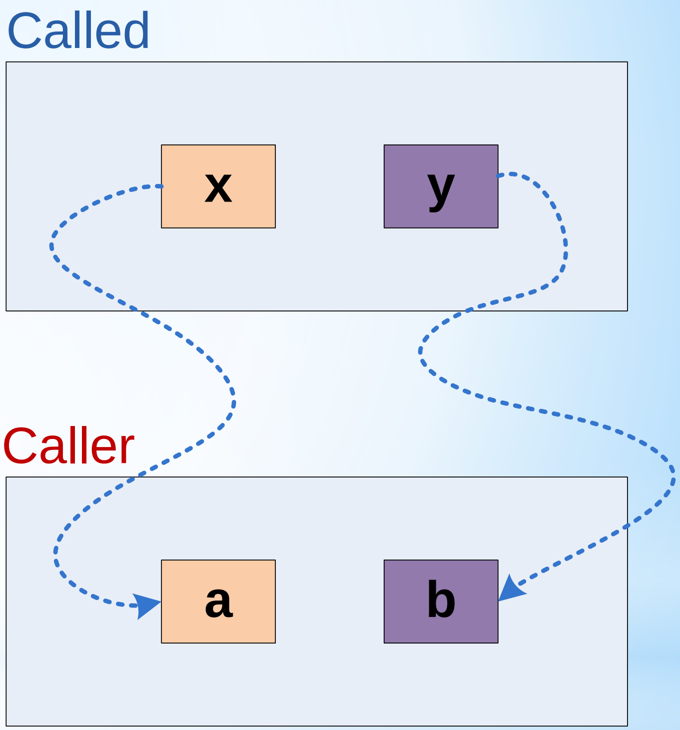
and

```
foo(a+i);
```

- Both pass to the function foo the address of the subarray that starts at a[i].

Pointers as Function Arguments

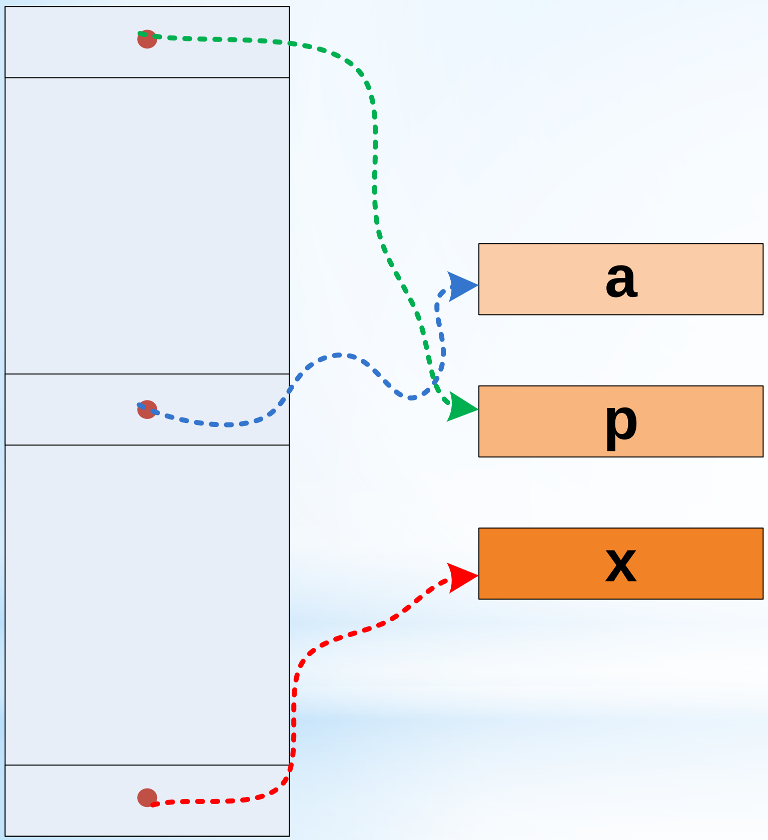
```
#include <stdio.h>

void swap (int *x, int *y)
{
      *x = *x + *y;
      *y = *x – *y;
      *x = *x – *y;

      return;
}

void main()
{
      int a, b;

      scanf (“a = %d, b = %d”, &a, &b);
      swap(&a, &b);

      printf (“a = %d, b = %d”, a, b);
}
```



Pointer Array

Pointer Array

- If an array stores pointers of some variables, then it is called a pointer array.




```
char * names[100];
```

// It stores pointers to 100 strings

```
int *marks[ ];
```

// It stores pointers to undefined `int` *values*

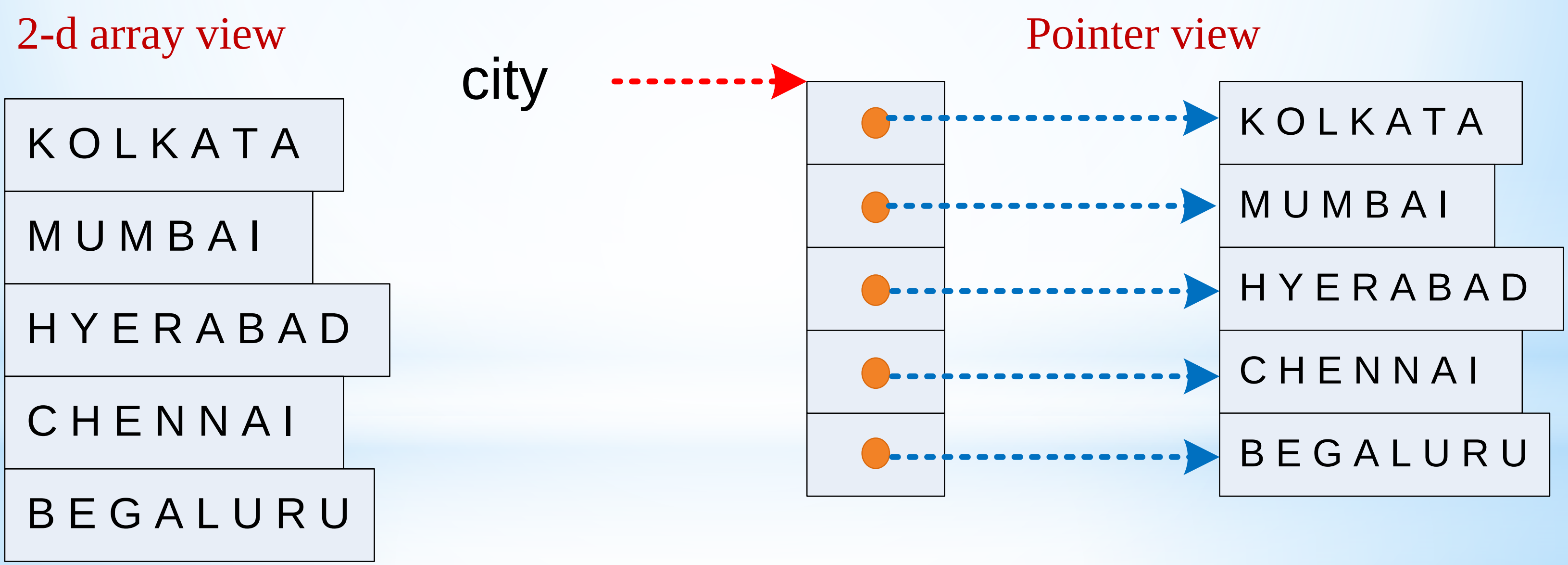
```
float **age;
```

// It stores pointers to undefined `float` *values*

Pointer and 2D Arrays

- We often use list of strings in many occasions
  - Names of cities, names of students, etc.

- Such a list of strings looks like a 2-d character array and better can be processed from the pointer view



Pointer and 2D Arrays

```
char city[20][15];        // 2-d array view: to store names of 20 cities
                          // of maximum 15 characters

char *city[20];           // Pointer view: to store names of 20 cities
                          // of maximum 15 characters

char *city[20] = {“Kolkata”, “Mumbai”, “Hyderabad”,
“Chennai, Bengaluru”};     // Initialization of  pointer array

printf(“%s”, city[i]);    // Using 2-d array view

printf(“%s”, city+i);     // Using pointer view
```

Command Line Arguments

Command Line Arguments

- It is a technique to supply input when the program is invoked

```
a.out Hello C!
findMin 55 33 66 88 44 22 11 99 77
```

- This can be done by mentioning two arguments in `main( )`

```
void main(int argc, char *argv[]){
    .........
    state the body
    .........
    return;
}
```

argc and argv

- **argc** : is an argument counter that counts the number of arguments on the command line
- **argv** : is an argument vector an represents an array of character pointers that point to the command line arguments

```
void main(int argc, char *argv[]){
   .........
   state the body
     .........
    return;
}
```

For a.out Hello C!

argc = 3

argv[0] ------→ a.out
argv[1] ------→ Hello
argv[2] ------→ C!

Command Line Arguments: Example

```
#include <stdio.h>

void main(int argc, char *argv[])
{
      int i;

    /* Print the values given as command line arguments */
      for(i=1;i< argc; i++)
          printf(“%s%s”,argv[i], (i<argc-1) ? “ “ : “”);

      printf(“\n”);
      return;
}
```

Command Line Arguments

- Rename the `a.out` to a command name, for example, `findMin` as if the main function is renamed as `findMin`
- `argc = 1` implies that there is no command line argument
- All command line arguments, if any, are read as string and store them in the pointer array `argv`
- Any input then converted into its appropriate type automatically at the time of their use in expression

Pointers to Functions

Pointers to Functions

- A function, like a variable, has an address location in the memory.
- It is therefore, possible to declare a pointer to a function
  - Which then can be used as an argument in another function

- A pointer to a function is declared as follows

```
<type> (*fptr)();
```
//This is the simplest way of declaration

  - This tells the complier that `fptr` is a pointer to a function which returns `<type>` value
- Alternatively, a pointer to a function also can be declared as

```
<type> (*fptr)(<type1>, <type2>, . . .);
```

Or

```
<type> (*fptr)(<type1><agrg1, <type2><arg2>, . . .);
```

Pointers to Functions : Example

Example

```
double (*p)();                      // p is a pointer to a function
double add(double x, double y);     // a function declaration
    ......
p = add;                            // pointer to function is assigned to p
    … …
add(a,b);                           // The function add() is called with its arguments

(*p)(a,b);                          // This is equivalent to add(a,b)
```

Passing Functions to other Functions

- A pointer to a function (say, *guest*) can be passed to another function (say, *host*) as an argument.
  - This allows the guest function to be transferred to host, as though the guest function were a variable.
- Successive calls to the host function can pass different pointers (i.e., different guest functions) to the host.
- When a host function accepts the name of a guest function as an argument, the formal argument declaration must identify that argument as a pointer to the guest function.

Passing Functions to other Functions

- **Declaring a guest function**
  - Guest function can be declared as usual

```
<type_g> gfName(<type1>(arg1), <type2><arg2>, . . .);
```

- **Declaring a host function**
  - Guest function can be declared as usual

Pointer to guest function

```
<type_h> hfName(..., <type_g>(*)( ),
        . . . data types of other arguments in host);
```

- Note that the indirection operator ‘*’ appears in parenthesis, to indicate a pointer to the guest function. Moreover, the data types of the guest function’s arguments follow in a separate pair of parenthesis, to indicate that they are the function’s arguments.

Passing Functions to other Functions

```
#include<stdio.h>

int process(int, int(*)(), char *);     // Declaration of a host function
int func1(int, int);                    // Declaration of a guest function
int func2(int, int);          // Declaration of another guest function

void main(){
   int i, j; char name[20]);
   ...
    i = process(i, func1, name);
   ...
    j = process(j, func2, name);

    return;
}
```

Contd…

Passing Functions to other Functions

```
int process(int x, int (*pf)(), char *s) {
        int a, b, c;
        ...
        c = (*pf)(a, b);                    // Access the guest function
        ...
        return(c);
}

int func1(int a, int b) {
      int c;
      ...                                   // Code in the guest function
      return(c);
}

int func2(int a, int b) {
      int d;
      ...                                   // Another code in the guest function
      return(d);
}
```

Pointers to Functions : Example

```
int *readDataN();                        // Read the numbers to be sorted
char *readDataS();                       // Read the string data to be sorted
int compN(int x, int y);                 // To compare two numbers
int compS(char *x, char *y);             // To compare two strings
void swap (void *, void *);              // To swap two values
void sort(void *p, int (*q)(void *,void *),void (*r)(void *,void *));

void main(int argc, char *argv[])
{
      void *pInt, *pString;
      if (argc > 1) && strcmp(argv[1],"-n")== 0){
         pInt = readDataN(); q = compN; r = swapN;
         sort(pInt, q, r);}
      else {pString = readDataS(); q = compS; r = swapS;
         sort(pString, q, r); }
   return;
}
```

Any question?

You may post your question(s) at the “Discussion Forum” maintained in the course Web page.

Problems to Ponder…

1. Is the following correct? If not, why?

   ```
   float x;   int *p; p = &x;
   ```

2. What amount of memory would be require to store the values of the two pointer variables `p` and `q` as declared below?

   ```
   float x, *p;    int y, *q;
   ```

3. Is the following statements valid? If not, why?

   ```
   int *count;   count = 1024;
   ```

4. How one can print the a) value stored in a pointer variable and b) memory location, where a pointer variable is stored?

5. A pointer essentially stores a memory location. Now, amount of byte to store a memory location of a variable of any type is same. Does a pointer array can store pointers of any type of variables?

Problems to Ponder…

6. Consider the following declarations.

```
int *x, *p;
char a[20];
p = &x;   x = 100;
```

Give the answer to the following.

a) How to print the address of x:

b) What *p indicates?

c) How to print the address where p is stored?

d) How to print the content (i.e., address) which is stored in p?

e) How to access the variable through p?

f) can we write p = 0xabc6, where the right-hand side is a hexa-decimal number?

Problems to Ponder…

7. Given that

int *p, x, y;

Tell (in terms of equivalent C statement) the implication of the following.

```
a)  p = &x;        y = *p:
b)  y = *&x;
c) *p = 255;
d) p++;
e) y += *p;
```

Problems to Ponder…

8. Consider the following statement

```
int a[10];

int *p;
```

With respect to the above declaration, which of the following statements is/are illegal?

a) `p = a;`

b) `a = p;`

c) `a+i;`

d) `a++`

e) `P + i;`

Problems to Ponder…

9. Consider the following program

```
include <stdio.h>
main() {
   int i = 3, *j, **k;
   j = &i;      k = &j;
   printf("a is %u", &i);
   printf("b is %u", j);
   printf("c is %u", k);
   printf("d is %u", &j);
   printf("e is %u", &k);
   printf("f is %d", i);
   printf("g is %d", *j);
   printf("h is %d", *(&i));
   printf("i is %d", **k);
}
```

What the output a, b, c, …, I signify? Draw an appropriate memory instance and the explain your answers.

Problems to Ponder…

10. Using command line arguments say `void main(in atrgc, char *argv[])` we pass string values as input to the program. What modification should be in the argument list, if any, if we have to pass the input other than string values?

11. What is the difference between the two declarations?

```
<type> (*f());
<type> *f();
```

Problems for Practice…

* **You can check the Moodle course management system for a set of problems for your own practice.**
  * Login to the Moodle system at http://cse.iitkgp.ac.in/
  * Select “PDS Spring-2017 (Theory) in the link “My Courses”
  * Go to Topic 7: Practice Sheet #07 : Pointer in C

* Solutions to the problems in Practice Sheet #07 will be uploaded in due time.

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.

Programming and Data Structures

Debasis Samanta

Computer Science & Engineering

Indian Institute of Technology Kharagpur

Spring-2017

Lecture #07

Memory Allocation Techniques

Today's discussion…

* Static Memory Allocation

* Dynamic Memory Allocation

* Functions in C for Memory Allocation

Memory Allocation

Why Memory Allocation?

- C language allows constants and variables to be processed
- All such variables should be maintained in primary memory at the time of execution.
- This needs that memory should be allocated to all variables.
- There are two ways to allocate memory for data in C
  - **Static allocation**
    - At the time of writing your program, you specify the memory requirement
  - **Dynamic allocation**
    - Allocate memory during run time as and when require

Static Memory Allocation

Static Memory Allocation

- **Static allocation**
  - Memory requirement should be specified at the time of writing programs
  - Once the memory allocated it cannot be altered. That is allocated memory remains fixed through out the entire run of the program
  - Sometimes we create data structures that are “fixed” and don’t need to grow or shrink.

```
int x;

char a[200][20];

x = 555; √
```

```
int x;

char a[200][20];

x = 555; ✕
```

We can not store > 200 names

Static Allocation: Pros and Cons

- Done at compile time.
- Global variables: variables declared “ahead of time,” such as fixed arrays.
- Lifetime
  - Entire runtime of program.
- Advantage
  - Efficient execution time.
- Disadvantage
  - If we declare more static data space than we need, we waste space.
  - If we declare less static space than we need, we are out of luck.

Dynamic Memory Allocation

Dynamic Memory Allocation

- Dynamic allocation (change in size)
  - Often, real world problems mean that we don’t know how much space to declare, as the number needed will change over time.
  - At other times, we want to increase and decrease the size of our data structures to accommodate changing needs.
  - To free space, when a variable is no more required

```
int x;

char a[200][20];

x = 555;
```

~~int x;~~

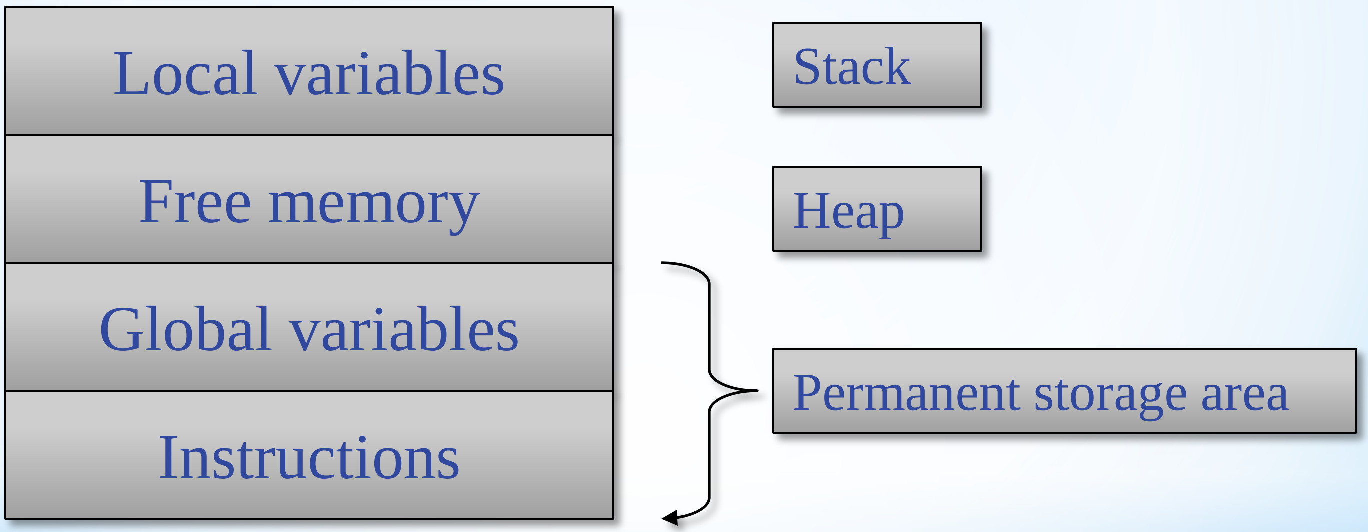
```
char a[500][20];
```

We can store > 200 names

Dynamic Allocation: Pros and Cons

- Done at run time.
- Data structures can grow and shrink to fit changing data requirements.
  - We can allocate (create) additional storage whenever we need them.
  - We can de-allocate (free/delete) dynamic space whenever we are done with them.
- Advantage:
  - We can always have exactly the amount of space required - no more, no less.

Memory Allocation Process in C



Memory Allocation Process in C

- The program instructions and the global variables are stored in a region known as permanent storage area.
- The local variables are stored in another area called stack.
- The memory space available for dynamic allocation during execution of the program is called heap.
  - This region is initially kept free.
  - The size of the heap keeps changing as a program runs.

Dynamic Memory Allocation

- Many a time, we face situations where data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.

- Such situations can be handled more easily and effectively using dynamic memory management techniques.

- C language requires the number of elements in an array to be specified at compile time.
  - Often leads to wastage or memory space or program failure.

- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

Memory Allocation Functions

- **`malloc()`**
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
- **`calloc()`**
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- **`free()`**
  - Frees previously allocated space.
- **`realloc()`**
  - Modifies the size of previously allocated space.

Allocating a Block of Memory

- A block of memory can be allocated using the function **`malloc()`**.
  - Reserves a block of memory of specified size and returns a pointer of type **`void`**.
  - The return pointer can be assigned to any pointer type.

- Syntax

```
ptr  =  (type *)  malloc (unsigned n);
```

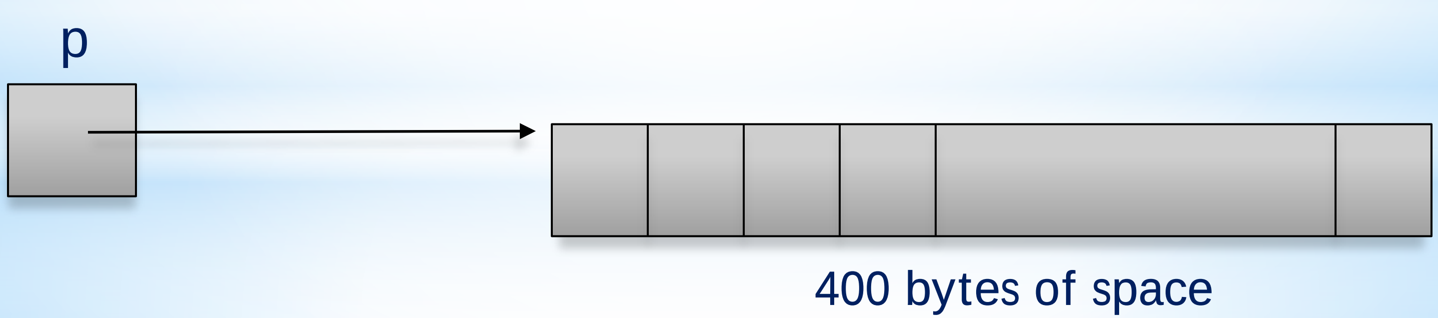
Returns a pointer to `n` bytes of uninitialized storage, or NULL if the request cannot be satisfied.

Allocating a Block of Memory

- **Examples**

```
p  =  (int *)  malloc (100 * sizeof (int));
```

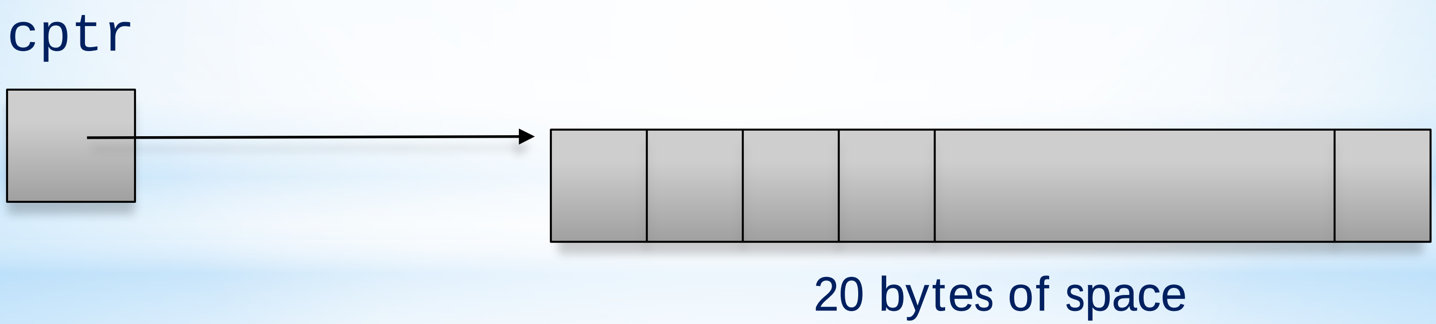
- A memory space equivalent to “100 times the size of an int” bytes is reserved.
- The address of the first byte of the allocated memory is assigned to the pointer `p` of type `int`.



Allocating a Block of Memory

```
cptr = (char *) malloc (20) ;
```

- A memory space of 20 bytes is reserved.
- The address of the first byte of the allocated memory is assigned to the pointer `cptr` of type `char`.



Points to Note

- `malloc()` always allocates a block of contiguous bytes.
- The allocation can fail if sufficient contiguous memory space is not available.
- If it fails, `malloc()` returns `NULL`.

Example: malloc()

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
   int i, N;
   float *height;
   float sum = 0, avg;
   printf("Input the number of students. \n");
      scanf("%d",&N);
   height=(float *)malloc(N * sizeof(float));
   printf("Input heights for %d students \n", N);
   for(i=0;i<N;i++)
      scanf("%f",&height[i]);
   for(i=0;i<N;i++)
      sum += height[i];
   avg = sum/(float) N;
   printf("Average height= %f \n", avg);
}
```

Output!
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000

calloc()

- The C library function

```
void *calloc(unsigned n, unsigned size)
```

  - Allocates the requested memory and returns a pointer to it.

- Allocates a block of memory for an array of `n` elements, each of them `size` bytes long, and initializes all its bits to zero.

Example

```
int n;
int *x;
. . .
x = (int *) calloc(n, sizeof(int));
```

$\rightarrow$ `int x[n];`

calloc() versus malloc()

```
void *malloc (unsigned n);

void *calloc(unsigned n, unsigned size)
```

- **`malloc()`** takes a single argument (memory required in bytes), while **`calloc()`** needs two arguments.

- **`malloc()`** does not initialize the memory allocated, while **`calloc()`** initializes the allocated memory to ZERO.

Example: calloc()

```
#include <stdio.h> /* printf, scanf, NULL */
#include <stdlib.h> /* calloc, exit, free */

int main ()
{
   int i, n;
   int *pData;
   printf ("Amount of numbers to be entered: ");
      scanf ("%d",&n);
   pData = (int*) calloc (n, sizeof(int));
   if (pData == NULL) exit (1);
   for (i=0;i<i;i++)
   {
      printf ("Enter number #%d: ",i+1);
         scanf ("%d",&pData[i]);
   }
   printf ("You have entered: ");
   for (i=0;i<n;i++)
      printf ("%d ",pData[i]);
   free (pData);
return 0;
}
```

Output!
Amount of numbers to be entered: 5
Enter number #1: 23
Enter number #2: 31
Enter number #3: 23
Enter number #4: 45
Enter number #5: 32
You have entered: 23 31 23 45 32

Releasing the Used Space

- When we no longer need the data stored in a block of memory, we may release the block for future use.

- How?
  - By using the **`free()`** function.

- Syntax

```
free (ptr) ;
```

where `ptr` is a pointer to a memory block which has been already created using **`malloc()` or `calloc()` or `realloc()` ;**

`realloc()`: Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed.
  - Memory allocated is larger than necessary.

- How?
  - By using the **`realloc()`** function.

- If the original allocation is done by the statement

```
ptr = malloc (size);
```

- Then reallocation of space may be done as

```
ptr = realloc (ptr, newsize) ;
```

`realloc()`: Altering the Size of a Block

- The new memory block may or may not begin at the same place as the old one.
  - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns **`NULL`**. But, it does not free the original block.

Example: realloc()

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   int *pa, *pb, n; /* allocate an array of 10 int */
   pa = (int *)malloc(10 * sizeof (int));
   if(pa) {
      printf("%u bytes allocated. Storing integers: ", 10*sizeof(int));
      for(n = 0; n < 10; ++n)
         printf("%d ", pa[n] = n);
   }                                  // reallocate array to a larger size
   pb = (int *)realloc(pa, 1000000 * sizeof(int));
   if(pb) {
      printf("\n%u bytes are allocated, after the first 10 integers are: ",
1000000*sizeof(int));
      for(n = 0; n < 10; ++n)
         printf("%d ", pb[n]); // show the array
      free(pb);
   }
   else { // if realloc failed, the original pointer needs to be freed
       free(pa);
   }
   return 0;
}
```

Output!
40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9
4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9

Memory Allocation for 2D Array

Version 1: Using a single pointer …

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *a2D;                 // Pointer to an array of integers
   int i, j, row, column;

   scanf("Enter number of rows: %d", &row);
   scanf("Enter number of columns: %d", &column);

   a2D = (int *) malloc(row*column*sizeof(int);     // Allocate net memory required for the 2D array

   for(i=0; i<row; i++)                // Put the data into the array…
       for(j=0; j<column; j++) {
            printf("\n a2D[%d][%d] = ",row, column); scanf("%d", arr +i*row+column);
       }

   return 0;
}
```

Memory Allocation for 2D Array

Version 2: Using an array of pointers …

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int i, j, row, column;

   scanf("Enter number of rows: %d", &row);
   scanf("Enter number of columns: %d", &column);

   int *a2D[row];           //  Declaration of array of pointers to integers

   for(i=0; i<row; i++)
        a2D[i] = (int *) malloc(column*sizeof(int);     // Allocate memory for a row

   for(i=0; i<row; i++)               // Put the data into the array…
      for(j=0; j<column; j++) {
         printf("\n a2D[%d][%d] = ",row, column); scanf("%d", arr +i*row+column);
       }
   return 0;
}
```

Memory Allocation for 2D Array

Version 3: Using pointer to a pointer …

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int **a2D;            //  Declaration of array of pointers to integers
   int i, j, row, column;

   scanf(“Enter number of rows: %d”, &row);
   scanf(“Enter number of columns: %d”, &column);

    *a2D = (int **) malloc(row * sizeof(int *)); // Allocate memory for the pointer array

     for(i=0; i<row; i++)
        a2D[i] = (int *) malloc(column*sizeof(int);     // Allocate memory for a row

     for(i=0; i<row; i++)            // Put the data into the array…
       for(j=0; j<column; j++) {
            printf(“\n a2D[%d][%d] = “,row, column); scanf(“%d”, arr +i*row+column);
       }
   return 0;
}
```

Any question?

You may post your question(s) at the “Discussion Forum” maintained in the course Web page.

Problems to Ponder…

1. What will happen if you call the following

```
malloc (n);          if n = 0
calloc (n1 ,n2);     if n1 = 0 or , n2 = 0
malloc(-100);
```

2. How to allocate memory for the following 3-D array

```
int     x[m][n][p];
```

   for any integer number m, n and p.

Problems to Ponder…

3. Using dynamic memory allocation technique, how you can allocate only non-zero elements in the following sparse matrices:

`* are non zero elements`

$$\begin{bmatrix} * & - & - & - & - & - & - & - \\ - & * & - & - & - & - & - & - \\ - & - & * & - & - & - & - & - \\ - & - & - & * & - & - & - & - \\ - & - & - & - & * & - & - & - \\ - & - & - & - & - & * & - & - \\ - & - & - & - & - & - & * & - \\ - & - & - & - & - & - & - & * \end{bmatrix}$$

(a) Diagonal Matrix

$$\begin{bmatrix} * & * & - & - & - & - & - & - \\ * & * & * & - & - & - & - & - \\ - & * & * & * & - & - & - & - \\ - & - & * & * & * & - & - & - \\ - & - & - & * & * & * & - & - \\ - & - & - & - & * & * & * & - \\ - & - & - & - & - & * & * & * \\ - & - & - & - & - & - & * & * \end{bmatrix}$$

(b) Tri-Diagonal Matrix

Problems to Ponder…

*** are non zero elements**

*	-	-	-	-	-	-	-
*	*	-	-	-	-	-	-
*	*	*	-	-	-	-	-
*	*	*	*	-	-	-	-
*	*	*	*	*	-	-	-
*	*	*	*	*	*	-	-
*	*	*	*	*	*	*	-
*	*	*	*	*	*	*	*

(c) Lower Triangular Matrix

*	*	*	*	*	*	*	*
-	*	*	*	*	*	*	*
-	-	*	*	*	*	*	*
-	-	-	*	*	*	*	*
-	-	-	-	*	*	*	*
-	-	-	-	-	*	*	*
-	-	-	-	-	-	*	*
-	-	-	-	-	-	-	*

(d) Upper Triangular Matrix