Pointers

Introduction

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

Basic Concept

- **Within the computer 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, 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.**

Contd.

- **Consider the statement**

  int   xyz = 50;

  - **This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.**
  - **Suppose that the address location chosen is 1380.**

xyz	➔	**variable**
50	➔	**value**
1380	➔	**address**

Contd.

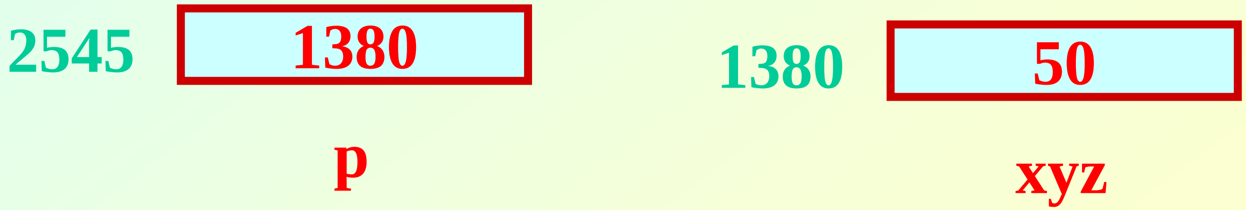
- **During execution of the program, the system always associates the name xyz with the address 1380.**
  - **The value 50 can be accessed by using either the name xyz or the address 1380.**
- **Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.**
  - **Such variables that hold memory addresses are called pointers.**
  - **Since a pointer is a variable, its value is also stored in some memory location.**

Contd.

- Suppose we assign the **address of xyz** to a variable **p**.
  - **p** is said to point to the variable **xyz**.

Variable	Value	Address
xyz	50	1380
p	1380	2545

p = &xyz;



Accessing the Address of a Variable

- The address of a variable can be determined using the ‘&’ operator.
  – The operator ‘&’ immediately preceding a variable returns the address of the variable.
- Example:

```
p = &xyz;
```

  – The address of xyz (1380) is assigned to p.
- The ‘&’ operator can be used only with a simple variable or an array element.

```
&distance
&x[0]
&x[i-2]
```

Contd.

- **Following usages are illegal:**

```
&235
```

  - Pointing at constant.

```
int   arr[20];
   :
&arr;
```

  - Pointing at array name.

```
&(a+b)
```

  - Pointing at expression.

Example

```
#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  (“%d is stored in location %u \n”,  a,  &a) ;
   printf  (“%f is stored in location %u \n”,  b,  &b) ;
   printf  (“%f is stored in location %u \n”,  c,  &c) ;
   printf  (“%ld is stored in location %u \n”,  d,  &d) ;
   printf  (“%c is stored in location %u \n”,  ch,  &ch) ;
}
```

Output:

10 is stored in location 3221224908 **a**

2.500000 is stored in location 3221224904 **b**

12.360000 is stored in location 3221224900 **c**

12345.660000 is stored in location 3221224892 **d**

A is stored in location 3221224891 **ch**

Incidentally variables a,b,c,d and ch are allocated to contiguous memory locations.

Pointer Declarations

- **Pointer variables must be declared before we use them.**
- **General form:**

  data_type   *pointer_name;

  Three things are specified in the above declaration:

  1. **The asterisk (*) tells that the variable pointer_name is a pointer variable.**
  2. **pointer_name needs a memory location.**
  3. **pointer_name points to a variable of type data_type.**

Contd.

- **Example:**

```
int    *count;
float  *speed;
```

- **Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:**

```
int     *p,  xyz;
:
p = &xyz;
```

  – **This is called pointer initialization.**

Things to Remember

- **Pointer variables must always point to a data item of the same type.**

```
float   x;
int    *p;
:                              ➔ will result in erroneous output
p = &x;
```

- **Assigning an absolute address to a pointer variable is prohibited.**

```
int   *count;
:
count = 1268;
```

Accessing a Variable Through its Pointer

- **Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the indirection operator (*).**

```
int   a, b;
int   *p;
:
p = &a;
b = *p;
```

Equivalent to → **b = a**

Example 1

```
#include  <stdio.h>
main()
{
   int   a, b;
   int   c = 5;
   int   *p;

   a  =  4  *  (c  +  5) ;

   p  =  &c;
   b  =  4  *  (*p  +  5) ;
   printf  ("a=%d  b=%d \n",  a, b) ;
}
```

Equivalent

Example 2

```
#include  <stdio.h>
main()
{
    int  x, y;
    int  *ptr;

    x = 10 ;
    ptr = &x ;
    y = *ptr ;
    printf  (“%d is stored in location %u \n”,  x,  &x) ;
    printf  (“%d is stored in location %u \n”,  *&x,  &x) ;
    printf  (“%d is stored in location %u \n”,  *ptr,  ptr) ;
    printf  (“%d is stored in location %u \n”,  y,  &*ptr) ;
    printf  (“%u is stored in location %u \n”,  ptr, &ptr) ;
    printf  (“%d is stored in location %u \n”,  y,  &y) ;

   *ptr = 25;
    printf  (“\nNow x = %d \n”, x);
}
```

***&x⇔x**

ptr=&x;
&x⇔&*ptr

Output:

10 is stored in location 3221224908

10 is stored in location 3221224908

10 is stored in location 3221224908

10 is stored in location 3221224908

3221224908 is stored in location 3221224900

10 is stored in location 3221224904

Now x = 25

Address of x:	3221224908
Address of y:	3221224904
Address of ptr:	3221224900

Pointer Expressions

- **Like other variables, pointer variables can be used in expressions.**
- **If p1 and p2 are two pointers, the following statements are valid:**

```
sum  =  *p1  +  *p2 ;
prod  =  *p1  *  *p2 ;
prod  =  (*p1)  *  (*p2) ;
*p1  =  *p1  +  2;
x  =  *p1  /  *p2  +  5 ;
```

Contd.

- **What are allowed in C?**
  - **Add an integer to a pointer.**
  - **Subtract an integer from a pointer.**
  - **Subtract one pointer from another (related).**
    - **If p1 and p2 are both pointers to the same array, them p2–p1 gives the number of elements between p1 and p2.**
- **What are not allowed?**
  - **Add two pointers.**

    ```
    p1 = p1 + p2 ;
    ```
  - **Multiply / divide a pointer in an expression.**

    ```
    p1 = p2 / 5 ;
    p1 = p1 – p2 * 10 ;
    ```

Scale Factor

- **We have seen that an integer value can be added to or subtracted from a pointer variable.**

```
int   *p1,  *p2 ;
int   i,  j;
:
p1  =  p1  +  1 ;
p2  =  p1  +  j ;
p2++ ;
p2  =  p2  –  (i + j) ;
```

- **In reality, it is not the integer value which is added/subtracted, but rather the scale factor times the value.**

Contd.

Data Type	Scale Factor
char	1
int	4
float	4
double	8

– **If p1 is an integer pointer, then**

p1++

will increment the value of p1 by 4.

Returns no. of bytes required for data type representation

```
#include  <stdio.h>
main()
{
   printf ("Number of bytes occupied by int is %d \n", sizeof(int));
   printf ("Number of bytes occupied by float is %d \n", sizeof(float));
   printf ("Number of bytes occupied by double is %d \n", sizeof(double));
   printf ("Number of bytes occupied by char is %d \n", sizeof(char));
}
```

Output:

Number of bytes occupied by int is 4
Number of bytes occupied by float is 4
Number of bytes occupied by double is 8
Number of bytes occupied by char is 1

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called call-by-reference (or by address or by location).
- Normally, arguments are passed to a function by value.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

Example: passing arguments by value

```
#include  <stdio.h>
main()
{
    int  a, b;
    a = 5 ;   b = 20 ;
    swap (a, b) ;
    printf  ("\n a = %d,  b = %d", a, b);
}

void   swap  (int  x, int  y)
{
    int  t ;
    t = x ;
    x = y ;
    y = t ;
}
```

a and b do not swap

x and y swap

Output

a = 5, b = 20

Example: passing arguments by reference

```
#include  <stdio.h>
main()
{
     int  a, b;
     a = 5 ;   b = 20 ;
     swap (&a, &b) ;
     printf  ("\n a = %d,  b = %d", a, b);
}

void   swap  (int  *x, int  *y)
{
     int  t ;
     t = *x ;
     *x = *y ;
     *y = t ;
}
```

***(&a) and *(&b) swap**

***x and *y swap**

Output

a = 20, b = 5

scanf Revisited

```
int   x,  y ;
printf  (“%d %d %d”,  x, y, x+y) ;
```

- **What about scanf ?**

```
scanf   (“%d %d %d”, x, y, x+y) ;
```
NO

```
scanf   (“%d %d”, &x, &y) ;
```
YES

Example: Sort 3 integers

- **Three-step algorithm:**
  1. **Read in three integers x, y and z**
  2. **Put smallest in x**
     - **Swap x, y if necessary; then swap x, z if necessary.**
  3. **Put second smallest in y**
     - **Swap y, z if necessary.**

Contd.

```
#include  <stdio.h>
main()
{
     int  x, y, z ;
     ………..
     scanf  (“%d %d %d”, &x, &y, &z) ;
     if  (x > y)   swap (&x, &y);
     if  (x > z)   swap (&x, &z);
     if  (y > z)   swap (&y, &z) ;
     ………..
}
```

sort3 as a function

```
#include  <stdio.h>
main()
{
     int  x, y, z ;
     ………..
     scanf  ("%d %d %d", &x, &y, &z) ;
     sort3  (&x, &y, &z) ;
     ………..
}

void   sort3  (int *xp,  int *yp,  int *zp)
{
     if  (*xp > *yp)   swap (xp, yp);
     if  (*xp > *zp)   swap (xp, zp);
     if  (*yp > *zp)   swap (yp, zp);
}
```

xp/yp/zp are pointers

Contd.

- **Why no '&' in swap call?**
  - **Because xp, yp and zp are already pointers that point to the variables that we want to swap.**

Pointers and Arrays

- **When an array is declared,**
  - **The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.**
  - **The base address is the location of the first element (index 0) of the array.**
  - **The compiler also defines the array name as a constant pointer to the first element.**

Example

- **Consider the declaration:**

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

  – Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Contd.

x ⇔ &x[0] ⇔ 2500 ;

- p = x; and p = &x[0]; are equivalent.
- We can access successive values of x by using p++ or p- - to move from one element to another.

- **Relationship between p and x:**

p = &x[0] = 2500

p+1 = &x[1] = 2504

p+2 = &x[2] = 2508

p+3 = &x[3] = 2512

p+4 = &x[4] = 2516

*(p+i) gives the value of x[i]

Example: function to find average

```
#include <stdio.h>
main()
{
    int  x[100], k, n ;

    scanf  ("%d", &n) ;

    for  (k=0; k<n; k++)
        scanf  ("%d", &x[k]) ;

    printf  ("\nAverage is %f",
                            avg (x, n));
}
```

int *array

```
float  avg  (int array[ ],int  size)
{
    int  *p, i , sum = 0;

    p = array ;

    for  (i=0; i<size; i++)
        sum = sum + *(p+i);

    return  ((float) sum / size);
}
```

p[i]

Structures Revisited

- **Recall that a structure can be declared as:**

```
struct   stud   {
        int    roll;
        char  dept_code[25];
        float  cgpa;
    };
struct  stud  a, b, c;
```

- **And the individual structure elements can be accessed as:**

```
a.roll ,  b.roll ,  c.cgpa , etc.
```

Arrays of Structures

- **We can define an array of structure records as**

```
struct stud class[100] ;
```

- **The structure elements of the individual records can be accessed as:**

```
class[i].roll
class[20].dept_code
class[k++].cgpa
```

Example: Sorting by Roll Numbers

```
#include <stdio.h>
struct  stud
{
    int   roll;
    char  dept_code[25];
    float  cgpa;
 };

main()
{
    struc  stud  class[100], t;
    int  j, k, n;

    scanf  ("%d", &n);
                    /* no. of students */
```

```
for  (k=0; k<n; k++)
        scanf ("%d %s %f", &class[k].roll,
              class[k].dept_code, &class[k].cgpa);
   for  (j=0; j<n-1; j++)
        for  (k=j+1; k<n; k++)
          {
              if  (class[j].roll > class[k].roll)
                  {
                      t = class[j] ;
                      class[j] = class[k] ;
                      class[k] = t
                  }
           }
   <<<< PRINT THE RECORDS >>>>
}
```

Pointers and Structures

- **You may recall that the name of an array stands for the address of its zero-th element.**
  - **Also true for the names of arrays of structure variables.**
- **Consider the declaration:**

```
struct  stud  {
            int   roll;
            char  dept_code[25];
            float  cgpa;
        }  class[100],  *ptr ;
```

- The name class represents the address of the zero-th element of the structure array.
- ptr is a pointer to data objects of the type struct stud.

- The assignment

```
ptr = class ;
```

will assign the address of class[0] to ptr.

- When the pointer ptr is incremented by one (ptr++) :
  - The value of ptr is actually increased by sizeof(stud).
  - It is made to point to the next record.

- **Once ptr points to a structure variable, the members can be accessed as:**

```
ptr –> roll ;
ptr –> dept_code ;
ptr –> cgpa ;
```

  - **The symbol "–>" is called the arrow operator.**

Example

```c
#include <stdio.h>

typedef struct {
            float real;
            float imag;
           } _COMPLEX;
```

```c
swap_ref(_COMPLEX *a, _COMPLEX *b)
{
 _COMPLEX tmp;
  tmp=*a;
  *a=*b;
  *b=tmp;
}
```

```c
print(_COMPLEX *a)
{
 printf("(%f,%f)\n",a->real,a->imag);
}
```

```
(10.000000,3.000000)
(-20.000000,4.000000)
(-20.000000,4.000000)
(10.000000,3.000000)
```

```c
main()
{
 _COMPLEX x={10.0,3.0}, y={-20.0,4.0};

  print(&x); print(&y);
  swap_ref(&x,&y);
  print(&x); print(&y);
}
```

A Warning

- **When using structure pointers, we should take care of operator precedence.**
  - **Member operator "." has higher precedence than "*".**
    - **ptr –> roll and (*ptr).roll mean the same thing.**
    - ***ptr.roll will lead to error.**
  - **The operator "–>" enjoys the highest priority among operators.**
    - **++ptr –> roll will increment roll, not ptr.**
    - **(++ptr) –> roll will do the intended thing.**

Structures and Functions

- **A structure can be passed as argument to a function.**
- **A function can also return a structure.**
- **The process shall be illustrated with the help of an example.**
  - **A function to add two complex numbers.**

Example: complex number addition

```
#include  <stdio.h>
struct  complex  {
                    float  re;
                    float  im;
                  };

main()
{
   struct  complex  a, b, c;
   scanf  ("%f %f", &a.re, &a.im);
   scanf  ("%f %f", &b.re, &b.im);
   c  =  add (a, b) ;
   printf  ("\n %f %f", c.re, c.im);
}
```

```
struct  complex  add  (x, y)
struct complex  x, y;
{
    struct  complex  t;

    t.re = x.re + y.re ;
    t.im = x.im + y.im ;
    return (t) ;
}
```

Example: Alternative way using pointers

```
#include  <stdio.h>
struct  complex  {
                    float  re;
                    float  im;
                  };

main()
{
   struct  complex  a, b, c;
   scanf  (“%f %f”, &a.re, &a.im);
   scanf  (“%f %f”, &b.re, &b.im);
   add (&a, &b, &c) ;
   printf  (“\n %f %f”, c,re, c.im);
}
```

```
void  add  (x, y, t)
struct complex  *x, *y, *t;
{
    t->re = x->re + y->re ;
    t->im = x->im + y->im ;
}
```

Dynamic Memory Allocation

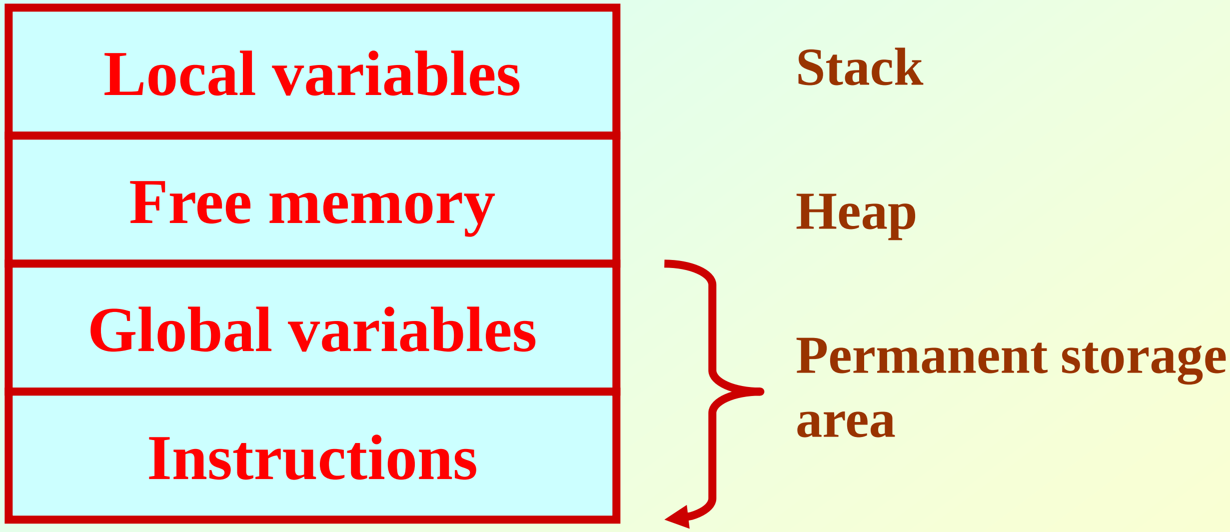
Basic Idea

- **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.**

Contd.

- **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 Process in C



Contd.

- **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 between these two areas is available for dynamic allocation during execution of the program.**
  - **This free region is called the heap.**
  - **The size of the heap keeps changing**

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.**
- **General format:**

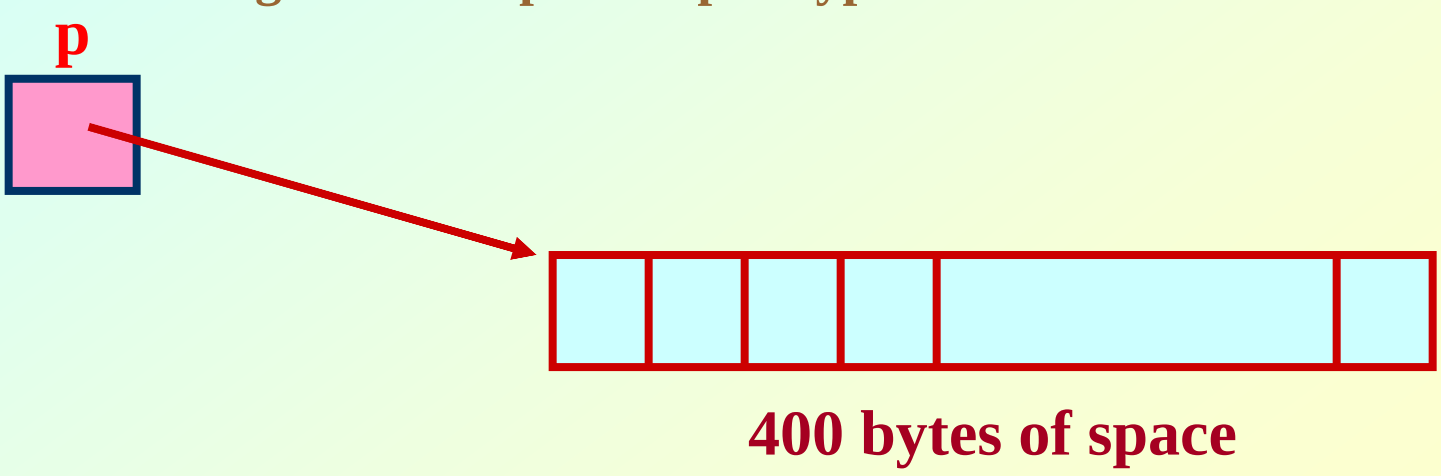
```
ptr  =  (type *)  malloc (byte_size) ;
```

Contd.

- 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.



Contd.

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

- Allocates 10 bytes of space for the pointer cptr of type char.

```
sptr  =  (struct stud *)  malloc (10 *
                                  sizeof (struct stud));
```

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

```
#include <stdio.h>

main()
{
  int i,N;
  float *height;
  float sum=0,av

  printf("Input the number of students. \n");
  scanf("%d",&N);

  height=(float *) malloc(N * sizeof(float));
```

```
printf("Input heights for %d
udents \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);
}
```

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

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.**
- **General format:**

free (ptr) ;

where ptr is a pointer to a memory block which has been already created using malloc.

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) ;
```

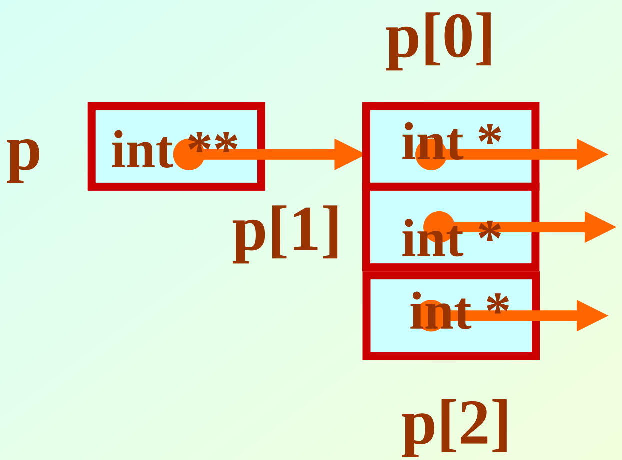
Contd.

- 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 and frees the original block.

Pointer to Pointer

- **Example:**

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
```



2-D Array Allocation

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

int **allocate(int h, int w)
  {
    int **p;
    int i,j;

   p=(int **) calloc(h, sizeof (int *) );
    for(i=0;i<h;i++)
     p[i]=(int *) calloc(w,sizeof (int));
    return(p);
  }
```

Allocate array of pointers

Allocate array of integers for each row

```c
void read_data(int **p,int h,int w)
 {
     int i,j;
     for(i=0;i<h;i++)
     for(j=0;j<w;j++)
       scanf ("%d",&p[i][j]);
 }
```

Elements accessed like 2-D array elements.

2-D Array: Contd.

```
void print_data(int **p,int h,int w)
 {
   int i,j;
    for(i=0;i<h;i++)
    {
    for(j=0;j<w;j++)
      printf("%5d ",p[i][j
     printf("\n");
    }
}
```

```
main()
{
  int **p;
  int M,N;

  printf("Give M and N \n");
  scanf("%d%d",&M,&N);
  p=allocate(M,N);
  read_data(p,M,N);
  printf("\n The array read as \n");
  print_data(p,M,N);
```

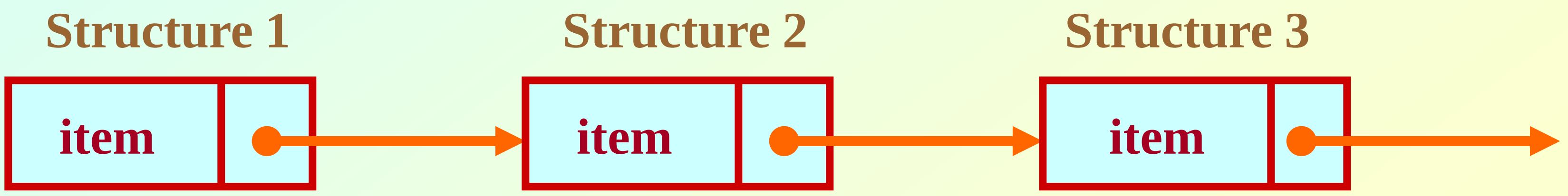
```
Give M and N
3 3
1 2 3
4 5 6
7 8 9

 The array read as
    1    2    3
    4    5    6
    7    8    9
```

Linked List :: Basic Concepts

- **A list refers to a set of items organized sequentially.**
  - **An array is an example of a list.**
    - **The array index is used for accessing and manipulation of array elements.**
  - **Problems with array:**
    - **The array size has to be specified at the beginning.**
    - **Deleting an element or inserting an element may require shifting of elements.**

Contd.

- **A completely different way to represent a list:**
  - **Make each item in the list part of a structure.**
  - **The structure also contains a pointer or link to the structure containing the next item.**
  - **This type of list is called a linked list.**



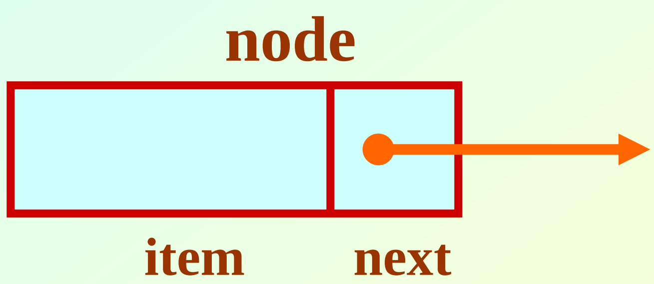
Contd.

- **Each structure of the list is called a node, and consists of two fields:**
  - **One containing the item.**
  - **The other containing the address of the next item in the list.**
- **The data items comprising a linked list need not be contiguous in memory.**
  - **They are ordered by logical links that are stored as part of the data in the structure itself.**
  - **The link is a pointer to another structure of the same type.**

Contd.

- Such a structure can be represented as:

```
struct  node
 {
    int    item;
    struct node  *next;
 } ;
```



- Such structures which contain a member field pointing to the same structure type are called **self-referential structures**.

Contd.

- **In general, a node may be represented as follows:**

```
struct  node_name
 {
    type  member1;
    type member2;
     ………
    struct  node_name  *next;
 };
```

Illustration

- **Consider the structure:**

```
struct  stud
 {
     int  roll;
     char  name[30];
     int  age;
     struct  stud  *next;
 };
```

- **Also assume that the list consists of three nodes n1, n2 and n3.**

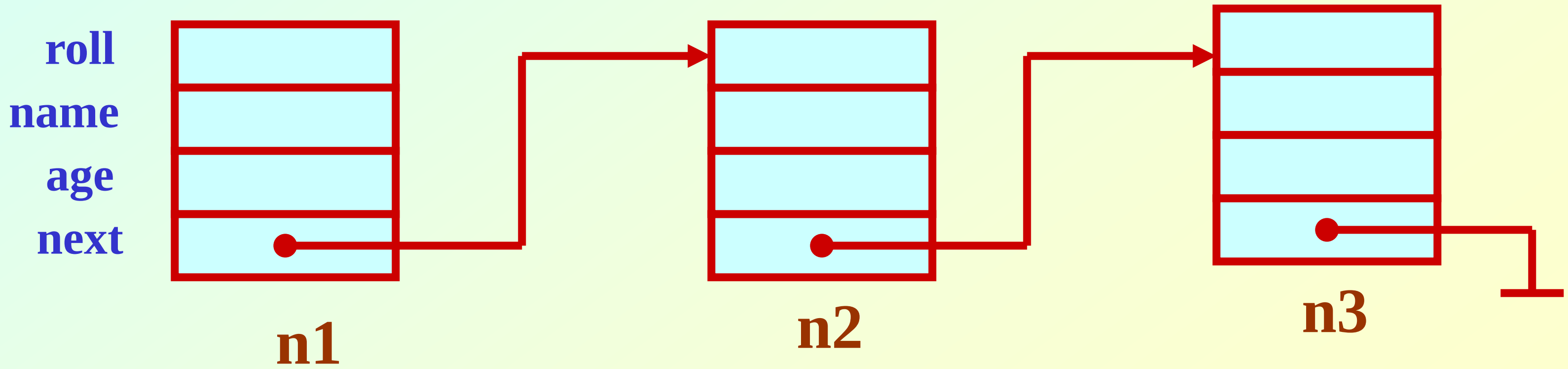
```
struct  stud  n1, n2, n3;
```

Contd.

- To create the links between nodes, we can write:

```
n1.next  =  &n2 ;
n2.next  =  &n3 ;
n3.next  =  NULL ;   /* No more nodes follow */
```

- Now the list looks like:



Example

```
#include  <stdio.h>
struct  stud
 {
    int  roll;
    char  name[30];
    int  age;
    struct  stud  *next;
 };

main()
{
   struct  stud  n1, n2, n3;
   struct  stud  *p;

   scanf (“%d %s %d”, &n1.roll,
                      n1.name, &n1.age);
   scanf (“%d %s %d”, &n2.roll,
                      n2.name, &n2.age);
   scanf (“%d %s %d”, &n3.roll,
                      n3.name, &n3.age);
```

```
   n1.next  =  &n2 ;
   n2.next  =  &n3 ;
   n3.next  =  NULL ;

/* Now traverse the list and print
  the elements */

   p  =  n1 ;   /* point to 1st element */
   while  (p != NULL)
   {
      printf (“\n %d %s %d”,
        p->roll, p->name, p->age);
        p  =  p->next;
   }
}
```