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

```
#include <stdio.h>

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

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

```
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)
```

```
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

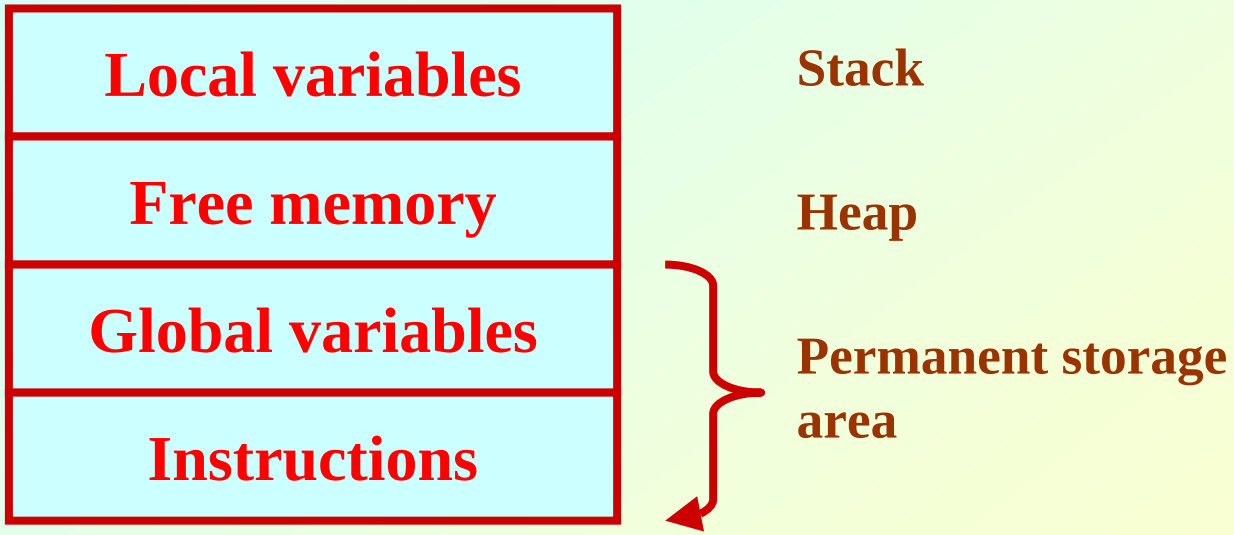
Basic Idea

- **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:**

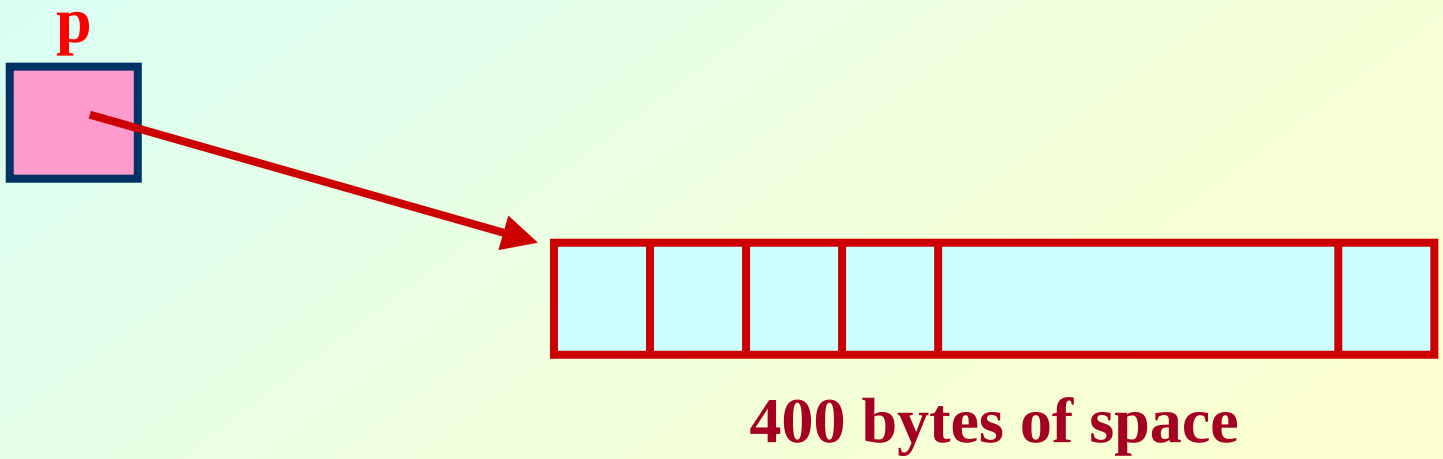
```
ptr = (type *) malloc (byte_size) ;
```

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,avg

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

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

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

```
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);
}
```

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) ;

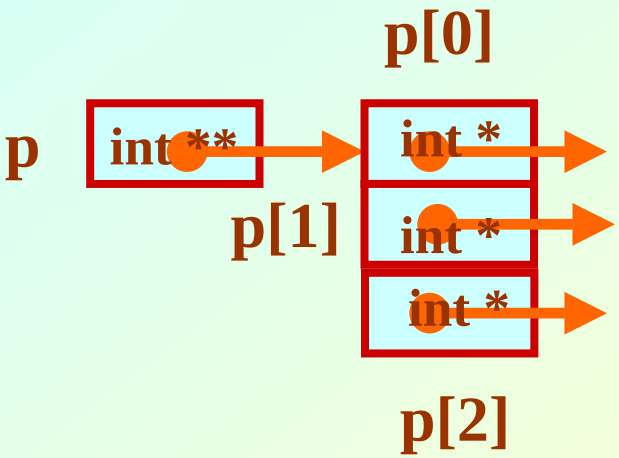
Contd.

- 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

```
#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

```
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");
     }
}
```

```
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
```

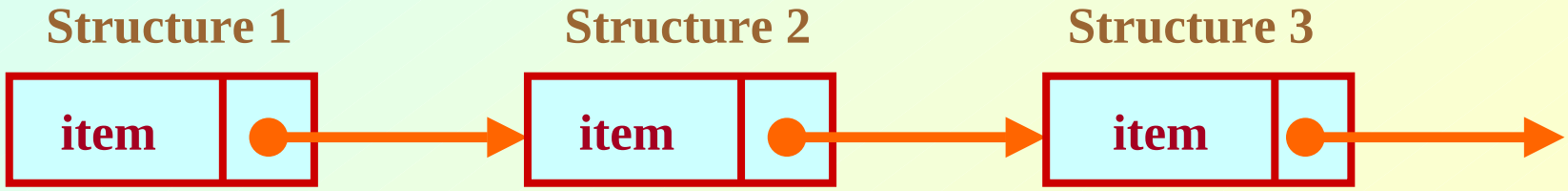
```
main()
{
  int **p;
  int M,N;

  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);
```

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.**



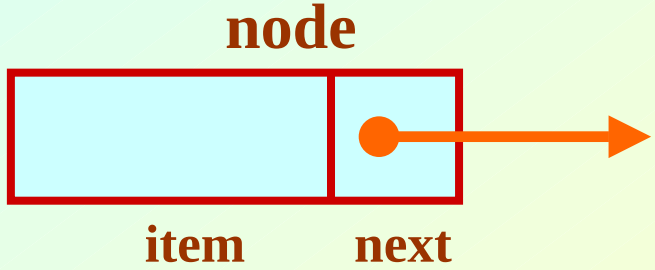
Contd.

- 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.**

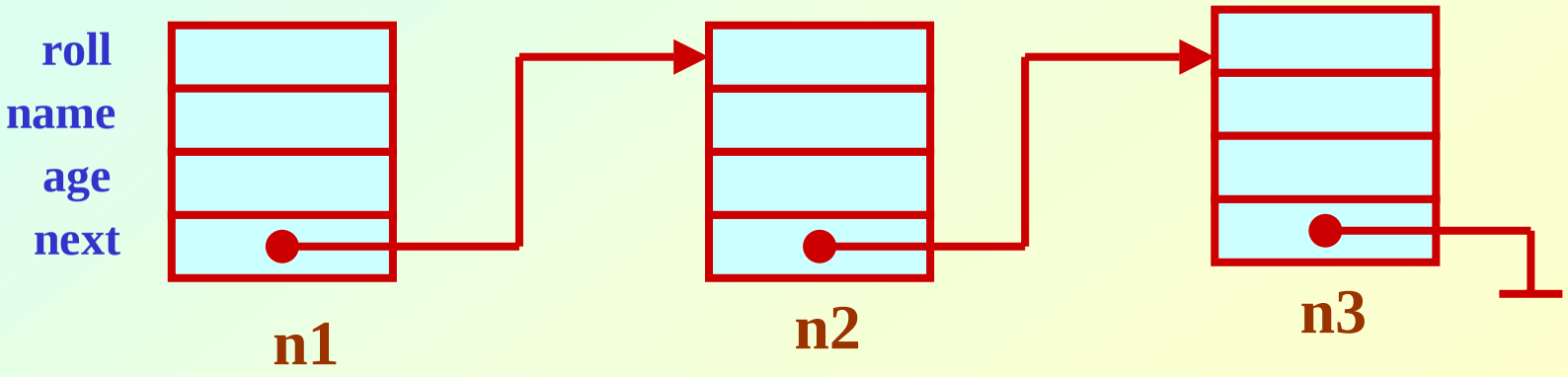
```
struct  stud  n1, n2, n3;
```

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;
   }
}
```