C Programming

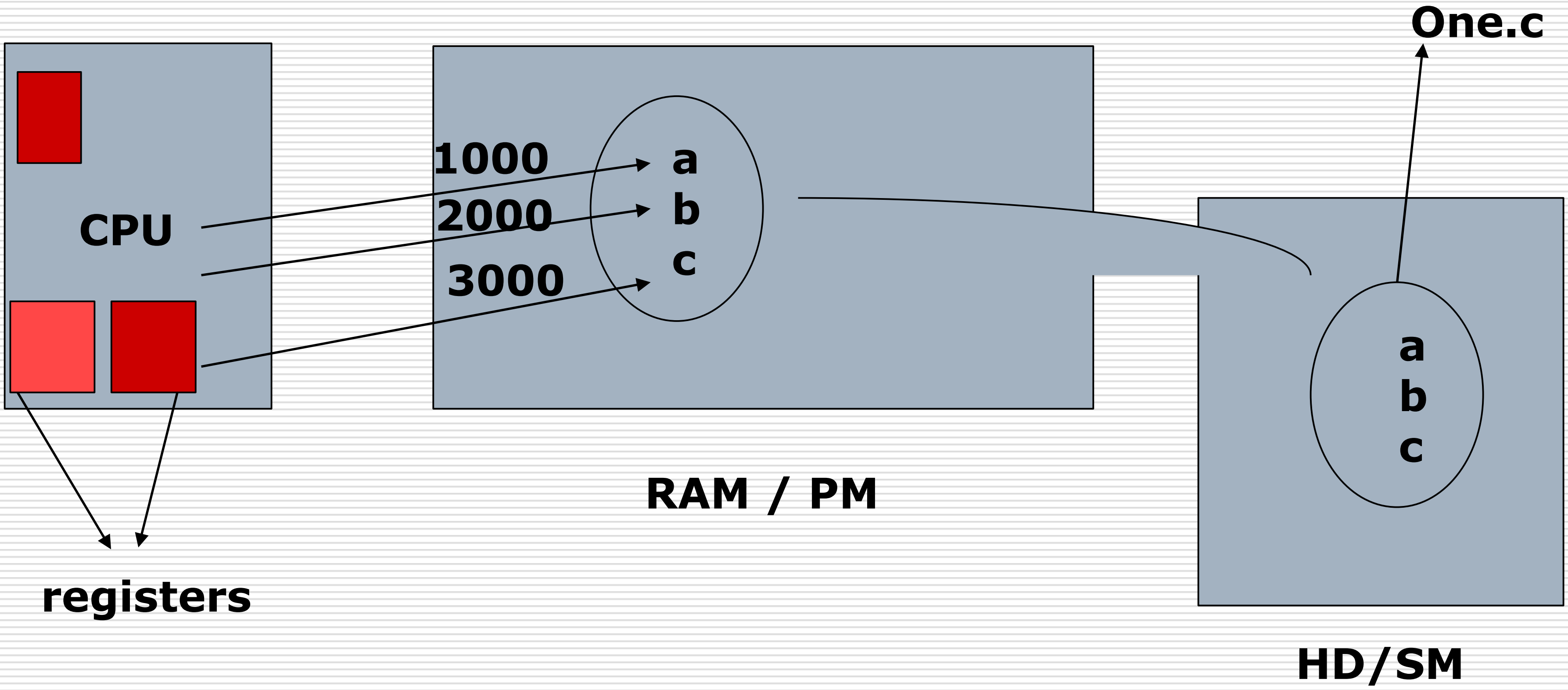
Ramaprasad H.C
9945426447
ramaprasadhc@10seconds.co.in

- a – 1000, b – 2000, c -3000

register int a, b, c; ✗

int a, b, c; ✗

register int c, b, a; SELECTED



int a;	register int b;	volatile int c;

□ 10 % 3 =

□ -10 % 3 =

□ 10 % -3 =

□ -10 % -3 =

□ 3 % 10 =

int a;
a = 5;
a++;
printf (“%d”, a);
	int a;
	a = 5;
	++a;
	printf (“%d”, a);

```
int a, b, c, d;
a = 5;
b = 6;
c = a++ + ++b;
d = ++a + b++;
```

int a = 5;

Printf (“%d%d%d%d%d%d”, a++, ++a, a--, --a, a++, ++a);

Comma operator

```
int a, b, c;
c = (a = 5, b = 6, a+b);
printf ("%d", c);
```

The if statement

- □ Syntax

```
if (this condition is true)
      execute one statement;

if (this condition is true)
{
      1st stmt;
      2nd stmt;
      3rd stmt;
      .
      .
      .
      Nth statement;
}
```

If one wants to execute more than one statement then braces are must.

Relational operators

□ ==

□ !=

□ <

□ >

□ <=

□ >=

□ 10 == 5 != 4 >= 5 < 9 > 10 <= 0

The if..else statement

Syntax:

```
if (this condition is true)
        execute one statement;
else
        execute one statement;
```

```
if (this condition is true)
{
        1st stmt;
        2nd stmt;
        .       More than one statement
        .
        Nth stmt;
}
else
{
        1st stmt;
        2nd stmt;
        .       More than one statement
        .
        Nth stmt;
}
```

Logical operators

- && Logical and
- || Logical or
- ! Logical Not

- !10&&5||0
- 0&&5||0
- 0||0
- 0

- 10==5 != 0 <= 4 > 45 < 90 >= 10
- 0!= 0 <= 4 > 45 < 90 >= 10

- 0 <=4 > 45 < 90 >= 10
- 1 > 45 < 90 ..=10
- 0< 90 >= 10
- 1 >= 10
- 0

Note:

```
# include <stdio.h>
void main ( )
{
        int i;

        printf (“Enter value for i \n”);
        scanf (“%d”, &i);
        if (i = 5)
                printf (“You entered 5
        \n”);
        else
                printf (“You entered
        something other than 5 \n”);
}
```

- □ In C language any non-zero value is considered as true and zero is considered as false.
- □ Another common mistake while using the if statement is to write a semicolon (;) after the condition.
- □ ; is considered as do nothing statement in C language.

```
If (this condition is true)
                execute one stmt;
else
                execute one stmt;
```

Note:

```
# include <stdio.h>
void main ( )
{
        int i;

        printf (“Enter value for i \n”);
        scanf (“%d”, &i);
        if (i = 5);
                printf (“You entered 5 
        \n”);
        else
                printf (“You entered 
        something other than 5 \n”);
}
```

- □ In C language any non-zero value is considered as true and zero is considered as false.
- □ Another common mistake while using the if statement is to write a semicolon (;) after the condition.
- □ ; is considered as do nothing statement in C language.

```
If (this condition is true)
                execute one stmt;
else
                execute one stmt;
```

Symbolic constants

- **Example 1: program without symbolic constants**

```
# include <stdio.h>
void main ( )
{
      float r, area;
      printf (“Enter radius \n”);
      scanf (“%f”, &r);
      area = 3.1412 * r * r;
      printf (“area of a circle = %f”, area);
}
```

Symbolic constants cont..

- **Example 2:**

```
# include <stdio.h>
# define PI 3.1412
void main ( )
{
    float r, area;
    printf (“Enter radius \n”);
    scanf (“%f”, &r);
    area = PI * r * r;
    printf (“area of a circle = %f”, area);
}
```

Keywords / Reserved words

- Extended keywords – specific to compiler –
- Different compilers?
- Turbo C – Microsoft
- Dev C++ -
- Gcc
- ANSI - _ _ asm, _ _ far, _ _ near;

Decision Control Problems

```
# include <stdio.h>
int main ( )
{
    int a = 300, b, c;
    if (a >= 400)
        b = 300;
    c = 200;
    printf (“%d%d”, b, c);
    return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int a = 500, b, c;
   if (a >= 400)
      b = 300;
   c = 200;
   printf (“%d%d”, b, c);
   return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int x = 10, y = 20;
   if ( x == y);
      printf (“%d%d”, x, y);
   return 0;
}
```

```
# include <stdio.h>
int main ( )
{
    int x = 3, y, z;
    y = x = 10;
    z = x < 10;
    printf (“x = %d y = %d z = %d”, x, y,
    z);
    return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int i = 65;
   char j = ‘A’;
   if (i == j)
      printf (“C is good”);
   else
      printf (“C is headache”);
   return 0;
}
```

```
# include <stdio.h>
void main ()
{
        if (condition)
              printf (“Hello”);
        else
              printf (“world”);
}
```

- **for statement**
- **Syntax**
-
  for (initialize counter; test counter; increment counter)
-
  execute one statement;
-
- **for (initialize counter; test counter; increment counter)**
- **{**
- **do this;**
- **and this;** **More than one statement.**
- **and this;**
- **}**

Note:

- is not necessary that a loop counter must only be an int. It can even be a float

Example:

```
# include <stdio.h>
void main ( )
{
    int i, j;
    for (i = 1; i <= 2; i++)
    {
        for (j = 1; j <= 2; j++)
        {
            if (i == j)
                continue;

            printf (“%d%d”, i, j);
        }
    }
}
```

Loop Control Problems

```
# include <stdio.h>
int main ( )
{
   int i = 1;
   while (i <= 10);
   {
      printf (“%d”, i);
      i++;
   }
   return 0;
}
```

```
# include <stdio.h>
int main  ( )
{
  int x = 4;
  while (x == 1)
  {
       x = x – 1;
       printf (“%d”, x);
       --x;
  }
  return 0;
}
```

```
# include <stdio.h>
int main ( )
{
    int x = 4, y, z;
    y = --x;
    z = x--;
    printf (“%d%d%d”, x, y, z);
    return 0;
X = 2
Y = 3
Z = 3
}
```

```
# include <stdio.h>
int main ( )
{
    int x = 4, y = 3, z;
    z = x-- - y;
    printf (“%d%d%d”, x, y, z);
    return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   while (‘a’ < ‘b’)
      printf (“Malayalam is a palindrome \n”);
   return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int i;
   while (i = 10)
   {
      printf (“%d”, i);
      i = i+1;
   }
   return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int x = 4, y = 0, z;
   while (x >= 0)
   {
          x--;
          y++;
          if (x == y)
                    continue;
          else
                    printf (“%d%d”, x, y);
   }
   return 0;
}
```

```
# include <stdio.h>
int main ( )
{
   int x = 4, y = 0, z;
   while (x >= 0)
   {
          if (x == y)
                    break;
          else
                    printf (“%d%d”, x, y);
          x--;
          y++;
   }
   return 0;
}
```

Case Control Instruction

```
# include <stdio.h>
int main ( )
{
  int k;
  float j = 2.0;
  switch (k = j + 1)
  {
       case 3:
              printf (“Trapped \n”);
              break;
       default:
              printf (“Caught \n”);
  }
  return 0;
}
```

```
# include <stdio.h>
int main ( )
{
  int ch = ‘a’ + ‘b’;
  switch (ch)
  {
       case ‘a’:
       case ‘b’:
              printf (“You entered b \n”);
       case ‘A’:
              printf (“a as in ashar”);
       case ‘b’ + ‘a’:
              printf (“You entered a and b”);
  }
  return 0;
}
```

```
# include <stdio.h>
int main ( )
{
  int i = 1;
  switch (i – 2)
  {
      case -1:
            printf (“Feeding fish \n”);
      case 0:
            printf (“Weeding grass \n”);
      case 1:
            printf (“Mending roof \n”);
      default:
            printf (“Just to survive \n”);
  }
  return 0;
}
```

Unit 4: Arrays & Datatypes

- Homogeneous group of elements are called arrays.
- **Declaration**
- int a[10];
- float b[20];

Array Initialization

- int num[5] = { 10, 20, 30, 40, 50};
- int n[ ] = {10, 20, 30};
- int a[3] = {10, 20, 30, 40};
- int a[4] = {10, 20};

Bound checking

- □ In C, there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array.
- □ This may lead to unpredictable results, and there will be no error message to warn you that you are going beyond the array size.
- □ In some cases computer may just hang.

Passing array elements to a function (Call by value)

```
# include <stdio.h>
void main ( )
{
        int i;
        void display (int);
        int marks[ ] = {10, 20, 30, 40, 50};
        for (i = 0; i < 5; i++)
                display (marks[i]);
}
void display (int m)
{
        printf (“%d”, m)
}
```

Passing array elements to a function (Call by reference)

```
# include <stdio.h>
void main ( )
{
        int i;
        void display (int *);
        int marks[ ] = {10, 20, 30, 40, 50};
        for (i = 0; i < 5; i++)
                display (&marks[i]);
}
void display (int *m)
{
        printf ("%d", *m)
}
```

Integers, long and short

- shorts are at least 2 bytes big.
- longs are at least 4 bytes big.
- shorts are never bigger than ints.
- ints are never bigger than longs.

Integers, long and short cont..

Compiler	Short	Int	Long
16 bit (Turbo C / C++)	2	2	4
32 bit (Visual C++)	2	4	4

Integers, long and short cont..

- Ex:
- short int a;
- short b;
- long int c;
- long d;

Integers, signed and unsigned

- unsigned int num_students;
- **Note:**
- With unsigned int, the range of permissible integer values (for 16-bit) will shift from -32767 to + 32767 to the range 0 to 65535.
- Thus declaring an integer as unsigned almost doubles the size of the largest possible value that it can otherwise take.
- This so happens because on declaring the integer as unsigned, the left-most is now free and is not used to store the sign of the number.
- Unsigned integer still occupies two bytes.

Chars, signed and unsigned

- Ordinary char has a range from -128 to +128.
- Unsigned char has a range from 0 to 255.

Floats, doubles and long doubles

- float – 4 bytes %f
- double – 8 byes %lf
- long double – 10 bytes %Lf
- **Note:**
- **The sizes and ranges of int, short and long are compiler dependent. The above are on 16-bit compiler.**

Data Type	Range	Bytes	Format
Signed char	-128 to + 127	1	%c
Unsigned char	0 to 255	1	%c
Short signed int	-32767 to +32767	2	%d
Short unsigned int	0 to 65535	2	%u
Signed int		4	%d
Unsigned int		4	%u
Long signed int		4	%ld
Long unsigned int		4	%lu
Float		4	%f
Double		8	%lf
Long double		10	%Lf

Unit 5: Functions & Pointers

- **C program to add 2 integer numbers without using functions**

```
# include <stdio.h>
void main ( )
{
      int a, b, res;
      printf (“Enter 2 numbers \n”);
      scanf (“%d%d”, &a, &b);
      res = a + b;
      printf (“Result = %d”, res);
      getch ( );
}
```

C program to add 2 integer numbers with the use of function

```
# include <stdio.h>


void main ( )
{
        int a, b, res;
        int add (int p, int q);  // Function declaration                    Calling Function
        printf ("Enter 2 numbers \n");
        scanf ("%d%d", &a, &b);
        res = add (a, b); // Function call or Function activation or Function invocation or
                                 Function instantiation.  In this function variable a and b
                                 are called actual parameters.
        printf ("Result = %d", res);
        getch ( );
}



int add (int p, int q)
{
        int sum;
                Called function.   In this function variables p and q are
        sum = p + q;               called formal parameters.
        return sum;
}
```

```
void praveen ( )
{
        int a, b, c;
        int suman (int,
int);
        a = 10;
        b = 20;
        c = suman (a, b);
        printf ("%d", c);
}
```

```
int suman (int p, int q)
{
        int r;
        r = p + q;
        return r;
}
```

C program to add two floating point numbers without using functions

```
# include <stdio.h>
void main ( )
{
    float p, q, res;
    printf (“Enter 2 floating point numbers \n”);
    scanf (“%f%f”, &p, &q);
    res = p + q;
    printf (“Result = %f”, res);
}
```

```
Void main ( )
{
      int a, b, c;
      int add (int, int);
      a = 10;
      b = 20;
     c =add (a, b);
      printf (“%d”, c);
}
```

```
int add (int p, int q)
{
      int r;
      r = p + q;
      retrun r;
}
```

C program to add two floating point numbers with the use of function

```
# include <stdio.h>
void main ( )
{
        float p, q, res;
        float sum (float, float);
        printf (“Enter 2 floating point numbers \n”);
        scanf (“%f%f”, &p, &q);
        res = sum (p, q);
        printf (“Result = %f”, res);
}
float sum (float p, float q)
{
        float res;

        res = p + q;
        return res;
}
```

Pointers

Ordinary Variable	Pointer Variable
int a;	int *a;
Meaning: a is an ordinary variable, which can hold numbers like 10, 20, 30 etc	Meaning: a is a pointer variable, which can hold the address of an integer variable.
float b;	float *b;
Meaning: b is an ordinary variable, which can hold numbers like 10.5, 24.5, 32.8	Meaning: b is a pointer variable, which can hold the address of a floating point variable

Definition:

- Pointer is a variable which can hold the address of another variable.

C program to swap contents of 2 variables without using pointers

```
□   # include <stdio.h>
□   void main ( )
□   {
□           int a, b;
□           void swap (int p, int q);
□           printf (“Enter 2 numbers
    \n”);
□           scnaf (“%d%d”, &a, &b);
□           printf (“Elements before
    swapping \n”);
□           printf (“a = %d \n b = %d”,
    a, b);
□           swap (a, b);
□           printf (“Elements after
    swapping \n”);
□           printf (“a = %d \n b = %d”,
    a, b);
□   }
```

```
□   void swap (int p, int q)
□   {
□           int temp;
□
□           temp = p;
□           p = q;
□           q = temp;
□   }
```

C program to swap contents of 2 variables using pointers

```
# include <stdio.h>
void main ( )
{
        int a, b;
        void swap (int * , int *);
        printf (“Enter 2 numbers \n”);
        scnaf (“%d%d”, &a, &b);
        printf (“Elements before swapping \n”);
        printf (“a = %d \n b = %d”, a, b);
        swap (&a, &b);
        printf (“Elements after swapping \n”);
        printf (“a = %d \n b = %d”, a, b);
}
```

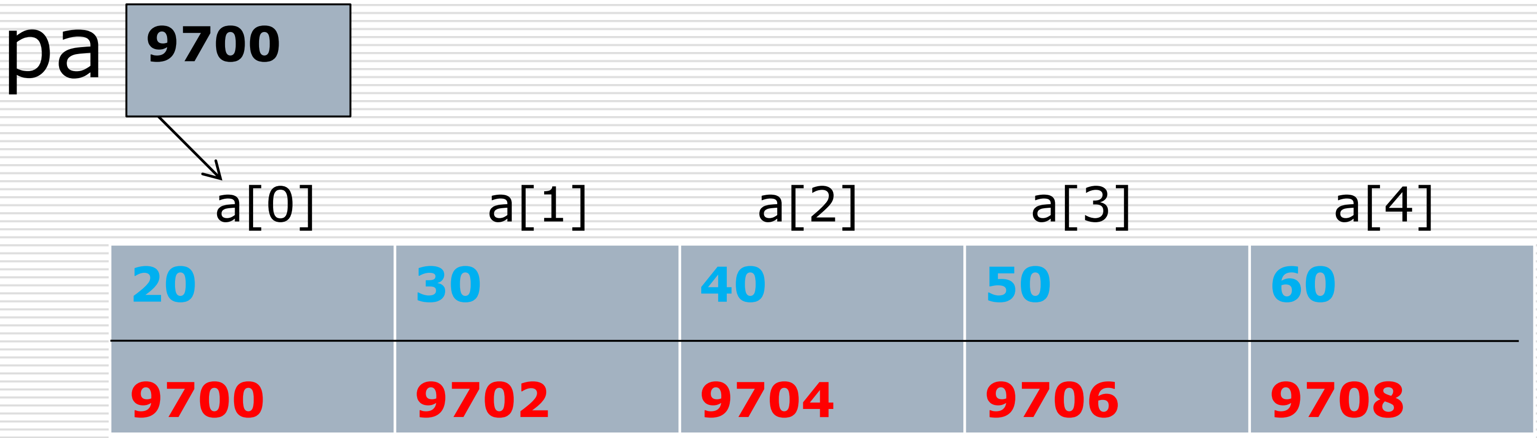
```
void swap (int *p, int *q)
{
        int temp;

        temp = *p;
        *p = *q;
        *q = temp;
}
```

Important points on array

- int num[]={24,34,12,44,56,17};
- By mentioning the name of the array, we get its base address.
- Thus by saying ***num**,we would refer to the zeroth element of the array(i.e 24 in this case).
- *num and *(num+0) both refers to 24.
- When we say num[i], the c compiler internally converts it to *(num+i).This means that all the **following notations are same**.
- num[i].....*(num+i)...*(i+num)...i[num]

Pointers and Arrays



pa 9700

a[0]	a[1]	a[2]	a[3]	a[4]
20	30	40	50	60
9700	9702	9704	9706	9708

The notation **a[i]** refers to the **ith** element of the array. If pa is a pointer to an integer declared as

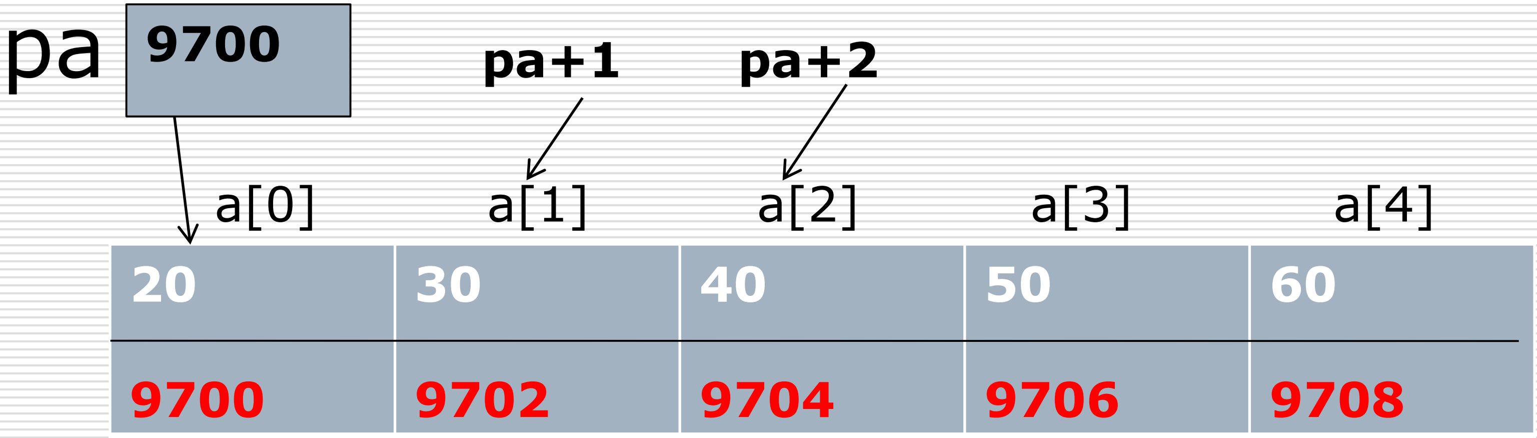
int *pa;

Then the assignment **pa=&a[0];** sets pa to point to the first element of a**.** **/* pa=a; */**

The assignment **x=*pa;** will copy the contents of a[0] into x.

10 SECONDS

Pointers and Arrays



pa 9700

pa+1

pa+2

a[0]	a[1]	a[2]	a[3]	a[4]
20	30	40	50	60
9700	9702	9704	9706	9708

***(pa+1) refers to contents of a[1].**
Note:
------

1) pa=&a[0];   (or pa=a;)
2) a[i] can also be written as *(a+i)-→ content
3) &a[i] can also be writen as (a+i).--→address
As such pa[i] and *(pa+i) are also identical.

10 SECONDS

Program which reads in array elements and finds the sum of the array elements using pointers.

```
#include <stdio.h>
int main()
{ int a[10],*pa,i,n=10,sum=0;
    pa=&a[0];
     for(i=0;i<10;i++)
        scanf(“%d”,pa+i);  /* or scanf(“%d”,&pa[i]); */
      for(i=0;i<10;i++)
        sum=sum + *(pa+i); /* or sum=sum + pa[i] */
    printf(“\n sum=%d”,sum);
}
```

Pointers and strings

char str[]=“Infosys”;

char *cptr=“Infosys”;

Both declarations are same.

The compiler allocates enough space to hold the string along with its terminating null character.

A pointer pointing to a string contains the base address of the character array and entire array or part of array can be accessed using this pointer.

Program to illustrate pointers and strings

```
#include <stdio.h>
int main()
{ char str1[]=“INFO TECH”;
 char str2[]=“Java and Internet”;
 char *cptr;
puts (str1); puts(str2);
cptr=str1;
puts(cptr);
cptr=cptr+5;
puts(cptr);
cptr=str2;
puts(cptr);
cptr=cptr+9;
puts(cptr);
}
```

Output:

```
INFO TECH
Java and Internet
INFO TECH
TECH
Java and Internet
Internet
```

Program:Length of string using pointers

```
#include <stdio.h>
int main()
{ char str[20]=“c++ and Java”;
 char *cptr;
int length=0;
cptr=str;cptr++)
for(;*cptr!=NULL;cptr++)
  length++;
Printf(“String
     length=%d”,length);
}
```

output

```
String length=12
```

Pointers and strings

NOTE 1

```
char str1[]=“hello”;
char str2[10];
char *s=“good”;
char *q;
```

We cannot assign a string to another

```
str2=str1;/* error */
```

Whereas ,we can assign a char pointer to another char pointer.

```
q=s; /* correct */
```

NOTE 2

```
char str1[]=“hello”;
char *p=“good”;
str1=“bye”;/*error*/
p=“bye”;/* correct */
```

Limitations of Array of Pointers and strings.

```
#include <stdio.h>
int main()
{char  *names[6];
int i;
for(i=0;i<6;i++)
{ printf("enter name\n");
   scanf("%s",names[i]);/* error */
}
return 0;}
```

solution to this problem dynamically allot memory

solution

```
#include <stdio.h>
#include  <string.h>
int main()
{char *names[6];
   char n[50];
   int len,i;
   char *p;
for(i=0;i<6;i++)
{printf(“Enter name\n”);
   scanf(“%s”,n);
   len=strlen(n);
p=(char *)malloc(len+1);
strcpy(p,n);
names[i]=p;
}
for(i=0;i<6;i++)
   printf(“%s\n”,names[i]);
return 0;
}
```

Pointers and multidimensional array

int arr[3][5];

A pointer variable **ptr** that can point to an element of arr(i.e, can point to a 5 element integer array) **is declared as:**

```
int  (*ptr)[5];
ptr=arr;
```

The parenthesis are required because the brackets have higher precedence than asterisk(*)

Program to demonstrate pointers and multi dimensional array

```
#include <stdio.h>
int main()
{ int arr[3][5]={{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
   int (*ptr)[5], i, j;
    ptr=arr;
 for(i=0;i<3;i++)
{   printf(“\n”);
    for(j=0;j<5;j++)
        printf(“%d  “,(*ptr)[j]);
    ptr++; /*moves pointer to first element of next row */
} }
```

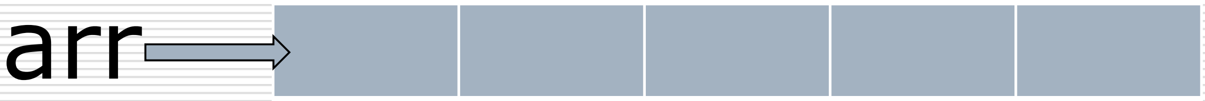
Diagramitic -2 dim array----- **int arr[3][5];**

		0	**1**	**2**	**3**	**4**
arr	**0**	**3501**	**3503**	**3505**	**3507**	**3509**
arr+1	**1**	3511	3513	3515	3517	3519
arr+2	**2**	3521	3523	3525	3527	3529

```
int (*ptr)[5];
ptr=arr;    /*ptr will have address 3501 */
ptr++; or ptr+1; -> now address 3511
ptr+2 ->   address will be 3521
```

Dynamic memory allocation- malloc()

```
int *arr;
arr=(int *)malloc (5*sizeof(int));
```



The header file **stdlib.h** contains memeory management functions malloc(),calloc(),realloc(),free()

program allocates a required block using malloc() displays addresses,reads and prints data.

```
#include <stdio.h>
#include<stdlib.h>
int main()
{ int *arr,n,i;
printf(“Enter no of elements\n”);
scanf(“%d”,&n);
arr=(int *)malloc(n*sizeof(int));
if(arr==NULL){
 printf(“could not allocate
     memory”);
 exit(1);
}
printf(“memory addresss \n”);
for(i=0;i<n;i++)
   printf(“%u”,arr+i);
printf(“Input data \n”);
for(i=0;i<n;i++)
   scanf(“%d”,arr+i);
printf(“Display data \n”);
for(i=0;i<n;i++)
   printf(“%d”,*(arr+i));

free(arr);
}
```

Arrays of pointers

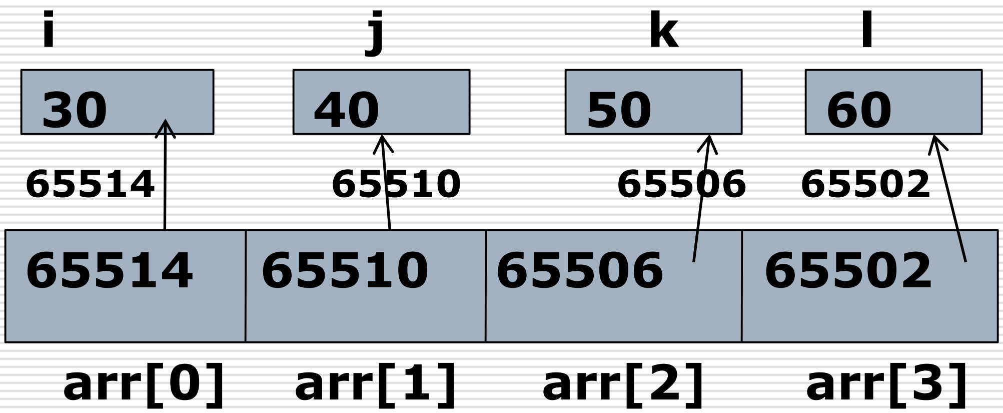
An array of pointers would be a collection of addresses. The addresses present in it can be addresses of individual variables or addresses of array elements.

int *arr[4]; /* array of integer pointers */

Assume we declare four variables,

int i=30,j=40,k=50,l=60;

then if we write arr[0]=&i;arr[1]=&j;arr[2]=&k;arr[3]=&l;



i j k l

30 40 50 60

65514 65510 65506 65502

65514 65510 65506 65502

arr[0] arr[1] arr[2] arr[3]

array of integer pointers… To access contents of i we write (*arr[0])

An array of pointers containing the addresses of other arrays.

```
#include <stdio.h>
int main()
{ int a[]={0,1,2,3,4};
  int *p[]={a,a+1,a+2,a+3,a+4};
printf("%u %u %d\n",p,*p,*(*p));
}
```

Output

????

contd

Prev output

Address of p

Address of a[0]

Content of a[0]…i.e 0

Note:

to print content of a[3],

((p+3)) or *(p[3])

2 dim array, using array of pointers-malloc()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int *arr[10];
   int m,n,i,j,sum=0;
 printf(“Enter number of rows nd columns\n”);
scanf(“%d %d”,&m,&n);
for(i=0;i<m;i++)
  arr[i]=(int *malloc(n*sizeof(int));
/* input */
for (i=0;i<m;i++)
  for(j=0;j<n;j++)
     scanf(“%d”,arr[i]+j);
```

```
/* display */
for (i=0;i<m;i++)
  for(j=0;j<n;j++)
    printf(“%d”,*(arr[i]+j));
}
```

Arrays of pointers to strings

```
#include <stdio.h>
int main()
{ char *names[]={

        “akshay”,”parag”,”raman”,”
    srinivas”,”gopal”,”rajesh”};
char *temp;
printf(“Original string: %s
    %s\n”,names[2],names[3]);
temp=names[2];
names[2]=names[3];
names[3]=temp;
printf(“New %s
    %s\n”,names[2],names[3]);
return 0;}
```

Output:

```
Original:raman srinivas
New:srinivas raman
```

In this program, the addresses (of the names }stored in array of pointers got exchanged.

Storage Classes in C

- If we don’t specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used.
- Thus, variables have certain default storage classes.

Storage Classes in C cont..

□ Locations in computer where value will be stored:

CPU registers and Memory.

Storage class tells following:

1. Where the variable would be stored.
2. What will be the initial value of the variable, if initial value is not specifically assigned. (i.e. default initial value)
3. What is the scope of the variable, i.e., in which functions the value of the variable would be available.
4. What is the life of the variable, i.e., how long would variable exist.

Storage Classes in C cont..

- There are four storage classes in C
  - Automatic storage class
  - Register storage class
  - Static storage class
  - External storage class

Automatic storage class

- Storage: Memory
- Default initial value: An unpredictable value, which is often called a garbage value.
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.

Automatic storage class cont..

```
# include <stdio.h>
int main ( )
{
    auto int i, j;  // int i, j;
    printf (“%d\n%d”, i, j);
    return 0;
}
```

Output:

Automatic storage class cont..

```
# include <stdio.h>
int main ( )
{
      auto int i = 1;
      {
            {
                        {
                                    printf (“%d”, i);
                        }
                        printf (“%d”, i);
            }
            printf (“%d”, i);
      }
      return 0;
}
```

Output:

Automatic storage class cont..

```
# include <stdio.h>
int main ( )
{
    auto int i = 1;
    {
        auto int i = 2;
        {
                auto int i = 3;
                printf (“%d”, i);
        }
        printf (“%d”, i);
    }
    printf (“%d”, i);
    return 0;
}
```

Automatic storage class cont..

Note:

In the above program, the compiler treats the three i’s as totally different variables, since they are defined in different blocks.

Register storage class

- Storage: CPU registers
- Default initial value: Garbage value
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.

Register storage class cont..

- Accessing register is always fast compared to accessing memory.

Ex:

```
# include <stdio.h>
int main ( )
{
    register int i;
    for (i = 1; i <= 10; i++)
        printf (“%d\n”, i);
    return 0;
}
```

Register storage class cont..

- Even though we have declared the storage class of i as register, we cannot say for sure that the value of i would be stored in CPU register.
- Because the number of CPU registers are limited, and they may be busy doing some other task.
- If registers are not available, the variable works as if its storage class is auto.

```
void main ()
{
        int a, b, c;
        int add (int, int);
        a = 10;
        b = 20;
        c = add (a, b);
        printf (“%d”, c);
}
```

```
int add (int p, int q)
{
        int r;
        r = p + q;
        return r;
}
```

Static Storage Class

- □ Storage: Memory
- □ Default Initial Value: Zero
- □ Scope: Local to the block in which the variable is defined
- □ Life: Value of the variable persists between different function calls.

Static Storage Class cont..

```
# include <stdio.h>
void increment ( );
void main ( )
{
        increment ( );
        increment ( );
        increment ( );
}
void increment ( )
{
        auto int i = 1;
        printf (“%d\n”, i);
        i = i + 1;
}
```

```
# include <stdio.h>
void increment ( );
void main ( )
{
        increment ( );
        increment ( );
        increment ( );
}
void increment ( )
{
        static int i = 1;
        printf (“%d\n”, i);
        i = i + 1;
}
```

a.C	**b.C**	**c.c**

int account;
Void main ()
{
account++;
}
int add ()
{
account--;
}
	extern int account;
	Void main ()
	{
	account++;
	}
		static int account;
		Void main ()
		{
		account++;
		}
		int fun1 ( )
		{
		account--;
		}

Static Storage Class cont..

- If the storage class is static, then the statement static int i = 1 is executed only once, irrespective of how many times the same function is called.

a.c	b.c	c.c

int account;
void main ( )
{
account ++;
}
int add ( )
{
account --;
}
	extern int account;
	Void main ()
	{
	account--;
	}
		Static int account;
		Void main ( )
		{
		
		account++;
		}
		Void add ()
		{
		account--;
		}

10 SECONDS

a.C	**b.C**	**c.c**

int account;
Void main ()
{
account ++;
}
Int add ()
{
account--;
}
	Extern int account;
	Void main ()
	{
	account--;
	}
		Static int account;
		Void main ()
		{
		account ++;
		}

External storage class

- Storage: Memory
- Default initial value: zero
- Scope: Global
- Life: As long as the program’s execution does not come to an end.

External storage class cont..

- External variables are declared outside all functions, yet are available to all functions.

External storage class cont..

```
# include <stdio.h>
int i;
void increment ( );
void decrement ();
int main ( )
{
     printf (“i = %d\n”, i);
     increment ( );
     increment ( );
     decrement ( );
     decrement ( );
     return 0;
}
```

```
void increment ( )
{
     int i = 5;
     i = i + 1;
     printf (“On incrementing
     i = %d\n”, i);
}
void decrement ( )
{
     static int i = 3;
     i = i – 1;
     printf (“On decrementing
     i = %d\n”, i);
```

External storage class cont..

```
# include <stdio.h>
int x = 21;
int main ( )
{
    extern int y;
    printf (“%d%d\n”, x, y);
    return 0;
}
int y = 31;
```

External storage class cont..

- □ extern int y;
- □ int y = 31;

1. In the above program fragment first statement is a declaration and second is the definition.
2. When we declare a variable no space is reserved for it, whereas, when we define it space gets reserved in memory.
3. We had to declare y since it is being used in printf ( ) before it is definition is encountered.

External storage class cont..

```
#include <stdio.h>
int x = 10;
void display ( );
int main ( )
{
      int x = 20;
      extern int x;
      printf (“%d\n”, x);
      display ( );
      return 0;
}
void display ( )
{
      printf (“%d\n”, x);
}
```

Structures

```
# include <stdio.h>
int main ( )
{
        struct book
        {
                char name;
                float price;
                int pages;
        };
        struct book b1, b2, b3;
        printf (“Enter names, prices and
        no of pages of 3 books\n”);
        scanf (“%c%f%d”, &b1.name,
        &b1.price, &b1.pages);
        scanf (“%c%f%d”, &b2.name,
        &b2.price, &b2.pages);
        scanf (“%c%f%d”, &b3.name,
        &b3.price, &b3.pages);
        printf (“This is what you entered
        \n”);
        printf (“%c%f%d”, b1.name,
        b1.price, b1.pages);
        printf (“%c%f%d”, b2.name,
        b2.price, b2.pages);
        printf (“%c%f%d”, b3.name,
        b3.price, b3.pages);
        return 0;
}
```

Structures cont..

```
struct book
{
    char name;
    float price;
    int pages;
};
struct book b1, b2,
    b3;
```

```
struct book
{
    char name;
    float price;
    int pages;
}b1, b2, b3;
```

Structures cont..

```
struct
{
   char name;
   float price
   int pages;
}b1, b2, b3;
```

Structures cont..

Initializing structures:

```
struct book
{
    char name[10];
    float price;
    int pages;
};
struct book b1 = {“Basic”, 130.0, 550};
struct book b2 = {“Physics”, 150.80, 800};
struct book b3 = {0};
```

Functions

```
# include <stdio.h>
int main ( )
{
   printf (“c to it that c survives \n”);
   main ( );
   return 0;
}
```

```
# include <stdio.h>
int i = 0;
void val ( );
int main ( )
{
   printf (“main’s i = %d\n”, i);
   i++;
   val ( );
   printf (“main’s i = %d\n”, i);
   val ( );
   return 0;
}
void val ( )
{
   i = 100;
   printf (“val’s i = %d\n”, i);
   i++;
}
```

```
# include <stdio.h>
int f (int);
int g (int);
int main ( )
{
  int x, y, s = 2;
  s*=3;
  y = f (s);
  x = g (s);
  printf (“%d%d%d”, s, y,
  x);
  return 0;
}
int t = 8;
```

```
int f (int a)
{
  a += -5;
  t -= 4;
  return (a + t);
}
int g (int a)
{
  a = 1;
  t += a;
  return (a + t);
}
```

```
# include <stdio.h>
int main ( )
{
   static int count = 5;
   printf (“count = %d\n”, count--);
   if (count != 0)
      main ( );
   return 0;
}
```

```
# include <stdio.h>
int g (int);
int main ( )
{
   int i, j;
   for (i = 1; i < 5;
   i++)
   {
      j = g( i );
      printf ("%d\n",
   j);
   }
   return 0;
}
```

```
int g (int x)
{
   static int v = 1;
   int b = 3;
   v += x;
   return (v + x +
   b);
}
```

```
# include <stdio.h>
int main ( )
{
   func ( );
   func ( );
   return 0;
}
void func ( )
{
   auto int i = 0;
   register int j = 0;
   static int k = 0;
i++, j++, k++;
printf (“%d%d%d”,
    i, j, k);
}
```

```
# include <stdio.h>
int x = 10;
int main ( )
{
    int x = 20;
    {
        int x = 30;
        printf (“%d”, x);
    }
    printf (“%d”, x);
    return 0;
}
```

Unions

- Union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type on another occasion

```
# include <stdio.h>
int main ( )
{
    union student;
    {
        int age;
        char grade;
        float per;
    };
    union student s1;
    printf (“Enter grade of student \n”);
    scanf (“%c”, &s1.grade);
    printf (“Grade is: %c”,s1. grade);
    printf (“Enter age of a student \n”);
    scanf (“%d”, &s1.age);
    printf (“Age is: %d”, s1.age);
    printf (“Enter percentage of student \n”);
    scanf (“%f”, &s1.per);
    printf (“Percentage is %f\n”, s1.per);
}
```

Unions cont..

In the above program, at any point of time, one can store age or grade or percentage. At a time one cannot store all the details. This is the major difference between structure and union.