Module Code: CSE401

Session 07: Pointers

Session Speaker:

Prema Monish
premamonish.tlll@msruas.ac.in

©Ramaiah University of Applied Sciences

Session Objective

- Pointer and pointer definition.
- Use of pointers in accessing single and double dimensional arrays.
- String handling using pointers.
- Pointers for functions
- Dynamic memory management

©Ramaiah University of Applied Sciences

Session Topic

- Introduction
- Address of a variables
- Initialization and declaration of pointer
- Accessing variable using pointer
- Pointers to functions
- Pointer to arrays
- Pointers to pointers
- Memory allocation techniques
- Dynamic memory allocation functions

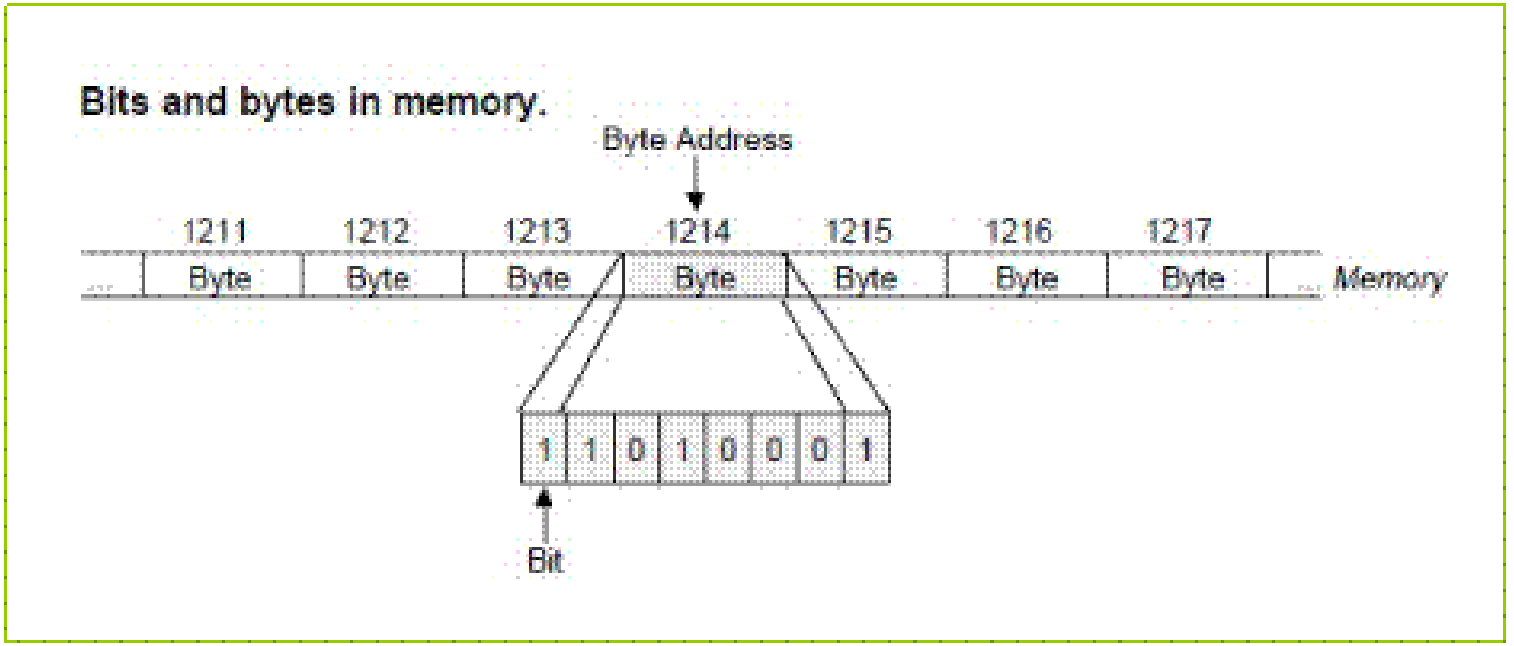
©Ramaiah University of Applied Sciences

Introduction

- RAM is the primary or main memory of computer. It is the place where the text, data, instructions and intermediate results are stored.
- The total memory is divided into number of bytes
- 1Byte = 8Bits
- These bits are actual places where the data is stored as 1's and 0's called binary data.
- Every byte is identified with a number called ***address.***
- It is always a positive number because the first byte address is 0 and the last byte address is depends on the size of memory.

©Ramaiah University of Applied Sciences

Introduction

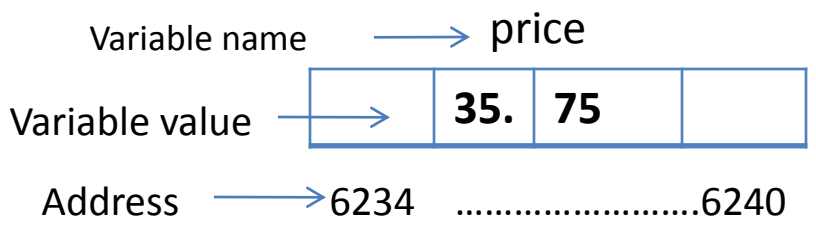


©Ramaiah University of Applied Sciences

Introduction

Any variable of any type has three common characteristics, which are

- Name of the variable
- Value of the variable
- The Address of variable
- A variable may take one or more bytes of memory but, the first byte number is considered as the address of variable.
- & Symbol called address operator used to access the address



E.g. float price=35.75;
price=35.75+10.0;
printf(" value of price %f",price);
printf("Address of price %d",&price);

Output :
Value of price :45.75
Address of price :6234

©Ramaiah University of Applied Sciences

Pointer

- Pointer is a variable which holds the address of another variable.
- While declaring the pointer makes sure that,The name of pointer must be prefixed with the symbol * (indirection operator)so that, the compiler restricts the variable only to store the address of another variable.
- The type of pointer must be same as the type of variable whose address will be stored.

Syntax

```
<Type>  *<Pointer Name>
```

©Ramaiah University of Applied Sciences

Pointer

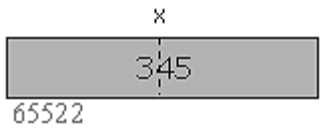
- * is the ***value at*** operator used to access the data indirectly through the address
- It is a unary operator
- It is also called indirection operator as it is used to access the data indirectly through the address
- Accessing the data indirectly using indirection operator is called ***indirect reference*** or ***dereferencing***

©Ramaiah University of Applied Sciences

Declaring pointers

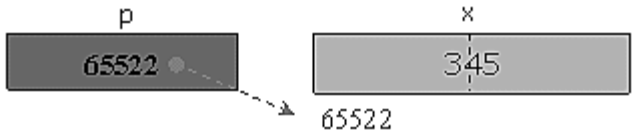
Steps to be followed while using pointers :

1. Declare a data variable E.x: int x=345;



2. Declare a pointer variable E.x: int *p;

3. Initialize a pointer variable E.x: p=&x;



4. Access data using pointer variable E.x: printf("\nx=%d",*p);

output :

x=345

©Ramaiah University of Applied Sciences

Accessing Variable


```
#include <stdio.h>
int main()
{
    int x=345,*p=&x;
    printf("x=%d",x);
    Printf("x=%d",*p);//dereferencing i.e.accessing variable value using
    *
    printf("\nx=%d\n",*(&x));
    printf("\nvalue of p=%d\n",p);
    printf("\naddress of x=%d\n",&x);
    return 0;
}
```

Output:
x=345
x=345
X=345
Value of p = 65522
Address of x = 65522

©Ramaiah University of Applied Sciences

Accessing Variable

```
#include <stdio.h>
int main()
{
     int *p;               /* declaration of pointer  */
     int x;                /* declaration of variable */
     p=&x;                   /* assigning the address to the pointer */
     *p=400;                 /* assigning value to x through dereference */
     printf("x=%d",*p);      /* printing the value of x through dereference */
     *p=*p+200;              /* incrementing the value of x through dereference */
     printf("\nx=%d\n",*p); /* printing the value of x through dereference */
     return 0;
}
```

Output:

x=400

x=600

©Ramaiah University of Applied Sciences

Multiple references to a common variable

Address of a variable can be stored in multiple pointers so that, any pointer can access the value of variable by dereferencing.

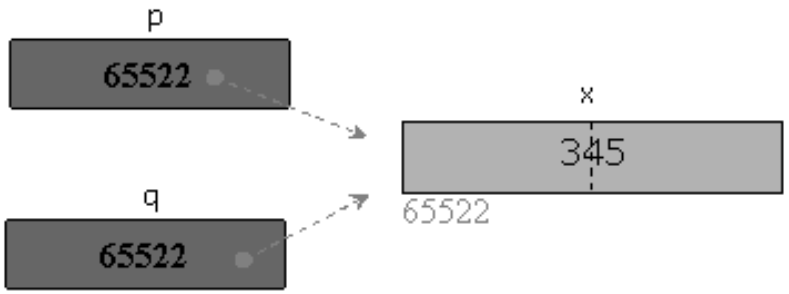
```
#include<stdio.h>
int main()
{
     int x=345;
     int *p,*q;
     /* p=q=&x; */
     p=&x;
     q=p;
     printf("Value of x %d",x);
     printf("\nValue of x %d",*p);
     printf("\nValue of x %d",*q);
     return 0;
}
```

Output:

Value of x 345

Value of x 345

Value of x 345



©Ramaiah University of Applied Sciences

Void pointer

- It is generally used as a generic pointer to store the address of any type of variable but, can't dereferencing
- We need to typecast void pointer type to variable pointer type before performing indirection or dereferencing.

```
#include<stdio.h>
int main()
{
     int x=45;
     float y=35.879656;
     char ch='a';
     void *p=&x,*q=&y,*r=&ch;
     printf("x=%d",*((int*)p));          /* type casting to int*   */
     printf("\ny=%f",*q);    /* error*/
     printf("\nz=%c",*((char*)r));     /* type casting to char* */
     return 0;
}
```

Output:

x=45

Error pointer1.c 9: Not an allowed type in function main

r = a

©Ramaiah University of Applied Sciences

Pointers and Function

- There are two ways of passing parameters to function:

1. Pass by value/Call by value (Refer session 5)
2. Pass by reference/Call by reference

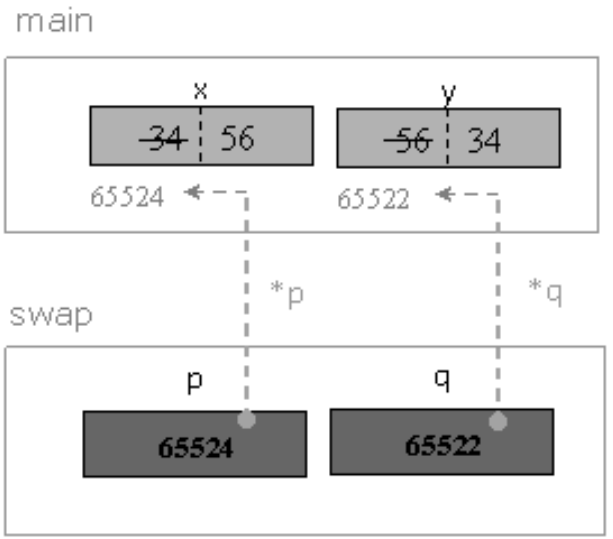
Pass by reference

- When function is called address of variables is sent to the calling function
- Send the addresses of actual arguments as arguments
- formal arguments would be pointers to the actual arguments.
- By dereferencing with formal arguments, we can access and modify the actual arguments.

©Ramaiah University of Applied Sciences

Pass by reference

```
#include<stdio.h>
void swap(int*,int*);
void main()
{
    int x,y;
    printf("Enter an integer into x:");
    scanf("%d",&x);
    printf("Enter an integer into y:");
    scanf("%d",&y);
    swap(&x,&y);
    printf("Value of x: %d",x);
    printf("\nValue of y: %d",y);
    return 0;
}
void swap(int *p,int *q)
{
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}
```



©Ramaiah University of Applied Sciences

Pass by reference

Advantages with pass by address

- It allows to change the actual arguments with a function. Some times it is needed to do so.
- It helps to return multiple values from a function, otherwise only possible to return a single value from a function

Disadvantages with pass by address

- Here actual arguments must be variables but, not constants and expressions because we can’t send the addresses of constants and expressions.
- Dereferencing through pointer is slower than accessing directly

©Ramaiah University of Applied Sciences

Pass by reference vs. Pass by value

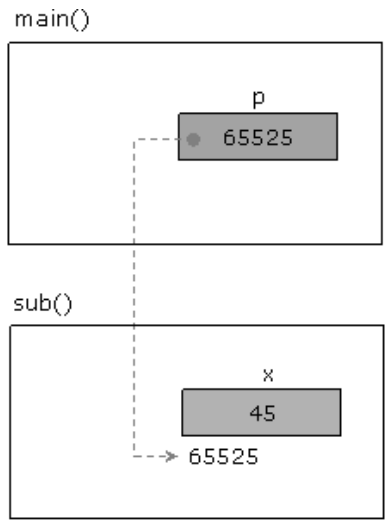
Pass by value	Pass by reference
When a function is called the values of variables are passed	When a function is called the **address of variable** are passed
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as pointers
Formal parameters in the function will not affect the actual parameters in calling function	The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters
Execution is slower since all the values have to be copied into formal parameters	Execution is faster since only address are copied
Multiple values cannot be returned	Multiple values can be returned using this

©Ramaiah University of Applied Sciences

Return by address

Return by value

```
int sub(int p,int q)
{
    int x;
    x=p-q;
    retun x;
}
```



Return by Reference

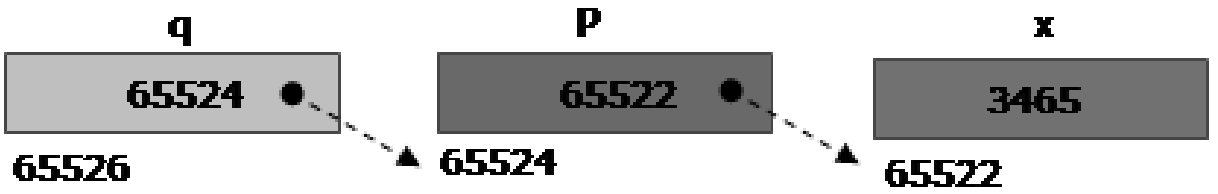
```
#include<stdio.h>
void *sub(int,int);
void main()
{
    int *p x,y;
    printf("Enter an integer into x:");
    scanf("%d",&x);
    printf("Enter an integer into y:");
    scanf("%d",&y);
    p=sub(x,y);
    printf("Value of x: %d",x);
    printf("\nValue of y: %d",y);
    return 0;
}
void *sub(int p,int q)
{
    int x;
    x=p-q;
    return &x;
}
```


©Ramaiah University of Applied Sciences

Pointer to pointer

- If a variable is used to store the address of a pointer then the variable is called pointer to pointer.
- ** is used as suffix to the pointer to pointer. We need to use *** as a suffix to declare a pointer to store the address of pointer to pointer.

Ex.:
```
int x=3465;
int *p;
p=&x;
int **q;
q=&p;
```

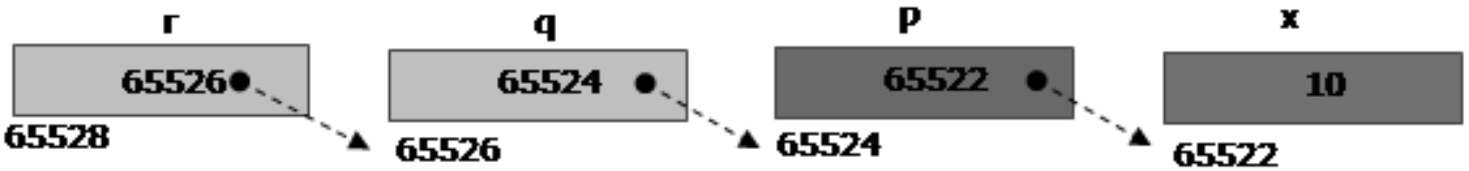


©Ramaiah University of Applied Sciences

Pointer to pointer

```
#include<stdio.h>
int main()
{
     int x=10,*p=&x,**q=&p,***r=&q;
     printf("%d",x);
     printf("\n%d",*p);
     printf("\n%d",**q);
     printf("\n%d",***r);
     return 0;
}
```

x	Value of x	10
&x	Address of x	65522
p		65522
*p	Value at 65522	10
&p	Address of p	65524
q		65524
*q	Value at 65524	65522
**q	Value at 65522	10

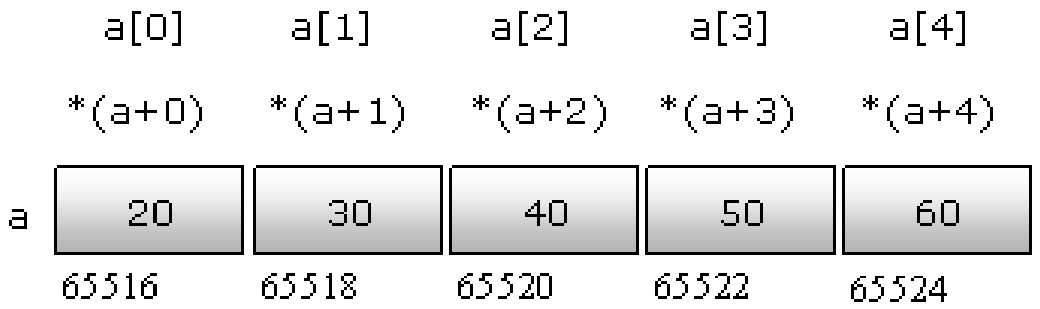


©Ramaiah University of Applied Sciences

Pointers to array

Dereferencing array: When an array is declared then the compiler allocates the set of elements in adjacent memory locations and the name of array gives the address of base element or we can say the address of array.

```
int a[5] ={20,30,40,50,60}
```



*(a+0)	value at 65516	20	a[0]
*(a+1)	value at 65518	30	a[1]
*(a+2)	value at 65520	40	a[2]
*(a+3)	value at 65522	50	a[3]

*(a+i) (Or) a[i]

©Ramaiah University of Applied Sciences

Pointers to arrays

Single Dimensional Array

```
#include<stdio.h>
int main()
{
    int a[]={34,56,67,22,34,76};
    int i;
    printf("Elements of array:\n");
    /*for(i=0;i<6;i++)
       printf("%d",a[i]);*/
    for(i=0;i<6;i++)
       printf("%d",*(a+i));   /* using *(a+i) in place of a[i]  */
    return 0;
}
```

Output:

Elements of array:

34 56 67 22 34 76

©Ramaiah University of Applied Sciences

Pointers to arrays

Multi-dimensional arrays

```
#include <stdio.h>
int main(void)
{
    char board[3][3] = {
    {'1','2','3'},
    {'4','5','6'},
    {'7','8','9'}};

printf("value of board[0][0] : %c\n", board[0][0]);
printf("value of *board[0] : %c\n", *board[0]);
printf("value of **board : %c\n", **board);
return 0;
}
```

Output:

value of board[0][0] : 1
value of *board[0] : 1
value of **board : 1

©Ramaiah University of Applied Sciences

Pointers to arrays

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

or

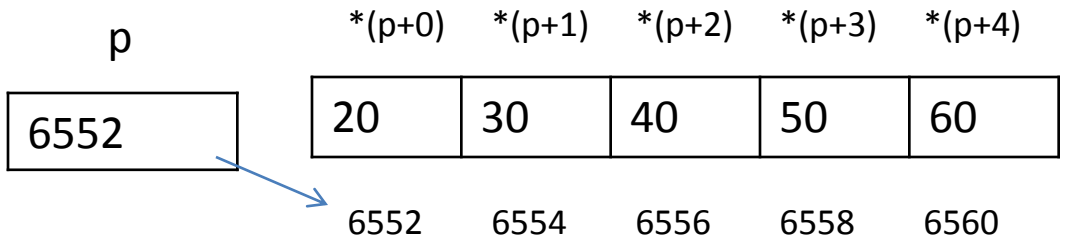
```
for(i=0;i<n;i++)
  scanf("%d",a+i);
```

We write &a[0] to read the data into the first element of array, the same thing can be written as a or (a+0).

©Ramaiah University of Applied Sciences

External pointer to an array

```
#include<stdio.h>
int main()
{
    int a[]={20,30,40,50,60},i;
    int *p=a;
    printf("Elements of array:\n");
    for(i=0;i<5;i++)
    {
     printf("%d",*p);
     p++;
    }
    return 0;
}
```



Output:
Elements of array:
20 30 40 50 60

©Ramaiah University of Applied Sciences

External pointer to an array

- Sending the address of an array as argument

```
#include<stdio.h>
void process(int[],int);
int main()
{
     int x[]={12,34,54,55,62,67};
     int i;
     process(x,6);
     printf("Elements of array:\n");
     for(i=0;i<6;i++)
       printf("%5d",x[i]);
     return 0;
}
void process(int *p,int n)
{
     int i;
     for(i=0;i<n;i++)
       *(p+i)=*(p+i)+10;
}
```

Output:
Elements of array:
22 44 64 65 72 77

©Ramaiah University of Applied Sciences

Return by address of an array

```
#include<stdio.h>
int* source();
int main()
{
    int *p,i;
    p=source();
    printf("Elements of array:\n");
    /* for(i=0;i<5;i++)
      printf("%5d",*(p+i)); */
    for(i=0;i<5;i++)
      printf("%5d",p[i]);
    return 0;
}
int* source()
{
    int x[]={12,34,56,66,77};
    return x;                    /* returning the address of array */
}
```

Output:

Elements of array:

12 34 56 66 77 88

©Ramaiah University of Applied Sciences

Memory allocation

- Memory can be reserved for the variables either during compilation or during execution time (run time)
- This gives rise to two different memory allocation techniques :

1. Static memory allocation (user declaration and definitions)
2. Dynamic memory allocation(using predefined functions)

©Ramaiah University of Applied Sciences

Static Memory allocation

- Memory is allocated for various variables during compilation time itself
- Once size of the memory allocated is fixed ,it can not be altered during execution time
- Allocated memory can not be expanded to accommodate more data or can not be reduced to accommodate less data

Disadvantage

- Memory allocated is fixed and cannot be altered during execution time
- Leads to under utilization
- Leads to overflow
- Imposes to know size of data before execution

©Ramaiah University of Applied Sciences

Dynamic memory allocation

- Allocated objects would be de-allocated only on completion of function execution in which objects are declared. We can't de-allocate the objects when we don't want them any more.
- It results wastage of memory because allocated memory can't be expanded or compressed.
- So Dynamic Memory Allocation i.e. Allocating and de-allocating memory during the execution of program (runtime). The functions used for dynamic memory allocation are:-

Data structures that grow and shrink during execution

1. malloc()
2. realloc()
3. calloc()
4. free()

©Ramaiah University of Applied Sciences

malloc()

- It is a function defined within the header file “stdlib.h”.
- Syntax:

  ptr=(data_type*)malloc(size);

- Where
  - ptr is a pointer variable of type data_type
  - data_type can be any basic data_type
  - size is the number of bytes

```
void *malloc(int size)
{
……
……
}
```

```
int*p;
p=(int*)malloc(20);
```

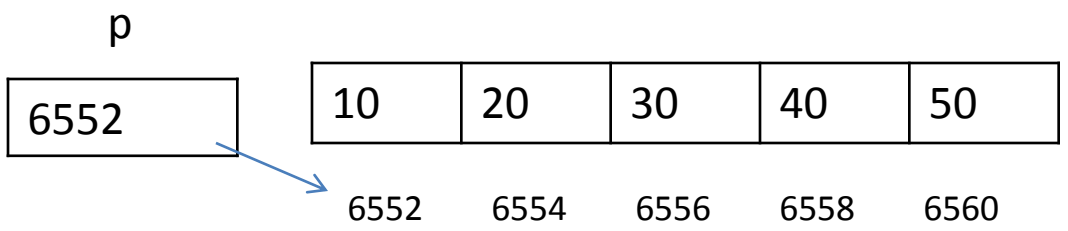
- It allocates 20bytes of memory in the heap and the address of first byte is returned as a void pointer.

©Ramaiah University of Applied Sciences

malloc()

- we can't perform dereferencing void pointer (address) returned by malloc() So, the allocated memory would be useless
- Hence we need to typecast the void pointer into other pointer type to perform address arithmetic and to make use of allocated memory.

```
int *p;
p=(int*)malloc(10);
```



- As the scale factor of int type is 2, adding 1 to the int pointer results increase in its value by 2. Hence p+0 is 6552, p+1 would be 6554 and p+4 would be 6558

©Ramaiah University of Applied Sciences

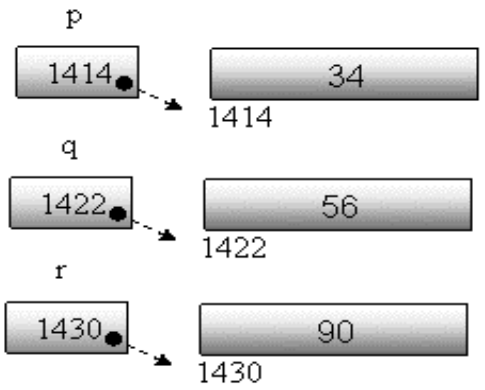
free()

- It is the function defined within the header file “stdlib.h”.
- It accepts the address of memory allocation allocated by malloc() and de-allocates the memory.
- The de-allocated memory would be used by some other source.
- Syntax : free(ptr);

```
int *p;
p=(int*)malloc(10); //allocates 10 bytes of memory
free(p);//de-allocates allocated memory
```

©Ramaiah University of Applied Sciences

```c
#include<stdio.h>
int main()
{
    /* declaration of pointers */
    int *p,*q,*r;
    /* allocating memory       */
    p=(int*)malloc(sizeof(int));
    q=(int*)malloc(sizeof(int));
    r=(int*)malloc(sizeof(int));
    printf("Enter two integers:\n");
    scanf("%d%d",p,q);   /* p,q holds the addresses of allocated memory */
    *r=*p+*q;
    printf("Sum of two numbers %d",*r);
    free(p);
    free(q);
    free(r); /* de-allocating memory */
    return 0;
}
```



Output:
Enter two integers:
34
56
Sum of two numbers 90

©Ramaiah University of Applied Sciences

calloc()

- It is the function defined in “stdlib.h”.
- On success returns the address of allocated memory as void pointer.
- Syntax: ptr=(data_type*)calloc(n,size);
- Where
  - ptr is a pointer variable of type data_type
  - data_type can be any basic data_type
  - n is the number of blocks to be allocated
  - size is the number of bytes in each block

©Ramaiah University of Applied Sciences

calloc()

Program to find maximum of n numbers using dynamic arrays

```
#include<stdio.h>
void main()
{
    int *p,i,j,n;
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    p=(int*)calloc(n,sizeof(int));
    printf("enter the elements\n");
    for(i=0;i<n;i++)
        scanf("%d",p[i]);
    j=0;
    for(i=1;i<n;i++)
    if(p[i]>p[j])j=i;
    printf("big =%d",p[j]);
    free(p);
}
```

©Ramaiah University of Applied Sciences

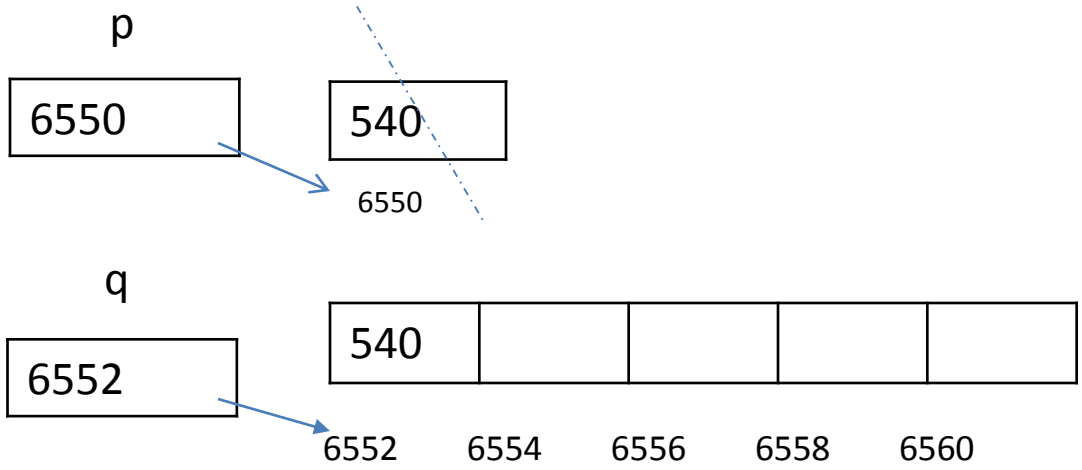
realloc()

- It is the function defined within the header file “stdlib.h”.
- It is used to expand or compress the memory allocation allocated by malloc().
- It allocates the new memory, copies the contents from existed memory to new memory and de-allocated the old memory
- It returns the address of newly allocated memory as void pointer.
- Syntax: ptr= (data_type*) realloc(ptr,size);
- Where
  - ptr is pointer variable to a block of previously allocated memory either malloc() or calloc()
  - size is new size of the block

©Ramaiah University of Applied Sciences

realloc()

```
int *p,*q;
p=(int*)malloc(2);
*p=540;
q=(int*)realloc(p,5);
free(q);
```



©Ramaiah University of Applied Sciences

Memory leakage

- A program which forgets to de-allocate dynamically allocated memory is said to have a “memory leak”.
- The memory leak gradually fills up the heap and there would be a time where program could not get free heap.
- It may result slow in running or some times system may crash.

©Ramaiah University of Applied Sciences

Possibilities of memory leak:

Case 1:

In the above function malloc() allocates the memory in the heap, p is the pointer to heap allocation. The scope of pointer ends on completion of function execution but, heap allocation would not be de-allocated which results memory leakage.

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

©Ramaiah University of Applied Sciences

Possibilities of memory leak:

Case 2:

- It may result memory leak when allocates for multiple times and points with the same pointer.
- The address returned from the second allocation overwrites the address of the first allocation. Consequently, the first allocation becomes a memory leak.

```
int *p;
p=(int*)malloc(10);
 p=(int*)malloc(20);
```

©Ramaiah University of Applied Sciences

Possibilities of memory leak:

Case 3:

- Some times copying pointers may leads to memory leakage.
- Both p and q points different memory locations, address of second allocation is copied into p.
- Now both p and q points the second allocation. Consequently, first allocation becomes a memory leak.

```
int *p,*q;
p=(int*)malloc(10);
q=(int*)malloc(10);
p=q
```

©Ramaiah University of Applied Sciences

Summary

- Pointer is variable which holds address of another variable
- Pointer to function
- Pointer to arrays
- Dynamic Memory Management
- Memory Leakage

©Ramaiah University of Applied Sciences