First Year First Semester Course

M.Tech. (CS) [Batch 2021-23]

Lecture #07

Introduction to Programming

Pointers in C

Course Instructor:

Dr. Monidipa Das

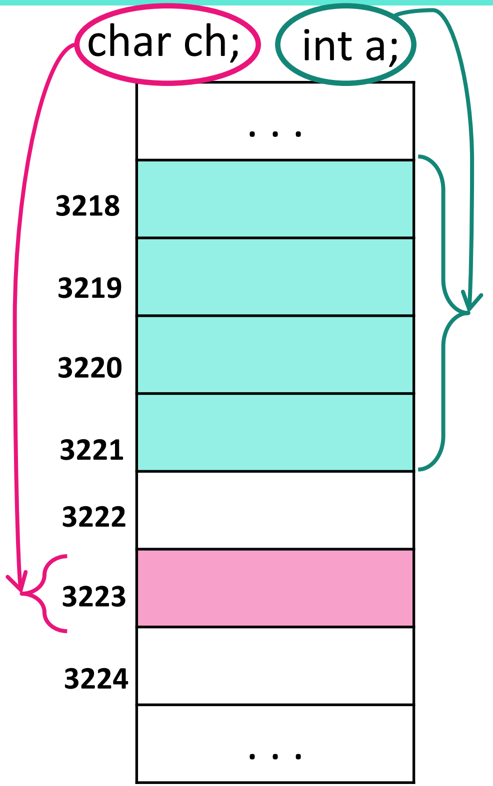
DST-INSPIRE Faculty

Machine Intelligence Unit (MIU), Centre for Artificial Intelligence and Machine Learning (CAIML)

Indian Statistical Institute (ISI) Kolkata, India

Introduction

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

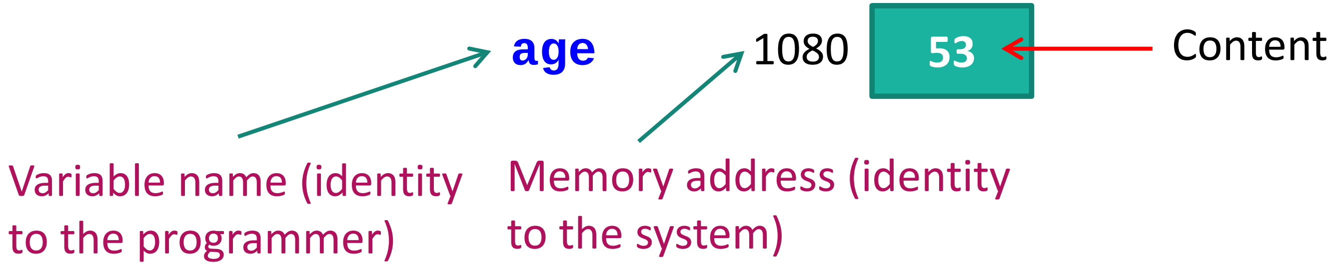


Example

- Consider the statement

```
int age = 53;
```

- This statement instructs the compiler to allocate a location for the integer variable *age*, and put the value 53 in that location.
- Suppose that the address location chosen is 1080.



Pointers

- Variables that hold memory addresses are called pointers.
- Since a pointer is a variable, its value is also stored in some memory location.

Variable	Value	Address
age	53	1080
ptr	1080	2152

1080 **53**
age

`ptr=&age;`

2152 **1080**
ptr

<u>**NOTE**</u>

Pointer constants→ Memory Addresses; We cannot change these;

Pointer value→ Value of the memory address of a variable;

Pointer variable→ Variable that contains a pointer value.

Example


```
#include <stdio.h>
int main()
{
        char a='1';
        int b=1;
        long int c=1;
        float d=1.0;
        double e=1.0;

        printf("a: size is %dB, address is %X and content is %c\n", sizeof(a), &a,a);
        printf("b: size is %dB, address is %X and content is %d\n", sizeof(b), &b,b);
        printf("c: size is %dB, address is %X and content is %ld\n", sizeof(c), &c, c);
        printf("d: size is %dB, address is %X and content is %f\n", sizeof(d), &d, d);
        printf("e: size is %dB, address is %X and content is %lf\n", sizeof(e), &e, e);
        return 0;
}
```

Returns no. of bytes required for data type representation

Memory address in hexadecimal

Content

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:

  int age;

  ptr = &age; // the address of ***age*** is assigned to ***ptr***.

What is the data type of ptr?

Data Type

- Pointer must have a data type. That is the data type of the variable whose address will be stored.

```
int *p;              // p is the pointer to data of type int.
float *p1;           // p1 is the pointer to data of type float.
long int *p2;        // p2 is the pointer to data of type long int.
```

NOTE

int *ptr and int* ptr are the same. However the first one helps you to declare in one statement:

int *ptr, var1;

REMEMBER

int x;

float *a;

a=&x; // NOT ALLOWED

Declaration and Initialization of Pointer


```
int age;
int *ptr;       //declaration
ptr=&age;   //initialization
```

```
int *p;
```

Data Type   Pointer   Variable

- printf(“%d”,age); is equivalent to printf(“%d”,*ptr);
- So **age** and ***ptr** can be used for the same purpose.
- Declaration of a pointer variable can also be made in a single statement along with other normal variables: `int age, *ptr;`
- Declaration and initialization can be combined: `int *ptr=&age;`
- Pointers can be initialized with NULL and 0 (zero). However, no other constant value can be assigned to a pointer: `int *p= 2186; //wrong`

Dereferencing Pointers

- Dereferencing is an operation performed to access and manipulate data contained in the memory location.

- A pointer variable is said to be dereferenced when the unary operator *, in this case called the ***indirection operator***, is used like a prefix to the pointer variable or pointer expression.

- An operation performed on the dereferenced pointer directly affects the value of the variable it points to.

Example


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

        a = 10 * (c + 8);
        p = &c;
        b = 10 * (*p + 8);

        printf ("a=%d b=%d \n", a, b);
        return 0;
}
```

Equivalent

Example


```
#include<stdio.h>
int main() {
        int *iptr, var1, var2;
        iptr=&var1;
        *iptr=32;
        *iptr += 10;
        printf("variable var1 contains %d\n",var1);
        var2=*iptr;
        printf("variable var2 contains %d\n",var2);
        iptr=&var2;
        *iptr += 20;
        printf("variable var2 now contains %d\n",var2);
        return 0;
}
```

Example [contd.]

Output
variable var1 contains 42
variable var2 contains 42
variable var2 now contains 62

- Thus the two uses of * are to be noted.
  - int *p for declaring a pointer variable
  - *p=10 is for indirection to the value in the address pointed by the variable p.

- This power of pointers is often useful, where direct access via variables is not possible.

Example


```c
#include <stdio.h>
int main()
{
            int x, y;
            int *ptr;
            x = 50 ;
            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 ("%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);
            return 0;
}
```

Output:

```
50 is stored in location 2293436
50 is stored in location 2293436
50 is stored in location 2293436
2293436 is stored in location 2293428
50 is stored in location 2293432

Now x = 25
```

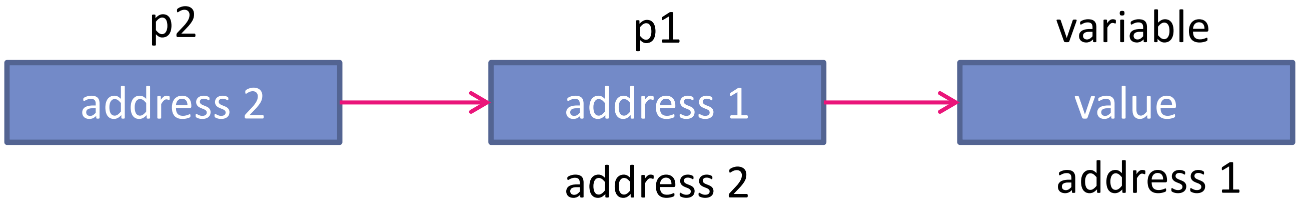
$*\&x \leftrightarrow x$

ptr=&x;

$\&x \leftrightarrow \&*ptr$

Chain of Pointers

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointers



- A variable which is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.

```
int **p2;
```

- The target value, indirectly pointed to by pointer to a pointer can be accessed by applying indirection operator twice

Example


```
#include <stdio.h>
int main()
{
        int x, *p1, **p2;
        x=400;
        p1=&x;
        p2=&p1;

        printf("%d %d %d", x,*p1,**p2);
        return 0;
}
```

Pointer Expressions

- Let p1, p2, p3 are properly declared and initialized pointers. Then, following statements are valid.

```
y=*p1 * *p2    //same as (*p1) * (*p2)
sum=sum + *p1;
Z= 6* - *p2/ *p1;  //same as (6* (-(*p2)))/(*p1)
```

- A few more valid expressions:

```
p2=p1 + 4;
p2= p1 – 2;
p3= p2 - p1;
p1++;
sum += *p3;
```

Pointer can also be used in relational expressions:

```
p1 > p2     //valid
p1 == p2   //valid
p1 != p2    //valid
```

Invalid Pointer Arithmetic

- Following are illegal:

  p1 / p2

  p1 * p2

  p1 / 3

  p1 + p2

Pointer Increment and Scale Factor

- When we increment a pointer, its value is incremented by the length of the data type that it points to.
- This length is called as the ***scale factor***
- Let p1 be an integer pointer and the initial value of p1 is 5140

  Then, p1 = p1 + 1 causes p1 to become 5144; **not 5141**

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***.

```
int x[5]={15,21,33,56,45};
```

```
x = &x[0] = 1000
```

Elements →	x[0]	x[1]	x[2]	x[3]	x[4]
Value →	**15**	**21**	**33**	**56**	**45**
Address →	1000	1004	1008	1012	1016

Base Address

Pointers and Arrays

- The elements of an array can be efficiently accessed by using a pointer.
- Consider an array of integers and an int pointer:

```
#define MAXSIZE 10
int A[MAXSIZE], *p;
```

- The ***following are legal assignments*** for the pointer p:

```
p = A; /* p point to the 0-th location of the array A */
p = &A[0]; /* p point to the 0-th location of the array A */
p = &A[1]; /* p point to the 1-st location of the array A */
p = &A[i]; /* p point to the i-th location of the array A */
```

- Whenever p is assigned the value &A[i], the value *p refers to the array element A[i].

Pointers and Arrays

- **Pointers can be incremented and decremented by integral values**.
- After the assignment p = &A[i]; the increment p++ (or ++p) lets p one element down the array, whereas the decrement p-- (or --p) lets p move by one element up the array. (Here "up" means one index less, and "down" means one index more.)
- Incrementing or decrementing p by an integer value n lets p move forward or backward in the array by n locations.

```
p = A; /* p point to the 0-th location of the array A */
p++; /* Now p points to the 1-st location of A */
p = p + 6; /* Now p points to the 8-th location of A */
p += 2; /* Now p points to the 10-th location of A */
--p; /* Now p points to the 9-th location of A */
p -= 5; /* Now p points to the 4-th location of A */
p -= 5; /* Now p points to the (-1)-th location of A */
```

Remember:

Increment/ Decrement is by data type not by bytes.

Example

- Consider the declaration:

```
int *p;
int x[5] = {10, 22, 34, 46, 58} ;
```

Suppose that the base address of x is 1500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	10	1500
x[1]	22	1504
x[2]	34	1508
x[3]	46	1512
x[4]	58	1516

Relationship between p and x:

```
p = &x[0]  (= 1500)     /* Equivalent to  p=x;
p+1 = &x[1] (= 1504)
p+2 = &x[2] (= 1508)
p+3 = &x[3] (= 1512)
p+4 = &x[4] (= 1516)
```

Accessing Array elements


```
#include<stdio.h>
int main()
{
        int iarray[5]={1,2,3,4,5};
        int i, *ptr;
        ptr=iarray;
        for(i=0;i<5;i++) {
                printf("iarray[%d] (%x): %d\n",i,ptr,*ptr);
                ptr++;
        }
        printf("=====================\n");
        for(i=0;i<5;i++) {
                printf("iarray[%d] (%x): %d\n",i, (iarray+i),*(iarray+i));
        }
        return 0;
}
```

Output

```
iarray[0] (22fea4): 1
iarray[1] (22fea8): 2
iarray[2] (22feac): 3
iarray[3] (22feb0): 4
iarray[4] (22feb4): 5
=================
iarray[0] (22fea4): 1
iarray[1] (22fea8): 2
iarray[2] (22feac): 3
iarray[3] (22feb0): 4
iarray[4] (22feb4): 5
```

NOTE : The name of the array can be used as a normal pointer, to access the other elements in the array.

More examples


```
#include<stdio.h>
int main()
{
        int i;
        int a[5]={1,2,3,4,5}, *p = a;
        for(i=0;i<5;i++,p++) {
                printf("%d %d",a[i],*(a+i));
                printf(" %d %d %d\n",*(i+a),i[a],*p);
        }
         return 0;
}
```

Output

1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5

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-pointers (or pass-by-pointers).

- 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: Swapping two numbers


```
void swap(int *a, int *b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

```
void swap(int a, int b)
{
        int temp = a;
        a = b;
        b = temp;
}
```

```
int main( )
{
        int i, j;
        scanf("%d %d", &i, &j);
        printf("Before swap: i=%d j=%d\n",i,j);
        swap(&i,&j);
        printf("After swap: i=%d j=%d",i,j);
}
```

Passing Arrays to a Function

- An array name can be used as an argument to a function.
  - Permits the entire array to be passed to the function.
  - Array name is passed as the parameter, which is effectively the address of the first element.

- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts.
  - The corresponding formal argument is written in the same manner.
    - Declared by writing the array name with a pair of empty brackets.
    - Dimension or required number of elements to be passed as a separate parameter.

Example: Function to find average


```
#include <stdio.h>
int 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));
            return 0;
}
```

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]

Dynamic Memory Allocation

Basic Idea

- Data may be 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.

Basic Idea

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

- 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.

malloc()

- 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) ;
```

malloc()

- **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.


```
cptr = (char *) malloc (10) ;
```

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

Example: malloc()


```
#include <stdio.h>
#include <stdlib.h>
int main()
{
            int i,N;
            float *height;
            float sum=0,avg;
            printf("Input the number of students: ");
            scanf("%d",&N);

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

            printf("Input heights for %d students \n", N);
            for(i=0;i<N;i++)
                        scanf("%f",&height[i]);
            for(i=0;i<N;i++)
                        sum+=height[i];
            avg=sum/(float) N;
            printf("Average height= %f \n", avg);
            return 0;
}
```

Output

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

Point 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.

calloc()

The C library function

```
void * calloc(nitems, size)
```

allocates the requested memory and returns a pointer to it.

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

calloc() vs. malloc()

- **malloc()** takes a single argument (memory required in bytes), while **calloc()** needs two arguments.
- **malloc()** does not initialize the memory allocated, while **calloc()** initializes the allocated memory to ZERO.
- **calloc()** allocates a memory area, the length will be the product of its parameters.

Example: calloc()


```
#include <stdio.h>
#include <stdlib.h>
int main () {
          int i, n, *pData;
          printf ("Amount of numbers to be entered: ");
          scanf ("%d",&n);

          pData = (int*) calloc (n,sizeof(int));

          if (pData==NULL)
                    exit (1);
          for (i=0;i<n;i++){
            printf ("Enter number #%d: ",i+1);
            scanf ("%d",&pData[i]);
          }
          printf ("You have entered: ");
          for (i=0;i<n;i++)
            printf ("%d ",pData[i]);

          return 0;
}
```

Output

Amount of numbers to be entered: 5
Enter number #1: 65
Enter number #2: 28
Enter number #3: 75
Enter number #4: 33
Enter number #5: 96
You have entered: 65 28 75 33 96

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() / calloc() / realloc().

Altering the Size of a Block

- Sometimes we need to alter the size of some previously allocated memory block.
  - More memory needed
- 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) ;
```

Altering the Size of a Block

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

Example: realloc()


```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
        int *pa, *pb, n; /* allocate an array of 10 int */
        pa = (int *)malloc(10 * sizeof *pa);
        if(pa) {
                printf("%u bytes allocated. Storing ints: ", 10*sizeof(int));
                for(n = 0; n < 10; ++n)
                        printf("%d ", pa[n] = n);
        }
        else{ printf("Memory is not allocated. \n"); exit(0);}

        pb = (int *)realloc(pa, 1000000 * sizeof *pb); // reallocate array to a larger size
        if(pb) {
                printf("\n%u bytes allocated, first 10 ints are: ", 1000000*sizeof(int));
                for(n = 0; n < 10; ++n)
                        printf("%d ", pb[n]); // show the array
                free(pa);   free(pb);     }
        else{ printf("Memory is not re-allocated. \n"); exit(0);}
        return 0;
}
```

Output:

40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9

4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9

Example: realloc()


```
int main(void) {
    int *p,i=0; char ch;
    printf("Enter an element: ");
    p = (int *)malloc(1 * sizeof(int));
    scanf("%d",p+i);
    printf("Would you like to add more items?: ");
    fflush(stdin); ch=getchar();
    while(ch=='y'||ch=='Y'){
        i++; p = (int *)realloc(p,(i+1)*sizeof(int));
        if(p) {
                printf("Enter the item: ");   scanf("%d",p+i);
                printf("Would you like to add more items?: ");
                fflush(stdin); ch=getchar();}
        else
                printf("Contiguaous memory space of required size is no longer available"); }
     printf("\nTotal allocated memory space size is: %u bytes\n",(i+1)*sizeof(p));
     printf("The elements are: \n");
     for(int j=0;j<=i;j++) printf("%d --> address: %x\n",p[j],p+j);
     return 0; }
```

```
Enter an element: 12
Would you like to add more items?: y
Enter the item: 24
Would you like to add more items?: y
Enter the item: 36
Would you like to add more items?: y
Enter the item: 4
Would you like to add more items?: y
Enter the item: 60
Would you like to add more items?: n
Total allocated memory space size is: 20 bytes
The elements are:
12 --> address: 880f18
24 --> address: 880f1c
36 --> address: 880f20
48 --> address: 880f24
60 --> address: 880f28
```

Questions?