Pointers in C

1

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.

2

Basic Concept

- In memory, every data item occupies one or more contiguous memory cells (bytes).
  - The number of bytes required to store a data item depends on its type (char, int, float, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) for the variable.
  - Since every byte in memory has a unique address, this location will also have its own (unique) address.

3

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

4

Contd.

- During execution, 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 *pointer* variable p.
  - p is said to point to the variable xyz.

Variable	Value	Address
xyz	50	1380
p	1380	2545

```
int xyz=50;
int *p;
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 a 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
2.500000 is stored in location 3221224904
12.360000 is stored in location 3221224900
12345.660000 is stored in location 3221224892
A is stored in location 3221224891
```


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:
  - The asterisk (*) tells that the variable pointer_name is a pointer variable.
  - pointer_name needs a memory location.
  - 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*.**

11

Remember …

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

```
float   x;
int    *p;
:
p = &x;
```

→ will result in erroneous output

12

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;
```

13

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

```
a=40 b=40
```

14

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);
}
```

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;
```

*p1 can appear on the left hand side

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, then 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***.

19

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.

20

- **Note:**
  - The exact scale factor may vary from one machine to another.
  - Can be found out using the sizeof function.
  - Syntax:

    `sizeof (data_type)`

Example: to find the scale factors

```
#include  <stdio.h>
main()
{
  printf ("No. of bytes occupied by int is %d \n",    sizeof(int));
  printf ("No. of bytes occupied by float is %d \n",  sizeof(float));
  printf ("No. of bytes occupied by double is %d \n", sizeof(double));
  printf ("No. 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
```

Pointers and Arrays

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.

Both `x` and `&x[0]` have the value `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));
}
```

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

  p = array;

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

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

27

Pointers with 2-D arrays

TO BE DISCUSSED LATER

28

Pointers and Structures

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
```

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
```


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.

33

- 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.
  - `ptr->roll` and `(*ptr).roll` mean the same thing.

34

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.

35

Example: complex number addition

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

main()
{
   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);
}
```

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

   t.re = x.re + y.re ;
   t.im = x.im + y.im ;

   return (t)  ;
}
```

36

Example: Alternative way using pointers

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

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

37

Dynamic Memory Allocation (of 1-D arrays)

38

Basic Idea

- Many a time we face situations where data are dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data items keeps changing during program execution.
- Such situations can be handled more easily and effectively using dynamic memory management techniques.

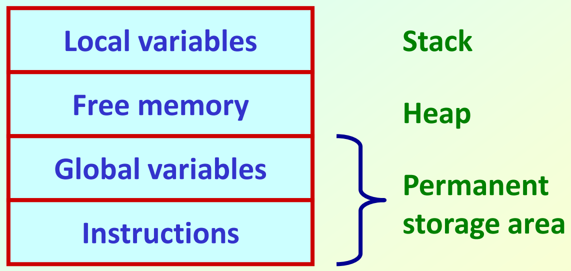
39

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.
  - Some compilers (e.g., C-99) may allow specifying a variable as size of array, but not all.
- Dynamic Memory Allocation
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

40

Memory Allocation Process in C



41

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.

42

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.

43

(a) malloc: 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 type-casted to any pointer type.
- General format:

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

44

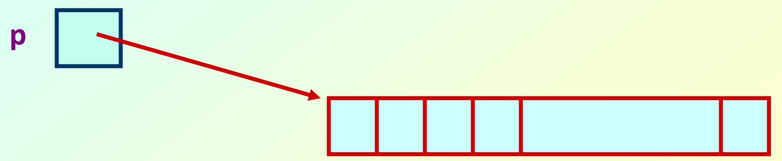
Contd.

`int *p;`

- 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`.



400 bytes of space

Contd.

`char *cptr;`

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

- Allocates 20 bytes of space for the pointer cptr of type char.

`struct stud *sptr;`

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

- Allocates space for a structure array of 10 elements. `sptr` points to a structure element of type "`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`.

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
  {
     printf ("\n Memory cannot be allocated");
     exit();
  }
```


(b) free: 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 syntax:

```
free (ptr);
```

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

Example 1: 1-D array of floats

```
#include <stdio.h>

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

  printf("Input no. of students\n");
  scanf("%d", &N);

  height = (float *)
       malloc(N * sizeof(float));
  if(height == NULL){
      printf("Cannot allocate mem");
      exit(1);
  }
```

```
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);
  free (height);
}
```


Example 2: 1-D array of structures

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

main()
{
  stud  *class, t;
  int  j, k, n;  float avg;

  scanf  ("%d", &n);
        /* no. of students */
class = (stud *)
    malloc(n * sizeof(stud));
```

```
  for (k=0; k<n; k++)
  {
    scanf ("%d %s %f", &class[k].roll,
               class[k].dept_code,
               &class[k].cgpa);
  }

  // compute average cgpa
  avg = 0.0;
  for(k=0;k<n;k++)
     avg += class[k].cgpa;

  avg = avg / (float) n;
  printf("Avg cgpa:%f",avg);
  free(class);
}
```

Some Points

- `class` is a pointer to the starting address of the allocated memory block.
- We can therefore access the individual elements of the structure array using array notation:
  - Example: `class[k]`, `class[k].roll`, etc.
- As an alternative, we can also access using *pointers* and the *arrow notation*:
  - Example: `(class+k)`, `(class+k)->roll`, etc.

51

(c) realloc: 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 as:

```
ptr = malloc (size);
```

then reallocation of space may be done as:

```
ptr = realloc (ptr, newsize);
```

52

Contd.

- The new memory block may or may not begin at the same place as the old block.
  - `realloc()` may create the new block in an entirely different region, and move the contents of the old block into the new block. The old block will be released using `free()`.
- The function guarantees that the old data remains intact.
- If `realloc()` is unable to allocate, it returns `NULL`. The original block is left untouched; it is NOT freed or moved.

53

(d) calloc: Allocate memory and initialize

- "calloc" is similar to "malloc" with two differences:
  - It takes two parameters instead of one.
  - It initializes the allocated memory to all zeros.
  - It also returns a pointer of type "void".
- General syntax:

```
ptr =  (type *) calloc (int n, int size);
```

"n" is the number of elements to be allocated, and "size" denotes the size of each element in bytes.

- Example:

```
int *p;
p = (int) calloc (20, sizeof(int));
```

54