1. Pointer

The simple variables are used to store the literal values. Whenever a variable is declared in a program it is having same memory location in the storage device. The memory location is called address of that variable. Hence there are some values which do not hold the literal values but stores the address (memory address) of any other variable. Such variables are called pointers.

Reasons for Using Pointers:
Pointers are used for many reasons like:
1) Pointers reduce the length and complexity of program.
2) They increase the processing speed.
3) They save the memory to a very large extent.
4) A pointer enables to access any variable whether it may be outside the function i.e. a direct control over a variable can be made using pointers.

1.1 Definition:
“A pointer is a variable which contains the address in memory of another variable”.

1.2 Declaration:
The pointer variables are declared by using the pointer notation * (asterisk). They are also associated with a data type which will represent the type of data to which that particular pointer is pointing.

Syntax:
Data type *<pointer name>;
The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Examples:
1) int *p; /* pointer to an integer */
2) double *d; /* pointer to a double */
3) float *fp; /* pointer to a float */
4) char *ch /* pointer to a character */
1.3 Initialization :
Initialization of pointer can be done using following 4 Steps:
1) Declare a pointer variable and note down the Data Type.
2) Declare another variable with same data type as that of pointer variable.
3) Initialize ordinary variable and assign some value to it.
4) Now initialize pointer by assigning the address of ordinary variable to pointer variable.

Syntax :
pointer = &variable;

Example :
int a = 5;
int *ptr;
ptr = &a;

About variable a :
1) Name of variable : a
2) Value of variable which it keeps: 5
3) Address where it has stored in memory : 1025 (assume)

About variable ptr :
1) Name of variable : ptr
2) Value of variable which it keeps: 1025
3) Address where it has stored in memory : 5000 (assume)

Pictorial representation :

1.4 Pointer to Pointer :
Pointers are themselves variables by nature and are stored in memory locations. A pointer variable can hold the address of another pointer variable. Visual conceptualization of pointer to pointer is given below :
A) Declaration:
A pointer to a pointer is declared as follows:
```c
int **ptr_to_ptr;
```
A pointer to a pointer allocates a single memory location, which holds an address of another pointer. A pointer to a pointer is used in functions where it is necessary to change the variable pointed by the pointer.

B) Example:
Let’s suppose have a pointer ‘p1’ that points to yet another pointer ‘p2’ that points to a character ‘c’. In memory, the three variables can be visualized as

```
1001
  ↓
  5000
  ↓
  8000
  ↓
'c'
```

In memory, pointer p1 holds the address of pointer p2. Pointer p2 holds the address of character ‘ch’.
So ‘p2’ is pointer to character ‘ch’, while ‘p1’ is pointer to ‘p2’ or can also say that ‘p2’ is a pointer to pointer to character ‘ch’.
Now, in code ‘p2’ can be declared as :
```c
char *p2 = &ch;
```
But ‘p1’ is declared as :
```c
char **p1 = &p2;
```
So that ‘p1’ is a double pointer (i.e. pointer to a pointer to a character) and hence the two *s in declaration.
Now,
1) ‘p1’ is the address of ‘p2’ i.e. 5000
2) ‘*p1’ is the value held by ‘p2’ i.e. 8000
3) ‘**p1’ is the value at 8000 i.e. ‘c’

1.5 Indirection Operator and Address of Operator:
Array subscript operator provides access to the elements in the internal array similarly indirection and address operators, member of pointer and pointer to member of
pointer operators provide pointer semantics for any object. Member of pointer and pointer to member of pointer operators return a pointer to the actual object which will be used for member access.

A) **Indirection Operator (*)** :
   The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses. If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

   **Conditions to Invalidate Pointer Value** :
   If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

   1) The pointer is a null pointer.
   2) The pointer specifies the address of a local item that is not visible at the time of the reference.
   3) The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
   4) The pointer specifies an address not used by the executing program.

B) **The Address-of Operator (&)** :
   The address-of operator (&) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the register storage-class specifier. The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

   The address of operator can only be applied to variables with fundamental, structure, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

C) **Examples** :
   ```c
   int *p, x;
   int a=20;
   double d;
   1) p = &a; // this statement uses the address-of operator
      The address-of operator (&) takes the address of a. The result is stored in the pointer variable p.
   2) x = *p;
   ```
The indirection operator (*) is used in this example to access the int value at the address stored in p. The value is assigned to the integer variable x.

3) if( x == *&x )
   printf( "True\n" );
This example prints the word True, demonstrating that the result of applying the indirection operator to the address of x is the same as x.

4) int roundup (void); /* Function declaration */
   int *proundup = roundup;
   int *pround = &roundup;
Once the function roundup is declared, two pointers to roundup are declared and initialised. The first pointer, proundup, is initialised using only the name of the function, while the second, pround, uses the address of operator in the initialisation. The initialisations are equivalent.

1.6 Pointer Arithmetic:
C pointer is an address which is a numeric value. Therefore, perform arithmetic operations on a pointer just as a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -. Multiplication and division operations are not allowed. C allows following arithmetic operations on pointers:

1) Increment and Decrement:
Incrementing or Decrementing pointer is generally used in array because contiguous memory in array and know the contents of next memory location. Incrementing or Decrementing pointer variable depends upon data type of the pointer variable. The increment (++) operator increases the value of a pointer by the size of the data object the pointer refers to. The decrement (--) operator decreases the value of a pointer by the size of the data object the pointer refers to. This differs from compiler to compiler as memory required to store integer vary compiler to compiler.

Formula:
(After incrementing)
new value = current address + i * sizeof(data type)
(After decrementing)
new value = current address - i * sizeof(data type)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Older Address stored in pointer</th>
<th>Next address stored in pointer after incrementing (ptr++)</th>
<th>Next address stored in pointer after decrementing (ptr--)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>1000</td>
<td>1002</td>
<td>998</td>
</tr>
<tr>
<td>Float</td>
<td>1000</td>
<td>1004</td>
<td>996</td>
</tr>
<tr>
<td>Char</td>
<td>1000</td>
<td>1001</td>
<td>999</td>
</tr>
</tbody>
</table>
Example:
```c
int *ptr1=(int *)1000;
int *ptr2=(int *)2000;
ptr1++;
printf( " Value of ptr1 : %u", ptr);
ptr2--;
printf( " Value of ptr2: %u", ptr);
```

**Output:**
Value of ptr1 : 1002
Value of ptr2 : 1098

2) **Addition and Subtraction:**
Integer can be added or subtracted from pointers.

**Formula:**
(for addition)
final value = (address) + (number * size of data type)
(for subtraction)
final value = (address) - (number * size of data type)

**Example:**
```c
int *p, a;
p = &a;
p = p + 5;
p = p - 3;
```
This code will increment `p` by 10 because `p` points to integer of size 2 bytes (i.e. 5*sizeof(int) i.e. 5*2=10) and similarly decrement `p` by 6.

**Note:**
two pointers cannot be added i.e. if *p1 and *p2 are two pointers then p1+p2 is not valid

3) **Differencing:**
Differencing means subtracting two pointers. Subtraction gives the total number of objects between them. Subtraction indicates “How apart the two Pointers are ”

**Formula** (while computing the difference between two pointers):
Final Result = (ptr2 – ptr1) / Size of data type

**Example:**
```c
int num , *ptr1 , *ptr2
ptr1 = &num ;
ptr2 = ptr1 + 2 ;
printf("%d", ptr2 – ptr1);
```
Pointers

Above example gives output 2 even though numerically ptr1 and ptr2 differ by 4 because, both point to the integer data type and difference between them is 2 objects.

4) **Comparison**:

Pointer comparison is valid only if the two pointers are pointing to same array. All Relational Operators can be used for comparing pointers of same type. All Equality and Inequality Operators can be used with all Pointer type. Pointers compared by using relational operators, such as ==, !, <, <=, >, and >=.  

**Example**:

```c
int *ptr1, *ptr2;
ptr1 = (int *)1000;
ptr2 = (int *)2000;
if(ptr2 > ptr1)
    printf("Ptr2 is far from ptr1");
```

### 2 Dynamic Memory Allocation:

The exact size of variables or array is unknown until the compile time, i.e. time when a compiler compiles code written in a programming language into an executable form. The size of array declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

#### 2.1 Meaning:

The process of allocating memory during program execution is called dynamic memory allocation. Dynamic memory is allocated on the "heap". The heap is the region of computer memory which is managed by the programmer using pointers to access the memory.

#### 2.2 Dynamic Memory Allocation Functions:

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Use of Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc()</td>
<td>Allocates requested size of bytes and returns a pointer first byte of allocated space</td>
</tr>
<tr>
<td>calloc()</td>
<td>Allocates space for an array elements, initializes to zero and then returns a pointer to memory</td>
</tr>
<tr>
<td>free()</td>
<td>Release the previously allocated space</td>
</tr>
</tbody>
</table>

1) malloc():
The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form. This malloc() function is used to allocate space in memory space in bytes for variables of any valid C data type during the execution of the program. malloc() does not initialize the memory allocated during execution. It carries garbage value. It returns null pointer if it couldn’t able to allocate requested amount of memory.

Syntax:
`pointer= (data type*)malloc(user_defined_size);`

Example:
`ptr=(int*)malloc(100*sizeof(int));`
This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

2) calloc() :
The name calloc stands for "contiguous allocation". It is used to allocate multiple blocks of memory dynamically during the execution (run-time) of the program. Calloc() function is also like malloc() function. The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax:
`pointer=(data type*)calloc(no of memory blocks, size of each block in bytes);`

Example:
`ptr=(float*)calloc(25,sizeof(float));`
This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e. 4 bytes.

3) realloc()
The realloc() function modifies the allocated memory size by malloc() and calloc() functions to new size. If enough space doesn’t exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block. The realloc function returns a pointer to the beginning of the block of memory. If the block of memory cannot be allocated, the realloc function will return a null pointer.

Syntax:
`pointer=realloc(pointer, size_t size);`

Example:
Pointers

ptr=malloc(10*sizeof(char));
This pointer ptr allocates space 10 bytes and the pointer points to the address of first byte of memory.
ptr=realloc(ptr,20*sizeof(char));
The pointer ptr can be allocated with new starting address of new memory block of 20 bytes other than previous address but at the same time.

4) free() :
Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space. free() function frees the allocated memory by malloc(), calloc(), realloc() functions and returns the memory to the system.
Syntax :
free(pointer);
Example :
Free(ptr);

2.3 Example: program for malloc(), calloc(), realloc() and free() :

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    /* or by calloc function 
    mem_allocation=(char*)calloc(20, sizeof(char)) */
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation,"Sharppublication.com");
    }
    printf("Dynamically allocated memory content : %s\n", mem_allocation);
    mem_allocation=realloc(mem_allocation,100*sizeof(char));
    if( mem_allocation == NULL )
```
Pointers

```c
{
printf("Couldn't able to allocate requested memory\n");
} 
else 
{
strcpy(mem_allocation, "space is extended up to 100 characters");
}
printf("Resized memory : %s\n", mem_allocation);
free(mem_allocation);
}
```

Output:
Dynamically allocated memory content : Sharppublication.com
Resized memory : space is extended up to 100 characters.

2.4 Advantages of Dynamic Memory Allocation over Static allocation:

There are some situations where use of dynamic memory allocation is advantageous over static memory allocation.

1) **Objects can be Deleted**:
In dynamic memory allocation, objects no longer in use can be deleted from memory to make room to create other objects. Thus it reuses the memory allocated to the objects.

2) **Objects Calculation at Run-time**:
At the time of programming, the number of objects to be created can be computed at run time only with the help of dynamic memory allocation.

3) **Automatic Allocation**:
In automatic dynamic allocation, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units.

4) **Program Controlled Allocation**:
It is also possible that different memory areas may be allocated to the same variable in different activations of a program unit, e.g. when some procedure is invoked in different blocks of a program. In program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution.

3 Functions and Pointers:
A pointer which keeps address of a function is known as function pointer. Just like pointer to characters, integers etc, we can have pointers to functions.

### 3.1 Pointers to Functions :

It is possible to declare a pointer pointing to a function which can then use as an argument in another function. A pointer to a function is declared as follows

1) **Pointer to Function** :

   **Syntax** :
   ```
   <return type of function> (*<name of pointer>) (function arguments)
   ```
   **Example** :
   ```
   int (*fptr) (int, int)
   ```
   The above line declares a function pointer 'fptr' that can point to a function whose return type is 'int' and takes two integers as arguments.

2) **Function Returning Pointer** :

   A function can return a pointer to the calling function.

   **Syntax** :
   ```
   <pointer data type> *<name of function>) (function arguments)
   ```
   **Example** :
   ```
   int *func (int a, float b);
   ```

**Example program for function pointer** :
```c
#include<stdio.h>
int func (int a, int b)
{
    printf ("\n a = %d\n", a);
    printf ("\n b = %d\n", b);
    return 0;
}
int main (void)
{
    int (*fptr) (int, int); // Function pointer
    fptr = func; // Assign address to function pointer
    func(2, 3);
    fptr (2, 3);
    return 0;
}
```

### 3.2 Pass a Pointer as an Argument to Function :


When, argument is passed using pointer, address of the memory location is passed instead of value.

Example:
/* C Program to swap two numbers using pointers and function. */
#include <stdio.h>

void swap (int *a, int *b);
int main ()
{
    int num1=5, num2=10;
    swap (&num1, &num2); /* address of num1 and num2 is passed to swap function */
    printf (“Number1 = %d
”, num1);
    printf (“Number2 = %d”, num2);
    return 0;
}

void swap (int *a, int *b)
/* pointer a and b points to address of num1 and num2 respectively */
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

Output:
Number1 = 10
Number2 = 5

4 Pointers and Arrays:
Arrays are closely related to pointers in C programming. Arrays and pointers are synonymous in terms of how they use to access memory. But, the important difference between them is that, a pointer variable can take different addresses as value whereas, in case of array it is fixed.

A) Relation between Arrays and Pointers:
Consider and array:
int arr[4];
In arrays of C programming, name of the array always points to the first element of an array. Here, address of first element of an array is &arr[0]. Also, arr represents the address of the pointer where it is pointing. Hence, &arr[0] is equivalent to arr. Also value inside the address &arr[0] and address arr are equal. Value in address &arr[0] is arr[0] and value in address arr is *arr. Hence, arr[0] is equivalent to *arr.

Similarly, 
&a[1] is equivalent to (a+1) AND, a[1] is equivalent to *(a+1).
&a[2] is equivalent to (a+2) AND, a[2] is equivalent to *(a+2).
&a[3] is equivalent to (a+1) AND, a[3] is equivalent to *(a+3).

&a[i] is equivalent to (a+i) AND, a[i] is equivalent to *(a+i).

In C, declare an array and can use pointer to alter the data of an array.

**B) Example Program for Array and Pointer :**

//Program to find the sum of six numbers with arrays and pointers
#include <stdio.h>
int main(){
    int i, class[6],sum=0;
    printf("Enter 6 numbers:\n");
    for(i=0;i<6;++i)
    {
        scanf("%d",(class+i)); // (class+i) is equivalent to &class[i]
        sum += *(class+i); // *(class+i) is equivalent to class[i]
    }
    printf("Sum=%d",sum);
    return 0;
}

**Output :**
Enter 6 numbers:
2
3
4
C) **Array of Pointers**

The following diagram visualizes an array of pointers.

![Array of Pointers Diagram](image)

An array of pointers is an array which consists of elements that either point to another element or another entire array itself. In an array of pointers, all the pointers need to point to the same kind of data. An array of pointers is declared as follows:

```c
int *array_ptrs[10];
```

In the above declaration, `array_ptrs` is an array of pointers to integers.

There is no way that there can be an array of pointers pointing to different types of elements. An array of pointers creates as many memory locations as specified by the size of the array.

---

Prepared by
- Prof. Ajay Gadhave
- Waghire College, Saswad