Pointers

- A pointer is a special variable that is capable of storing some address.
- It points to a memory location where the first byte is stored.

Declaring pointer variable

int *ptr

Syntax → data type *pointer name

Initialising pointer and assigning it.

```
int x = 5;
int *ptr;   ——→ declaring pointer variable
ptr = &x;   ——→ initialising its value
                to say it will store the
                address of variable x.
```

address operator

Address operator / address of operator / referencing operator

(&) ampersand

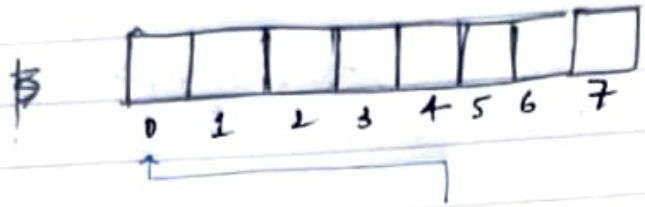
indicates the memory location of a particular element.

~~Def~~ Dereferencing / Value of operator / indirection operator

(*) star

Used to access the value stored at the location pointed by the pointer.

Pointer Arithmetic



p = &a[0] ⟶ this points to the address of 1st element in the array a

Addition

p = p + 3 :: p = &a[0+3]
= p = &a[3]

⟶ this points to the address of the 4th element in the array.

Subtraction

p = p - 3 = p = &a[3-3]
p = &a[0]

→ this points to the address of the 1st element in the array a.

* Arithmetic with pointers is only possible with arrays.

* Arithmetic with pointers which are pointing to different arrays is also not possible.

INCREMENT AND DECREMENT

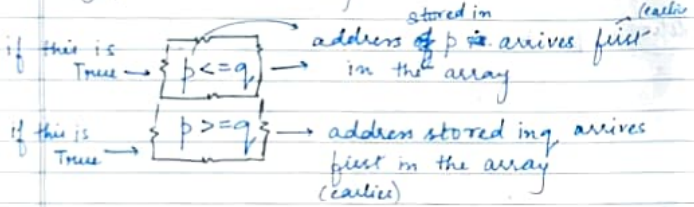
post increment

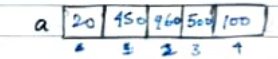
*(p++) ⟶ value of *p will be assigned first then it will increment the address.

++*p ⟶ increment the address and then the value will be assigned.

Comparing pointers

- We can use (>, <, <=, >=, ==, !=) operators to compare pointers.
- It is only possible to compare pointers when they point to same array.






```
p = &a[2]
q = &a[4]
```

(p <= q) : 1 (True)

while
↳ *p <= *q : 0 (False)
960 ≤= q

Pointer and Arrays

Name of an array can be used as a pointer. Name of an array represents the base address of the array. So, an array name is always treated as a pointer.

```
int a[5];
*a = 10;
```

dereference operator is being used with the name of the array.

Increment and decrement operations are not possible with array names

Array name represent the base address of an array. You can add or subtract to the array name to access the address element at the desired location but it is not possible to increment or decrement it because increment and decrement operations assign a new value (address) to the variable and you cannot change the base address of an array.

a + 1 (points to the second element)

a++ (INVALID)

Returning pointers

```
int *return_pointer (int a[]){
    return &a[2];
}
```
→ to give the address of the third element.

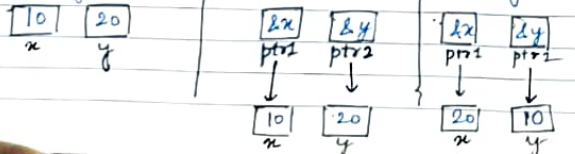
```
int main() {
    int a[] = {1, 2, 3, 4, 5};
    printf ("%d", *return_pointer(a));
    return 0;
}
```
→ using * operator to say an address will be returned

Output = 3

Passing pointers to functions.

```
int change (int *ptr1, int *ptr2) {
    *ptr1 = 20;
    *ptr2 = 10;
}

int main() {
    int x = 10, y = 20;
    change (&x, &y);
    printf ("x = %d, y = %d", x, y);
}
```

Initially



Finally

Fetching elements from array.

1) 1D arrays

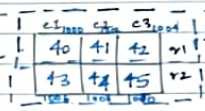
= a[5]

a represents 1000

*a = 40 *a+1 = 41 *a+2 = 42

a = 1000 a+1 = 1002 a+2 = 1004

2. 2D arrays = a[2][3] = a



a represents 1000 a+1 = 1006

→ *a = 1000 *a+1 = 1002 *a+2 = 1004

→ **a = 40 **a+1 = 41 **a+2 = 42

→ *(a+1) = 1006 *(a+1)+1 = 1008 *(a+1)+2 = 1010

→ **(a+1) = 43 *(*(a+1)+1) = 44 *(*(a+1)+2) = 45

3. 3D arrays a[2][3][3] =

a represents 1000 a+1 = 1018

↳ *a = 1000 *(a+1) = 1018

↳ **a = 1000 *(*(a+1)) = 1018

↳ *(**a) = 40 *(**a+1) = 49

↳ *(**a+1)+1) = 44 *(**(a+1)+1) = 53

↳ *(**a+2)+2 = 48 *(**(a+1)+2)+2 = 57

Strings

String constant is a sequence of characters enclosed within double quotes.

Example

"Hello everyone"

%s placeholder for strings

*Note Writing string is equivalent to writing the pointer to the first character of the string literal.

String literals cannot be altered because they have been allocated read only memory.

But character pointer itself has been allocated ~~red~~ read-write memory. So, the same pointer can point to some other string literal.

String constant Vs to Character constant.

- String contant is represented by a pointer to the first character. It is enclosed in " "
- Character constan is represented by an integer. It is enclosed in ' '.

Declaring A string variable

```
char name[20];
```

* Always make the array one character longer than the string. Extra one for the Null character.

Initialising a string variable.

```
char name[20] = "Hello";
```

String litral V/s char array

```
char *ptr = "Hello";
*ptr = 'M';
```

X This is not possible

```
char name[6] = "Hello";
name[0] = 'M';
```

Output-
Mello

✓ Correct

: Compiler adds \0 (Null character) to remaining places in case of short intialisation.

```
char name[7] = "Hello"
```

H	e	l	l	o	\0	\0

Writing string using Printf

```
char *ptr = "Hello World";
printf("%s", ptr)
```

output-

Hello World

Writing a part of the string

```
char *ptr = "Hello World";
printf("%.5s", ptr);
```

output-

Hello

Writing a string with defined field size.

```
char *ptr = "Hello";
printf("%6.5s", ptr);
```

output-

Hello

one extra empty space

Writing strings using puts

```
char *name = "Hello";
puts(name);
```

* if you write two printf statements, they'll be printed on the same line unless you use \n.

* if you write two puts statements, they'll be printed on consecutive lines.

How to read a string?

scanf function

```
char name[100];
printf("Your name: \n");
scanf("%s", name);
printf("%s", name);
```

Input-

My name is kiran.

Output-

My

* Do NOT use the gets function to read a string.

It is capable of overwriting into existing filled memory.

* Scanf doesn't read white spaces. it terminates the instructions when it encounters a white space.

gets() function

```
char name[25];
printf("Your name: \n");
```

input is kiranpatel

Kaue

Date:

Putchar()

Syntax → int putchar(int name)

accepts an integer argument (which repre
a character it wants to display) and returns
an integer representing the character.

String Library

<string.h>

1) strcpy() → String copy function

Syntax → char * strcpy(char* destination,
const char* source

returns the pointer to the first character of the string

Example →

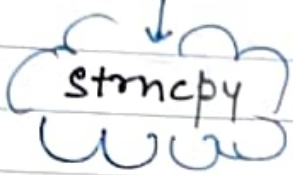
```
# include <stdio.h>
# include <string.h>

int main() {
    char str1[20] = "Hello";
    char str2[20];

    printf("%s\n", strcpy(str2, str1));
    printf("%s", str2);
    return 0;
}
```

Limitation of strcpy

strcpy function doesn't check that whether the destination is large enough to fit the contents of the source.

This leads to errors.

→ SOLUTION !



strncpy()

Syntax → strncpy(destination, source, sizeof(dest.));

Example I

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

int main() {
    char str1[6] = "Hello";
    char str2[4];
    strncpy(str2, str1, sizeof(str2));
    printf("%s", str2);
    return 0;
}
```

Output

Hell

strncpy does not add \0 (NULL) to the string if dest=<source

so, add \0 (NULL) in such cases.

strlen()

used to determine the length of the given string

Syntax

```
size_t strlen(const char *str);
```

↳ pointer to the string's 1st character whose length we want to find.

* It doesn't count the Null character.

Example ↴

```
#include <stdio.h>
#include <string.h>
int main() {
```

strcat()

appends the content of ~~strings~~ ~~test~~ 2nd argument string to the 1st argument string.

Syntax → char *strcat(char* str1, const char *str2);

Example

```
char str1, str2;
Strcpy(str1, "Welcome");
Strcpy(str2, " Home");
Strcat(str1, str2);
printf("%s", str1);
```

* strn includes the NULL character.

Output]
Welcome Home

If str1 isn't long enough to accomodate str2, it may give an error.

SOLUTION? → strncat()

Strncat() → appends the limited number of char. as specified in the third argument.

Syntax → strncat(str1, str2, size of string remaining str1)

Example]

```
char str1[5], str2[10];
strcpy(str1, "He");
Strcpy(str2, "llo");
Strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
printf("%s", str1);
```

↑ NULL char

OUTPUT

Strcmp() → string compare

Syntax → int strcmp(const char* s1, const char *s2);

returns value :
Less than 0, if s1 < s2
Greater than 0, if s1 > s2
Equal to 0, if s1 == s2

* To note :

- All upper case letters are less than all the lower case letters.
- Digits are less than letters
- Spaces are less than all printing characters.
- In case of two strings of same characters, the shorter one is less than the other.

Example

```
char *s1 = "abcd";
char *s2 = "abce";
if(strcmp(s1, s2) < 0){
printf("s1 is less than s2");}
else if(strcmp(s1, s2) > 0){
printf("s2 is less than s1");}
else {
printf("s1 is equal to s2");}
```

Output :
s1 is less than s2.

compare

1) abce vs bbce = abce < bbce

2) abcd vs abcd = 0 abcd = abc

3) abcd vs abc : 1 abcd > abc

4) Abcd vs abcd -1 Abcd < abc

6) I am Kiran vs Iamkiran 1 s1 > s2

7) I am kiran vs I am Kiran 1 s1 > s2

spaces are less than all printing characters but they have some value.

k > K

8) A bad Boy vs A naughty = -1 s1 < s2

↳ this is because]

1. both strings have equal length (9 chars)
2. s1 has 7 letters and s2 has 8 letters
3. s1 even has 2 upper case letters and s2 only has 1.
4. s1 has 2 spaces while s2 has only 1. and any letter > space.

9) Kiran 1 is (802) vs Kiran 0 is (833) = -1 s1 < s2

↳ this is because]

1. K < k and even though 1 > 0 the difference in letters is superior.

Array of String

An array of strings can be represented by an array of pointers.

Example :-

```
char *fruits[] = {"2 Oranges", "2 Apples", "3 Bananas", "1 Pineapple"};
```