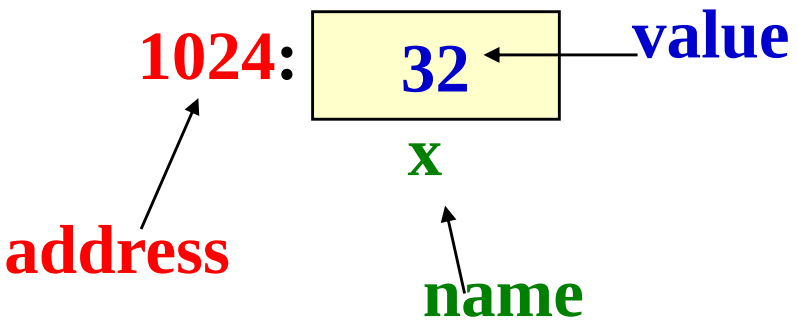
# What is a pointer?

- First of all, it is a variable, just like other variables you studied
  - So it has type, storage etc.
- Difference: it can only store the address (rather than the value) of a data item
- Type of a pointer variable – pointer to the type of the data whose address it will store
  - Example: int pointer, float pointer,…
  - Can be pointer to any user-defined types also like structure types

# Values vs Locations

- Variables name memory locations, which hold values



# 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** |

# Contd.

- During execution of the program, 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 variable p
  - p is said to point to the variable xyz

| Variable | Value | Address |
|---|---|---|
| xyz | 50 | 1380 |
| p | 1380 | 2545 |

p = &xyz;

*p=xyz (50)

# Pointers

- A pointer is just a C variable whose value can contain the address of another variable
- Needs to be declared before use just like any other variable
- 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

# Example

```
int       *count;
float   *speed;
char *c;
```

- 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

# Strings

# Strings

- 1-d arrays of type char
- By convention, a string in C is terminated by the end-of-string sentinel ‘\0’ (null character)
- char s[21] - can have variable length string delimited with \0
  - Max length of the string that can be stored is 20 as the size must include storage needed for the ‘\0’
- String constants : “hello”, “abc”
- “abc” is a character array of size 4

# Character Arrays and Strings

**char C[8] = { 'a', 'b', 'h', 'i', 'j', 'i', 't', '\0' };**

- C[0] gets the value 'a', C[1] the value 'b', and so on. The last (7th) location receives the null character '\0'
- Null-terminated (last character is '\0') character arrays are also called strings
- Strings can be initialized in an alternative way. The last declaration is equivalent to:

  char C[8] = "abhijit";

- The trailing null character is missing here. C automatically puts it at the end if you define it like this
- Note also that for individual characters, C uses single quotes, whereas for strings, it uses double quotes

# Reading strings: %s format

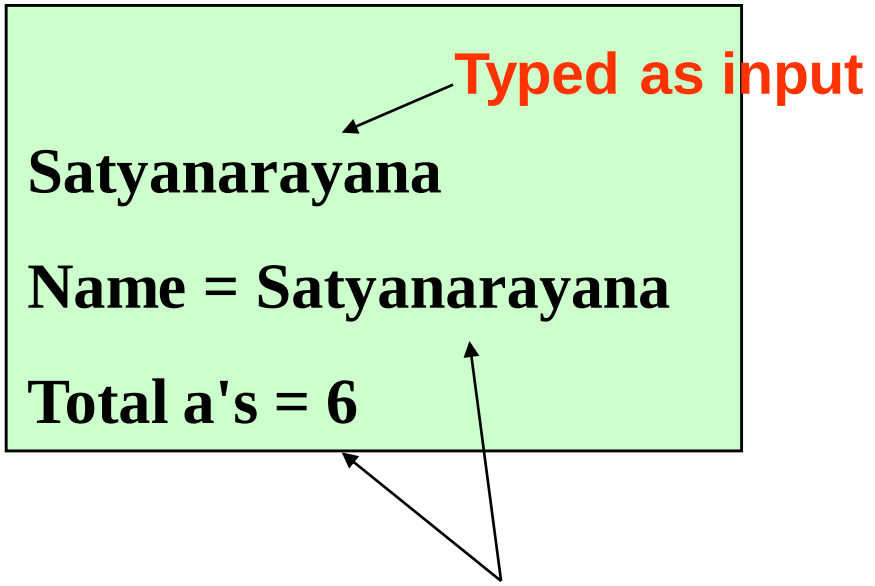
```
void main()
{
    char name[25];
    scanf("%s", name);
    printf("Name = %s \n", name);
}
```

**%s reads a string into a character array given the array name or start address. It ends the string with '\0'**

# An example

```
void main()
{
  #define SIZE 25
  int i, count=0;
  char name[SIZE];
  scanf("%s", name);
  printf("Name = %s \n", name);
  for (i=0; name[i]!='\0'; i++)
    if (name[i] == 'a') count++;
  printf("Total a's = %d\n", count);
}
```

Seen on screen

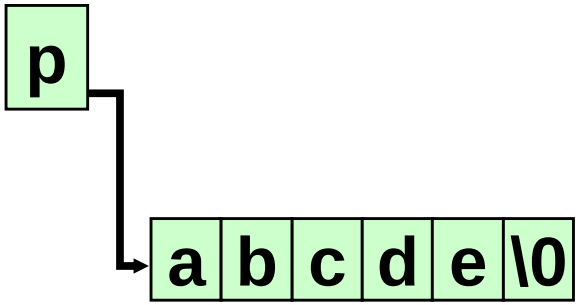



Note that character strings read in %s format end with '\0'

# Differences : array & pointers

char *p = “abcde”;

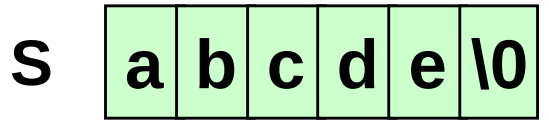
The compiler allocates space for p, puts the string constant “abcde” in memory somewhere else, initializes p with the base address of the string constant




char s[ ] = “abcde”;

≡ char s[ ] = **{‘a’,’b’,’c’,’d’,’e’.’\0’};**

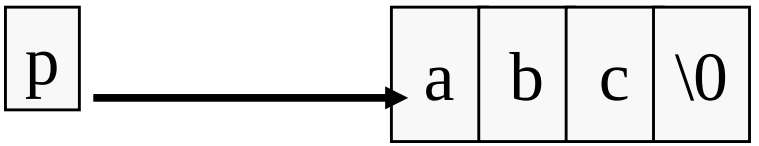
The compiler allocates 6 bytes of memory for the array s which are initialized with the 6 characters



# String Constant

- A string constant is treated as a pointer
- Its value is the base address of the string

```
char *p = “abc”;
```




```
printf (“%s %s\n”,p,p+1); /* abc bc is printed */
```

# Library Functions for String Handling

- You can write your own C code to do different operations on strings like finding the length of a string, copying one string to another, appending one string to the end of another etc.
- C library provides standard functions for these that you can call, so no need to write your own code
- To use them, you must do

  #include <string.h>

  At the beginning of your program (after #include <stdio.h>)

# String functions we will see

- strlen : finds the length of a string
- strcat : concatenates one string at the end of another
- strcmp : compares two strings lexicographically
- strcpy : copies one string to another

# strlen()

int strlen(const char *s)

- Takes a null-terminated strings (we routinely refer to the char pointer that points to a null-terminated char array as a string)
- Returns the length of the string, not counting the null (\0) character

**You cannot change contents of s in the function**

```
int strlen (const char *s)  {
      int n;
      for (n=0; *s!='\0'; ++s)
            ++n;
      return n;
}
```

# strcat()

- char *strcat (char *s1, const char *s2);
- Takes 2 strings as arguments, concatenates them, and puts the result in s1. Returns s1. Programmer must ensure that s1 points to enough space to hold the result.

**You cannot change contents of s2 in the function**

```
char *strcat(char *s1, const char *s2)
{
    char *p = s1;
    while (*p != '\0')   /* go to end */
        ++p;
    while(*s2 != '\0')
        *p++ = *s2++;  /* copy */
    *p = '\0';
    return s1;
}
```

# strcmp()

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

Two strings are passed as arguments. An integer is returned that is less than, equal to, or greater than 0, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

# strcmp()

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

Two strings are passed as arguments. An integer is returned that is less than, equal to, or greater than 0, depending on whether s1 is lexicographically less than, equal to, or greater than s2.

```
int strcmp(char *s1, const char *s2)
{
    for (;*s1!='\0'&&*s2!='\0'; s1++,s2++)
    {
        if (*s1>*s2) return 1;
        if (*s2>*s1) return -1;
     }
     if (*s1 != '\0') return 1;
     if (*s2 != '\0') return -1;
     return 0;
}
```

# strcpy()

char *strcpy (char *s1, char *s2);

The characters is the string s2 are copied into s1 until \0 is moved. Whatever exists in s1 is overwritten. It is assumed that s1 has enough space to hold the result. The pointer s1 is returned.

# strcpy()

char *strcpy (char *s1, const char *s2);

The characters is the string s2 are copied into s1 until ‘\0’ is moved. Whatever exists in s1 is overwritten. It is assumed that s1 has enough space to hold the result. The pointer s1 is returned.

```
char * strcpy (char *s1, const char *s2)
{
    char *p = s1;
    while (*p++ = *s2++) ;
    return s1;
}
```

# Example: Using string functions

```
int main()
{
char s1[ ] = "beautiful big sky country",
     s2[ ] = "how now brown cow";
printf("%d\n",strlen (s1));
printf("%d\n",strlen (s2+8));
printf("%d\n", strcmp(s1,s2));
printf("%s\n",s1+10);
strcpy(s1+10,s2+8);
strcat(s1,"s!");
printf("%s\n", s1);
return 0;
}
```

**Output**

```
25
9
-1
big sky country
beautiful brown cows!
```