UNIT-5

Introduction to Pointers

pointer constants,

pointer values,

pointer variables,

accessing variables through pointers,

pointer declaration and definition,

declaration versus redirection,

initialization of pointer variables,

Pointer for inter function communication,

pointer to pointers,

pointer to function.

Arrays and pointers

Pointer arithmetic and arrays,

array of pointers

Strings

Declaration, Initialization,

Input and Output functions,

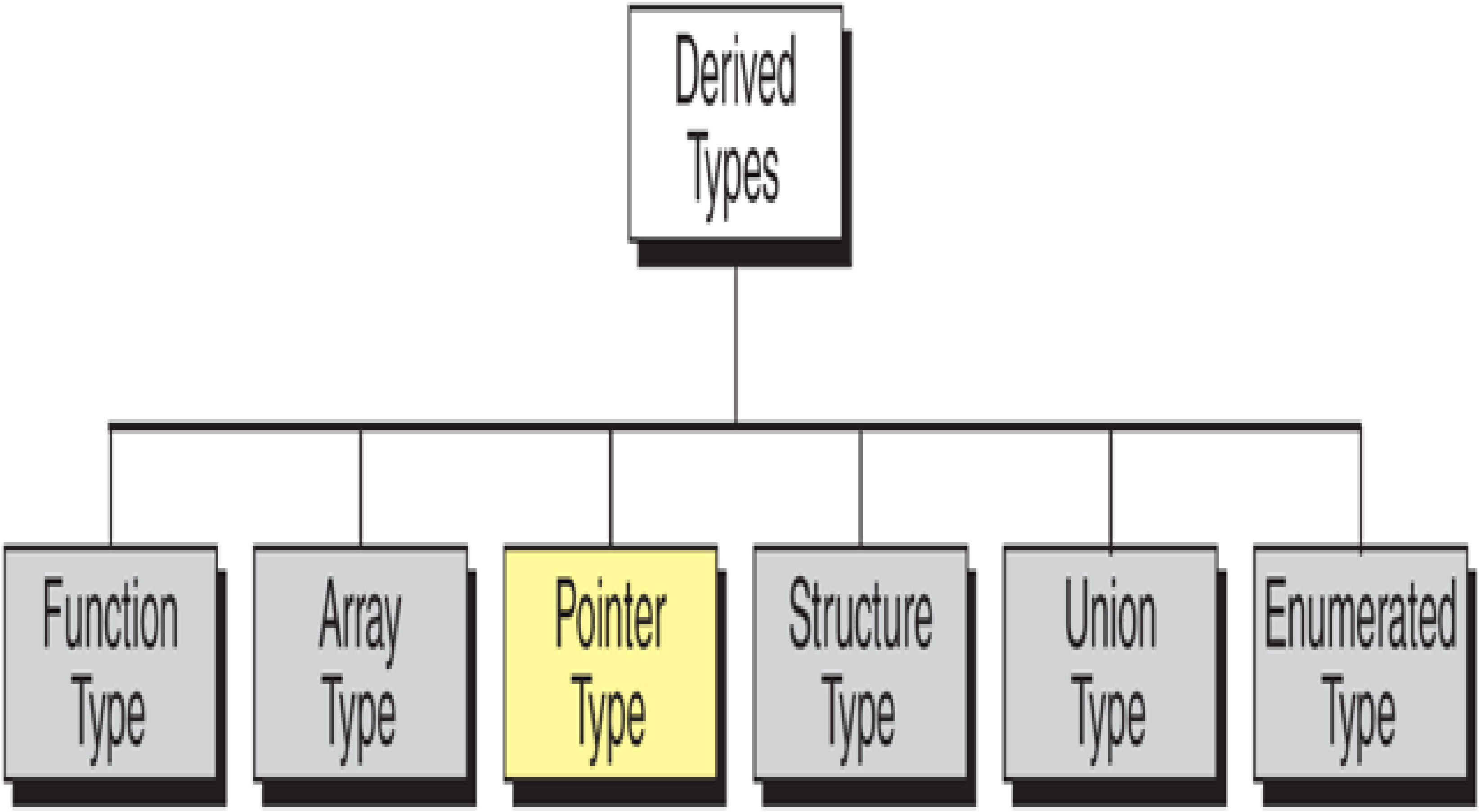
strings and pointer,

string handling functions

Pointers

Objectives

- To understand the concept and use of pointers
- To be able to declare, define, and initialize pointers
- To write programs that access data through pointers
- To use pointers as parameters and return types
- To understand pointer compatibility, especially regarding pointers to pointers



Derived Types

Pointers:

- A pointer is a constant or variable that contains an address that can be used to access data

- There are three concepts associated with the pointers are:

Pointer Constants
Pointer Values
Pointer Variables

POINTER CONSTANT

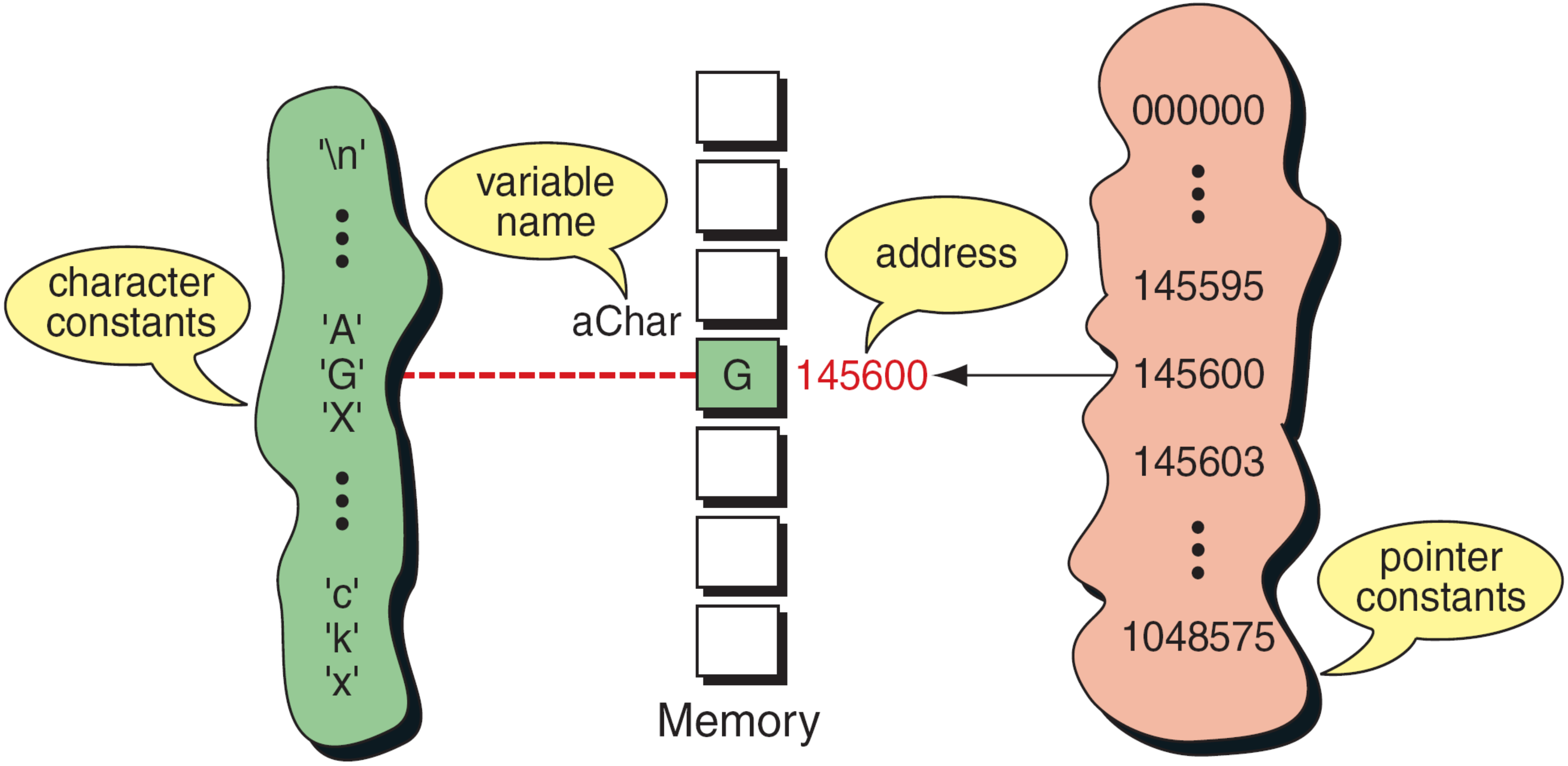
- The computer’s memory is a sequential collection of ‘storage cells’.

- Each cell can hold one byte of information, has a unique number associated with it called as ‘address’

- We cannot change them, but we can only use them to store data values.

- These memory addresses are called pointer constants.

Pointer constants, drawn from the set of addresses for a computer, exist by themselves. We cannot change them; we can only use them.



Pointer Constants

POINTER VALUE

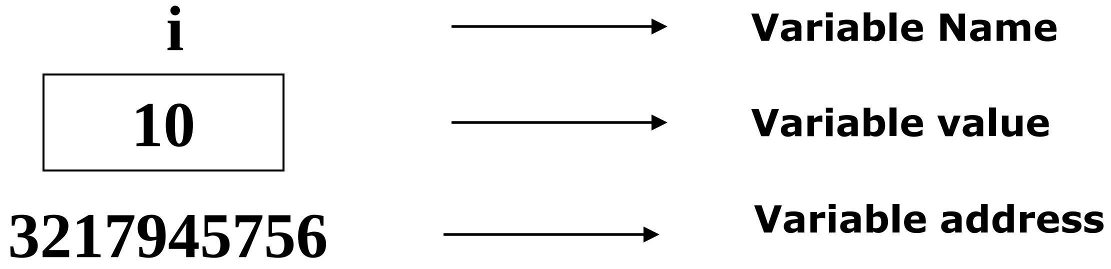
- Whenever we declare a variable, the system allocates, an appropriate location to hold the value of the variable somewhere in the memory,.

- Consider the following declaration,

```
int i=10;
```

- This declaration tells the C compiler to perform the following activities:

- Reserve space in the memory to hold the integer value.
- Associate the name i with this memory location.
- Store the value 10 at this location.
- We can represent i's location in the memory by the following memory map:



i → Variable Name

10 → Variable value

3217945756 → Variable address

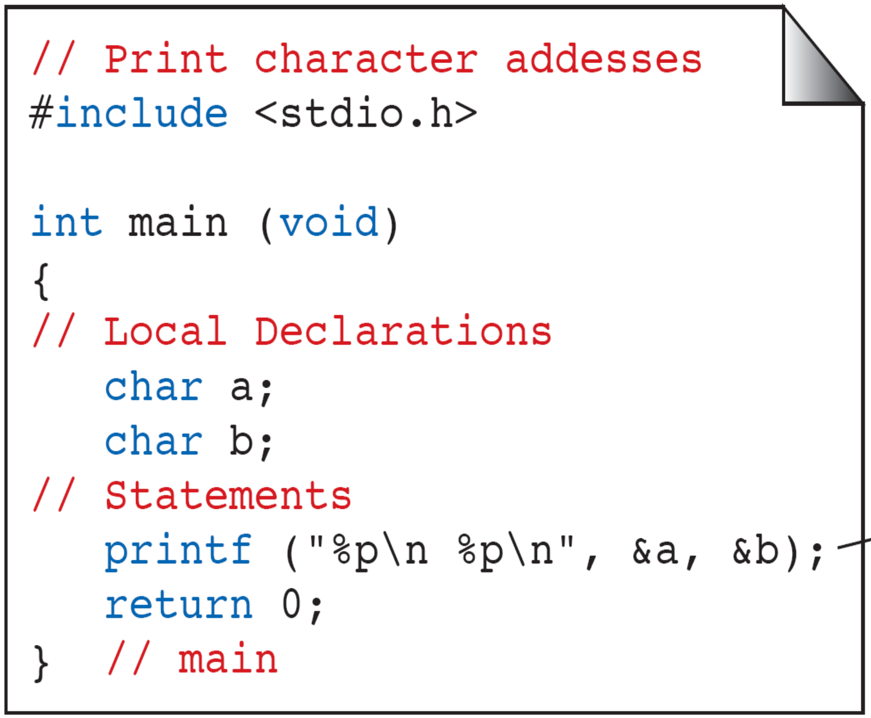
The & Operator:

- The address of the variable cannot be accessed directly. The address can be obtained by using address operator (&) in C language

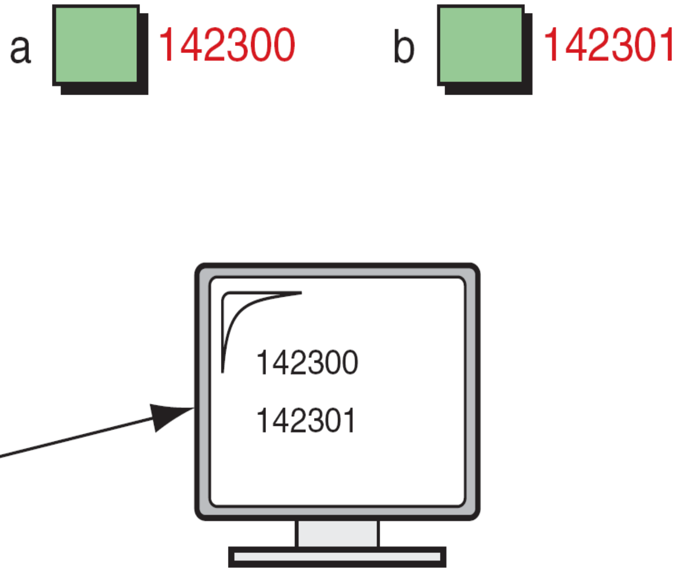
- The address operator can be used with any variable that can be placed on the left side of an assignment operator.

- The format specifier of address is %u(unsigned integer),the reason is addresses are always positive values.

- An address expression(unary expression) consists of an ampersand (&) and a variable name.
- The address operator (&) extracts the address for a variable.
- The address operator format is: **&variable_name**



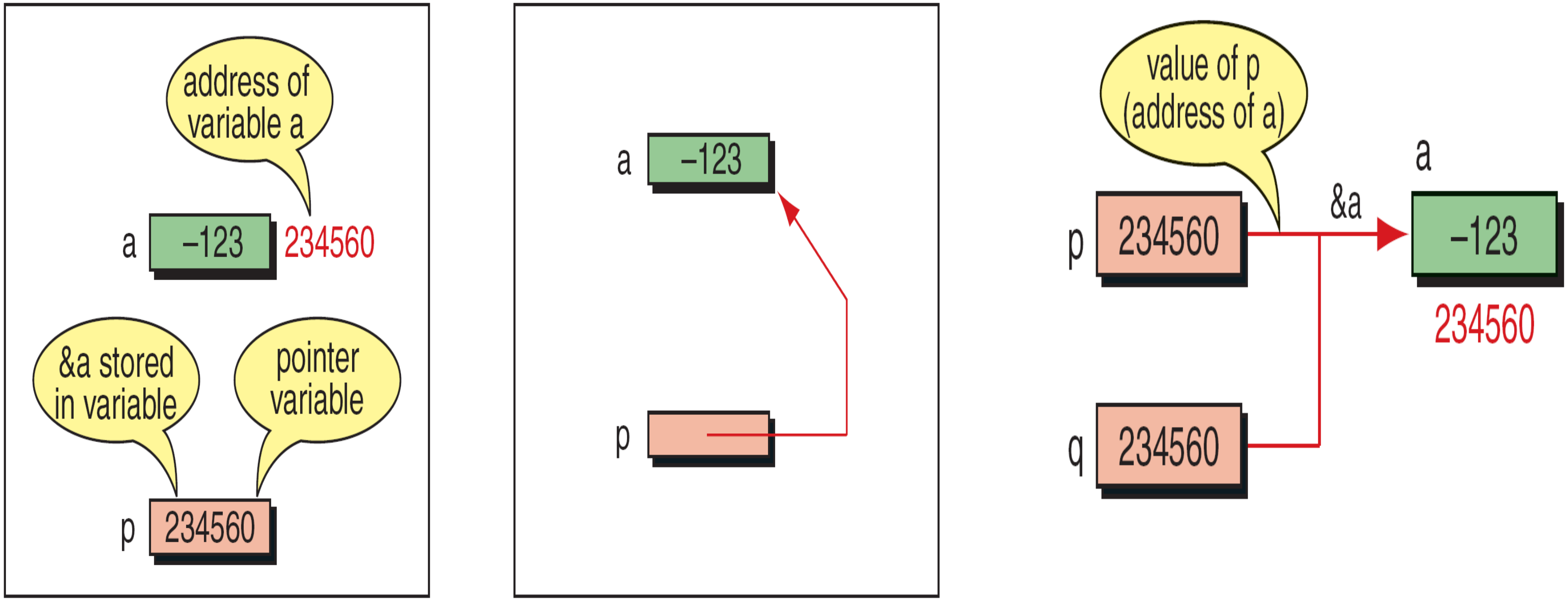
```
// Print character addesses
#include <stdio.h>

int main (void)
{
// Local Declarations
   char a;
   char b;
// Statements
   printf ("%p\n %p\n", &a, &b);
   return 0;
}  // main
```



Print Character Addresses

Pointer is a variable that holds address of another variable of same data type.

Note: A variable's address is the first byte occupied by the variable.



Physical representation

Logical representation

Pointer Variable

Pointer declaration and definition:

General syntax of pointer declaration is,

Data type *pointername;

Data type of a pointer must be same as the data type of a variable to which the pointer variable is pointing

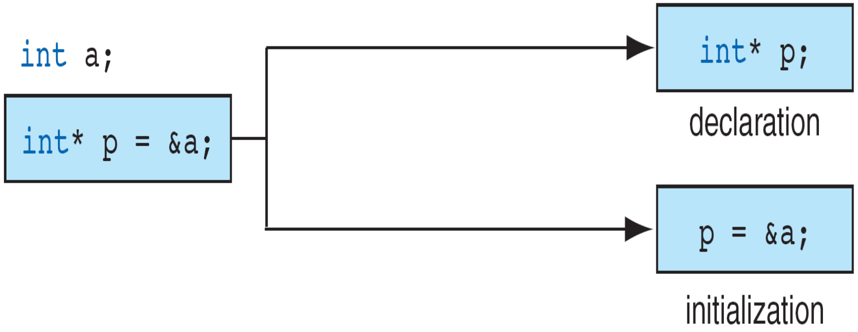
Examples:
int *ip
char *ch
float *fp
double *dp

Pointer initialization:

Pointer Initialization is the process of assigning address of a variable to pointer variable. Pointer variable contains address of variable of same data type.

Address operator or Reference operator(&):is used to determine the address of a variable, put the & in front of variable name returns the address of the variable associated with it.

```
int  a = 10 ;
int  *p;
p=&a
```



```
int a;
int* p = &a;
```

```
int* p;
```

declaration

```
p = &a;
```

initialization

Accessing Variables Through Pointers

Dereference Operator(*):

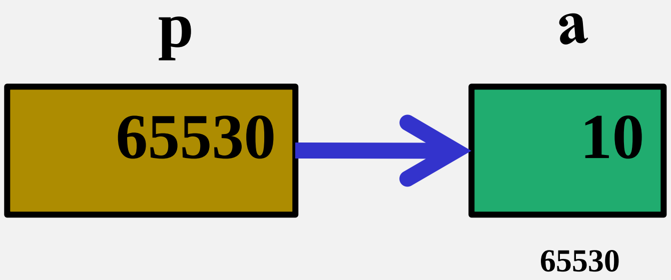
Dereference operator used to give the Value at Address

An **indirect expression**, one of the expression types in the unary expression category, is coded with an **asterisk (*) and an identifier**.

The indirection operator is a unary operator whose operand must be a pointer value.

For example, to access the variable **a** through the pointer p, we simply code *p

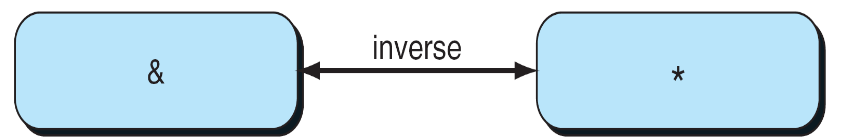
The indirection operator is: *p



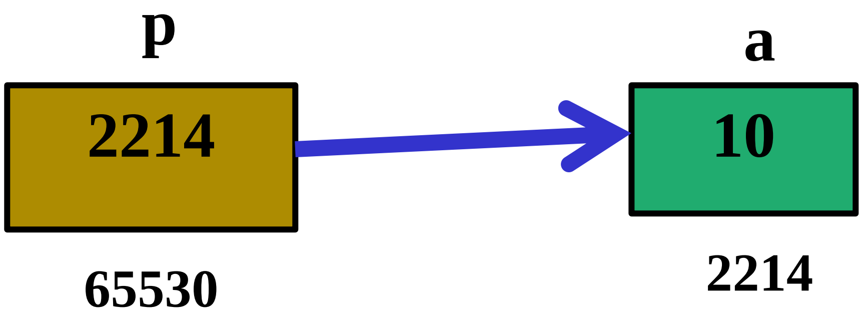
Example:

```
int a,*p;
a = 10;
 p = &a;
printf("%d",*p); //this will print the value of a.
  *p=20;                    // assign  value to variable using pointer
printf("%d",*&a); //this will also print the value of a
```

The Address and Indirection operators are the inverse of each other.



```
// sample program  on pointer

#include<stdio.h>
void  main()
{
int a=10;
int *p;
p=&a;
printf(“%d”,a);                //this will print value of a (10)
printf(“%u”,&a);               //this will print address of a (2214)
printf(“%u”, p);               //this will print value of p (2214)
printf(“value=%d”,*p);         // this will print value at  address (10)
printf(“%u”,&p);               //this will print address of p (65530)
printf(“%d”,*(&a));            // this will also print the value of a (10)

}
```



```
// One pointer for many varibles

#include<stdio.h>
void  main()
{
int a=10,b=5,c=20;
int *p;
p=&a;
printf(“value of a=%d”,*p);
p=&b;
printf(“value of b=%d”, *p);
p=&c;
printf(“value of c=%d”,*p);
}
```

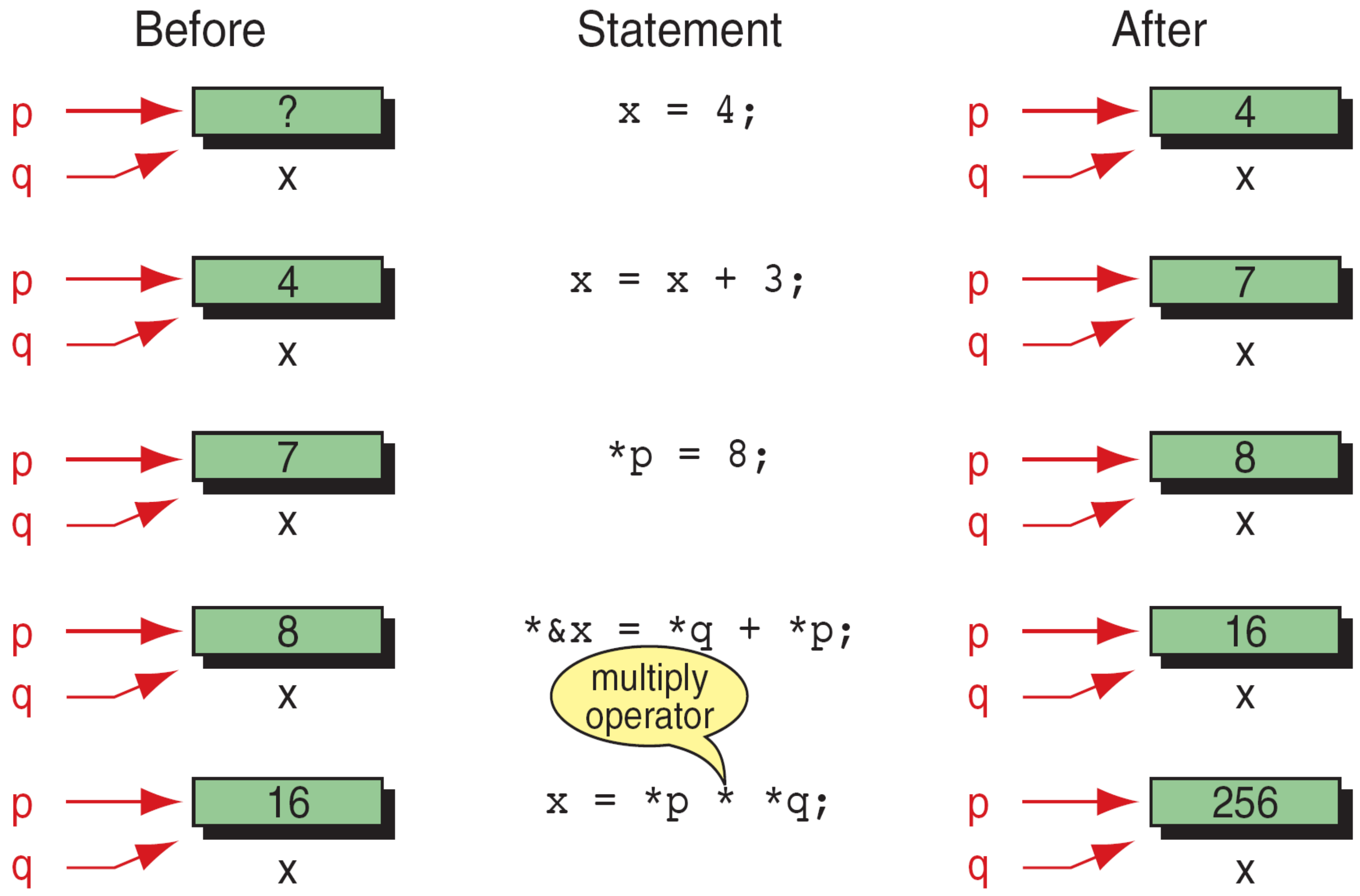
```
// Many pointers for  single varible

#include<stdio.h>
void  main()
{
int a=10,b=5,c=20;
int *p,*q,*r;
p=&a;
printf(“value of a=%d”,*p);
q=&a;
printf(“value of a=%d”, *q);
r=&a;
printf(“value of a=%d”,*r);
}
```

Assign pointer to pointer

```
#include<stdio.h>
void  main()
{
int a=10;
int *p,*q,*r;
p=&a;
printf(“value of a=%d”,*p);
q=p;
printf(“value of a=%d”, *q);
r=p;
printf(“value of a=%d”,*r);
}
```

Function returning pointer

```
#include<stdio.h>
int *  min(int  *pa, int  *pb);
void  main()
{
int  a,b, *p;
printf(“enter a and b”);
scanf(“%d%d”,&a,&b);
p=min(&a,&b);
printf(“minimum is=%d”,*p);
}
int * min(int *pa,int *pb)
{
return(*pa < *pb?pa:pb);
}
```



```
x = 4;
x = x + 3;
*p = 8;
*&x = *q + *p;
x = *p * *q;
```

Accessing Variables Through Pointers

Declaration versus Redirection

An asterisk operator can be used in two different contexts:
For declaration and for redirection.

Declaration: When an asterisk is used for declaration,
it is associated with a type.

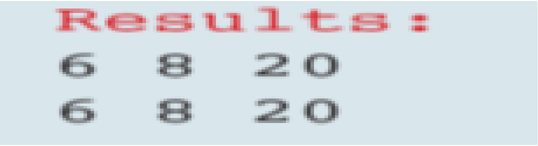
For example, we define a pointer to an integer as

```
int *pa;
int *pb;
```

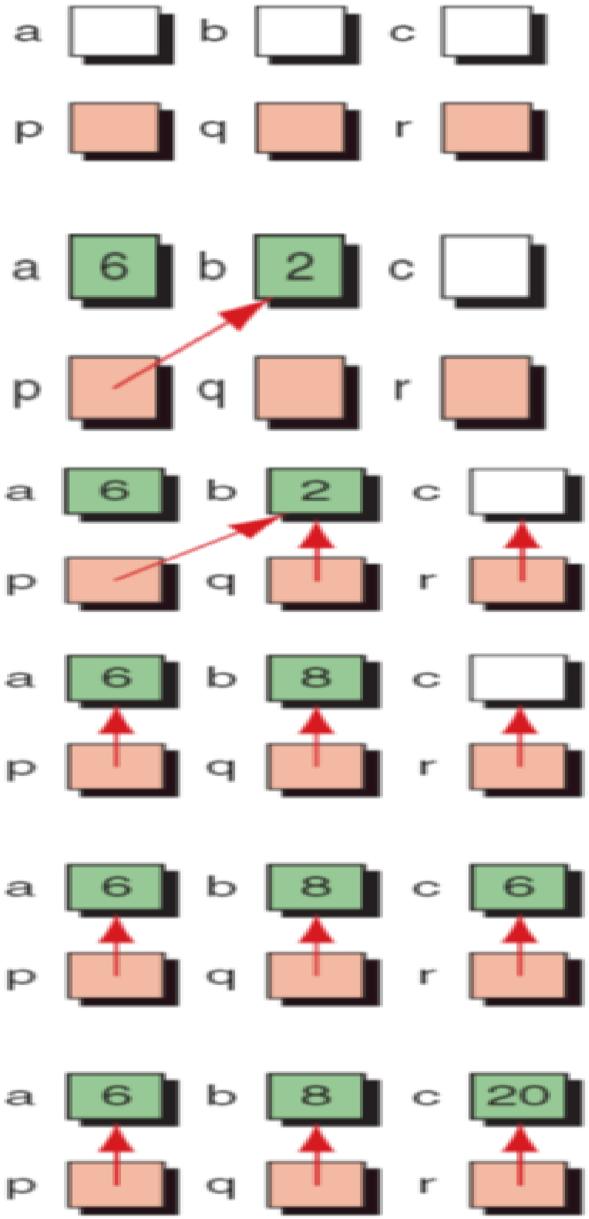
Redirection: When used for redirection, the asterisk is an operator that redirects the operation from the pointer variable to a data variable.

For example, given two pointers to integers, pa and pb, sum is computed as **int sum = *pa + *pb;**

```
// sample program  on pointer

#include<stdio.h>
void  main()
{
int a,b,c
int *p,*q,*r;
a=6;b=2;
p=&b;
q=p;
r=&c;
p=&a;
*q=8;
*r=*p;
*r=a+ *q + *(&c);
printf(“%d %d %d” ,a,b,c);
printf(“%d %d %d”,*p,*q,*r));
}
```



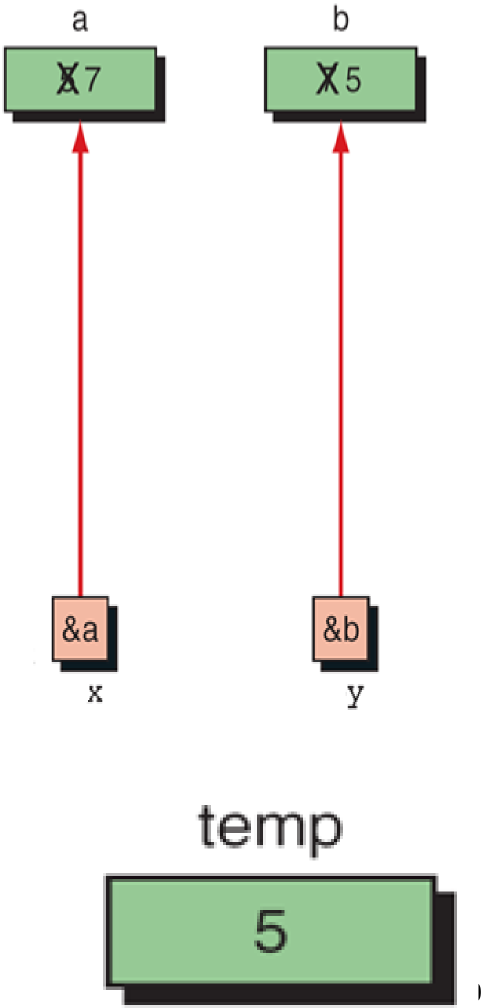
```
Results:
6   8   20
6   8   20
```



Pointer for Inter-function Communication

One of the most useful applications of pointers is in functions, C uses the **pass-by-value** for **downward** communication and the **pass-by-address** for **upward** communication.

```
#include<stdio.h>
void swap(int*,int*);
main()
{
int a,b;
printf("Enter two numbers");
scanf("%d%d",&a,&b);
printf(“the values before swapping a=%d,b=%d”,a,b);
swap(&a,&b);
printf("The values after swapping a=%d,b=%d",a,b);
}
void swap(int *x,int *y)
{
            int temp;
            temp=*x;
            *x=*y;
            *y=temp;
```



Every time we want a called function to have access to a variable in the calling function, we pass the address of that variable to the called function and use the indirection operator to access it.

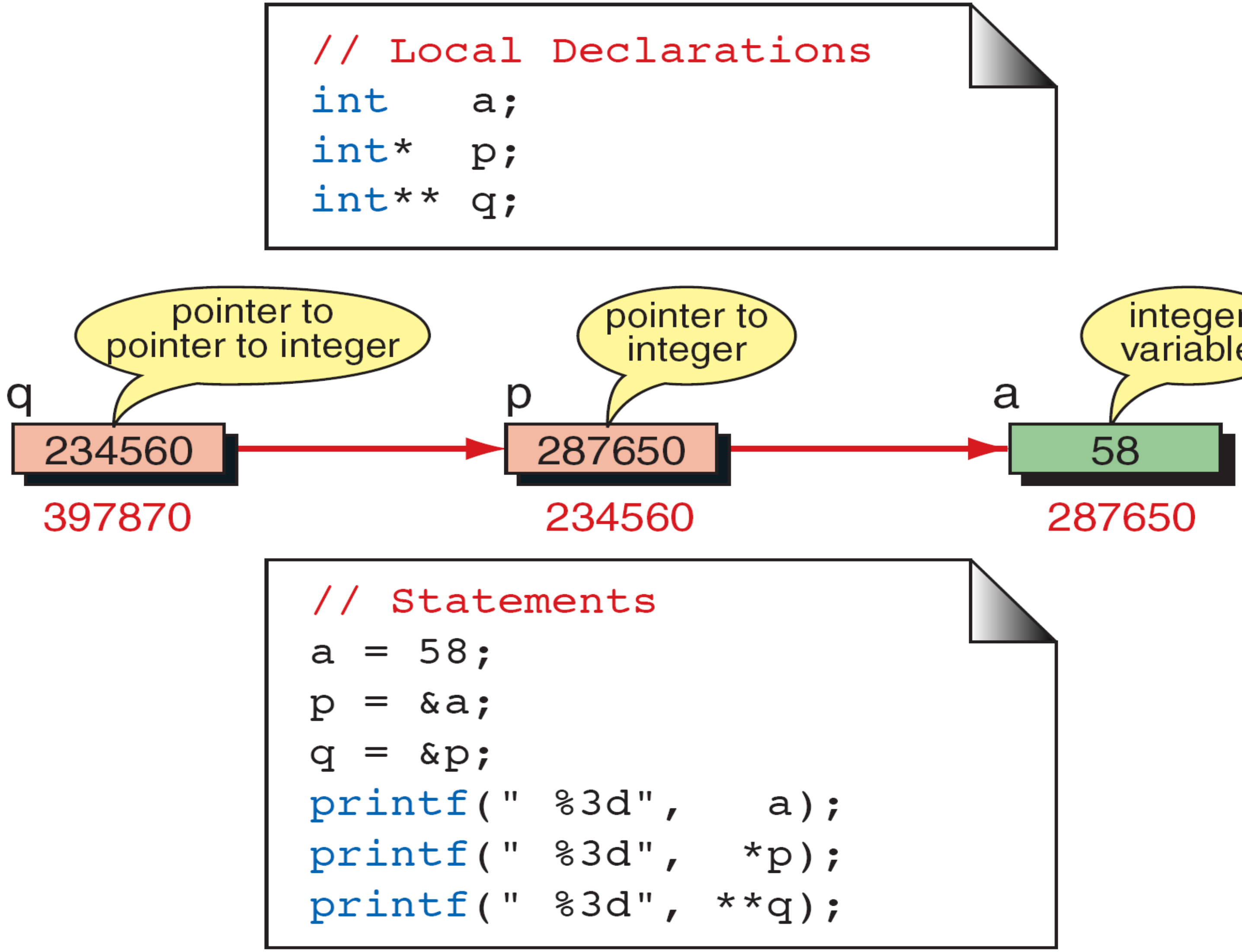
When several values need to be sent back to the calling function, use address parameters for all of them. Do not return one value and use address Parameters for the others.

Note

A void pointer cannot be dereferenced.

Pointers to Pointers

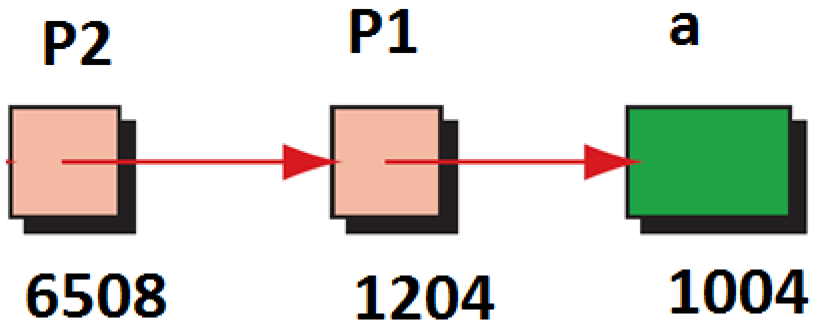
Pointer to pointer is **pointer variable which stores the address of another** **pointer variable.**



```
// Local Declarations
int     a;
int*    p;
int**   q;
```

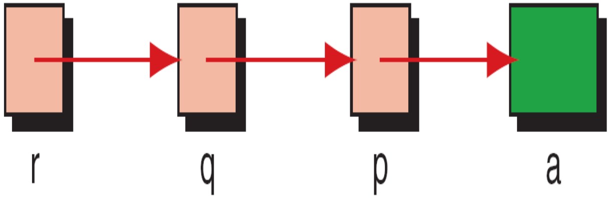
```
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

Pointer to Pointer



```
#include<stdio.h>
void  main()
{
int a=10;
int *p1;
int  **p2;
p1=&a;
p2=&p1;
printf(“%u”,&a);          //this will print address of a (1004)
printf(“%u”, &p1);        //this will print address of p1 (1204)
printf(“%u”,&p2);         // this will print  address of p2(6508)
printf(“%u”,*p2);         //this will print value (1004)
printf(“%d”,*p1);         // this will also print the value of a (10)
printf(“%d”,**p2);         // this will also print the value of a (10)
}
```

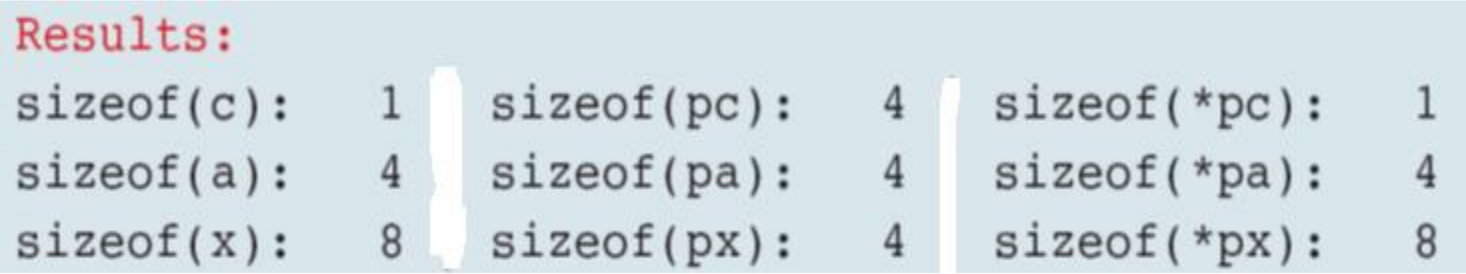
Pointer to Pointer



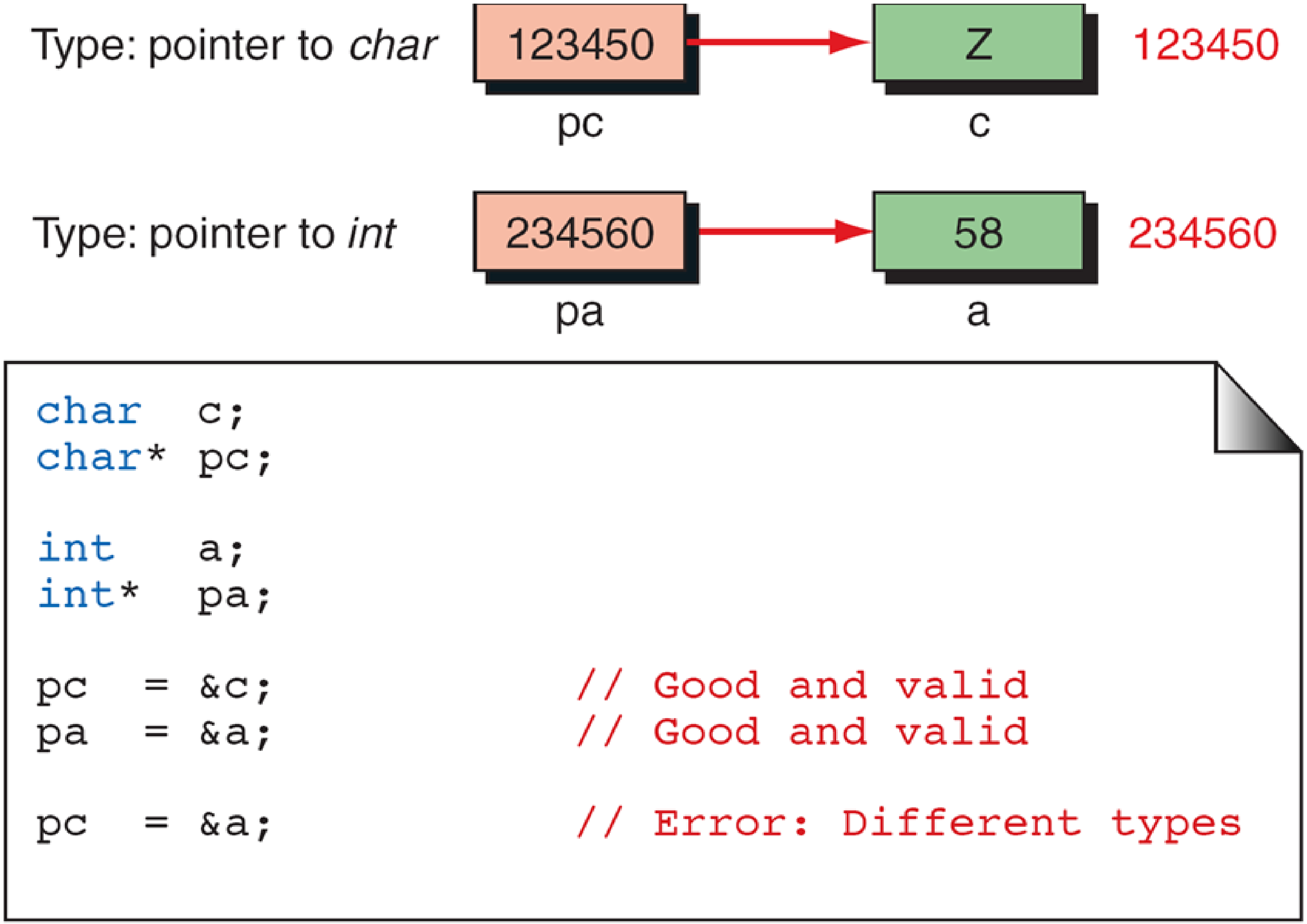
```
#include<stdio.h>
void  main()
{
int a;
int *p;    int  **q; int  ***r;
p=&a;    q=&p;    r=&q;
printf(“enter number”);                    //using  a
scanf(“%d”,&a);
printf(“Number is:%d”,a)
printf(“enter number”);                    //using  p
scanf(“%d”,p);
printf(“Number is:%d”,a)
printf(“enter number”);                    //using q
scanf(“%d”,*q);
printf(“Number is:%d”,a)
printf(“enter number”);                    // using r
scanf(“%d”,**r);
printf(“Number is:%d”,a)
}
```

Demonstrate size of Pointers

```
#include<stdio.h>
void  main()
{
char      c,*pc;
int        a, *pa;
double   x,*px;
printf(“size of c=%d”, sizeof(c));
printf(“size of  pc=%d”, sizeof(pc));
printf(“size of  starpc=%d”, sizeof(*pc));
printf(“size of  a=%d”, sizeof(a));
printf(“size of  pa=%d”, sizeof(pa));
printf(“size of  starpa=%d”, sizeof(*pa));
printf(“size of x=%d”, sizeof(x));
printf(“size of  px=%d”, sizeof(px));
printf(“size of  starpx=%d”, sizeof(*px));
}
```



```
Results:
sizeof(c):   1   sizeof(pc):   4   sizeof(*pc):   1
sizeof(a):   4   sizeof(pa):   4   sizeof(*pa):   4
sizeof(x):   8   sizeof(px):   4   sizeof(*px):   8
```



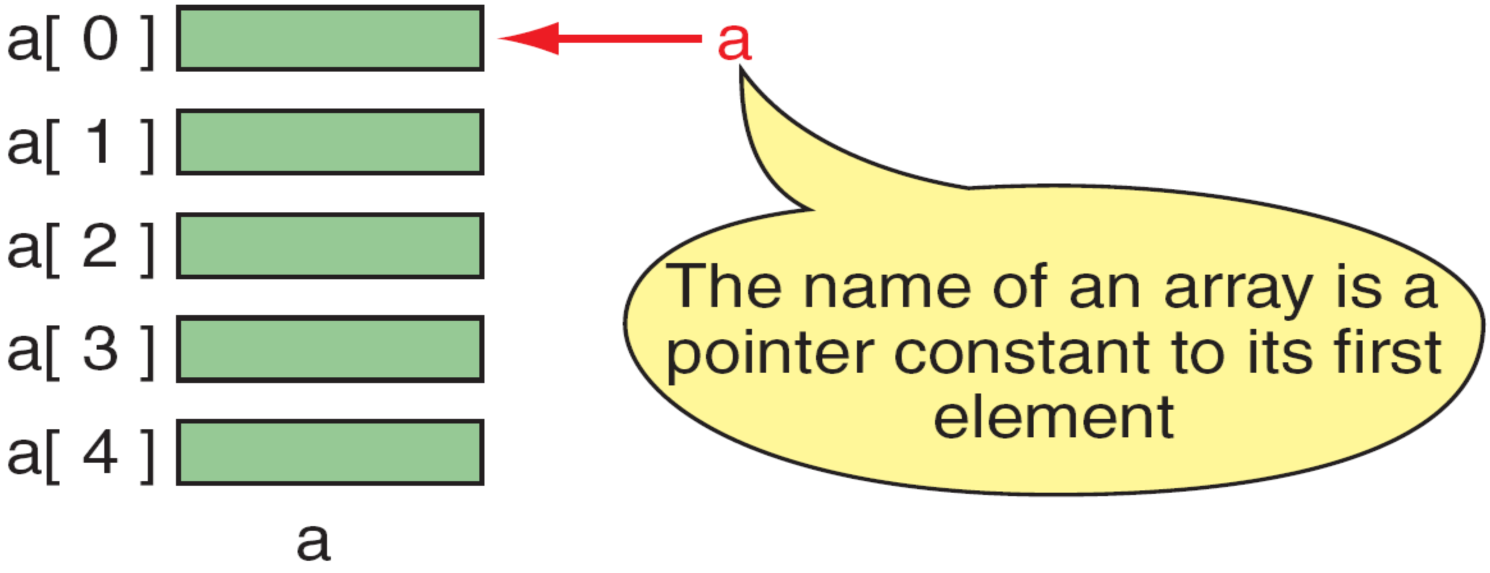
```
char  c;
char* pc;

int   a;
int*  pa;

pc  = &c;                   // Good and valid
pa  = &a;                   // Good and valid

pc  = &a;                   // Error: Different types
```

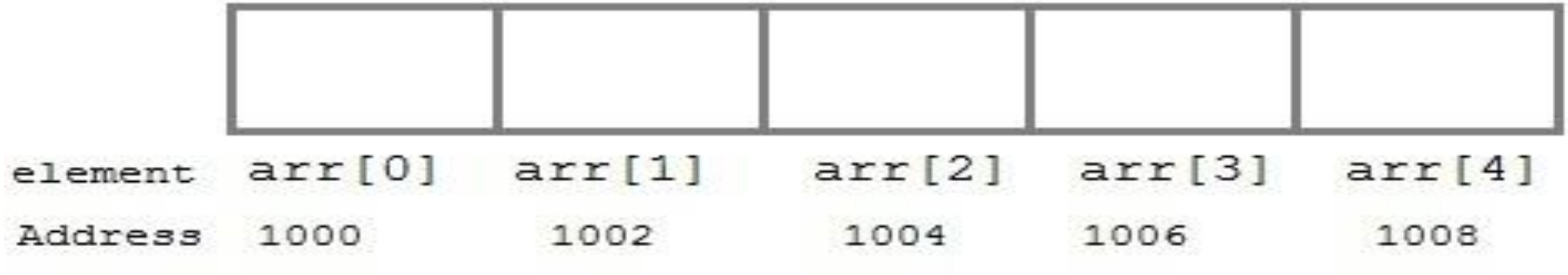
Dereference Type Compatibility

Arrays and Pointers

The name of an array is a pointer constant to the first element. Because the array's name is a pointer constant, its value cannot be changed.

Since the array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory.


a[0]
a[1]
a[2]
a[3]
a[4]
a
a
The name of an array is a pointer constant to its first element



element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

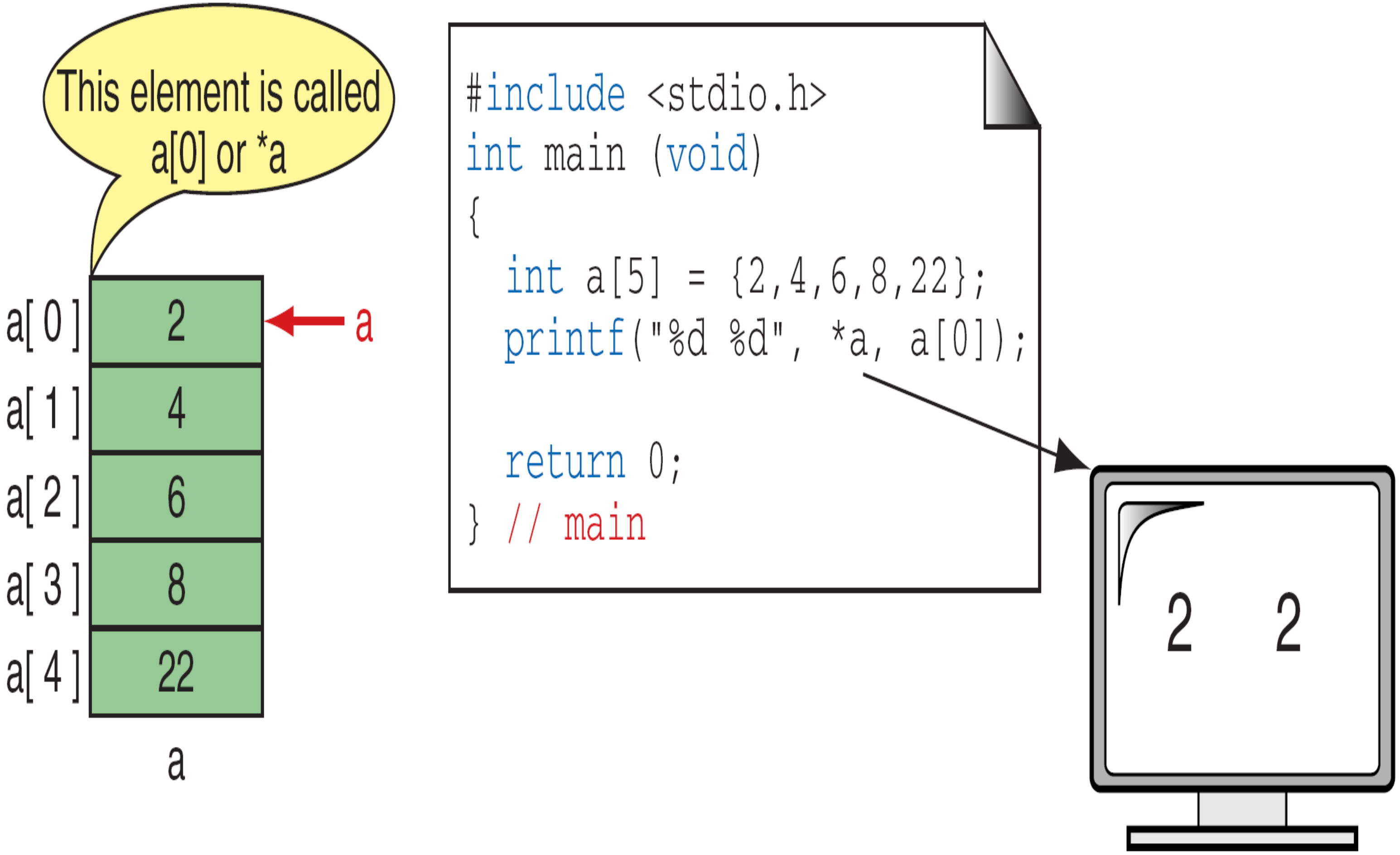
same

arr ↔ &arr[0]

a **is a pointer only to the first element—not the whole array.**

The name of an array is a pointer constant;

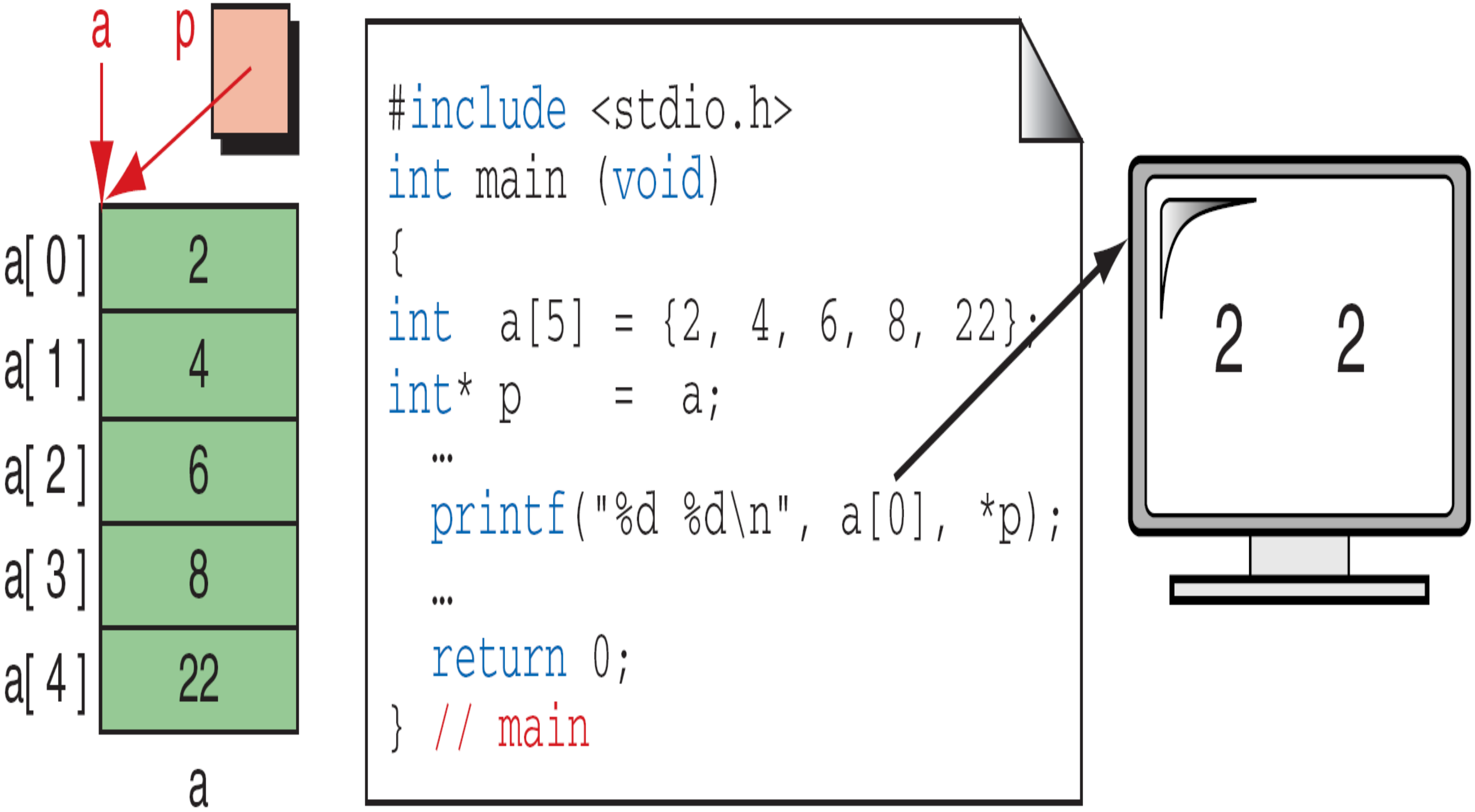
To access an array, any pointer to the first element can be used instead of the name of the array.



```
#include <stdio.h>
int main (void)
{
  int a[5] = {2,4,6,8,22};
  printf("%d %d", *a, a[0]);

  return 0;
} // main
```

```
2   2
```

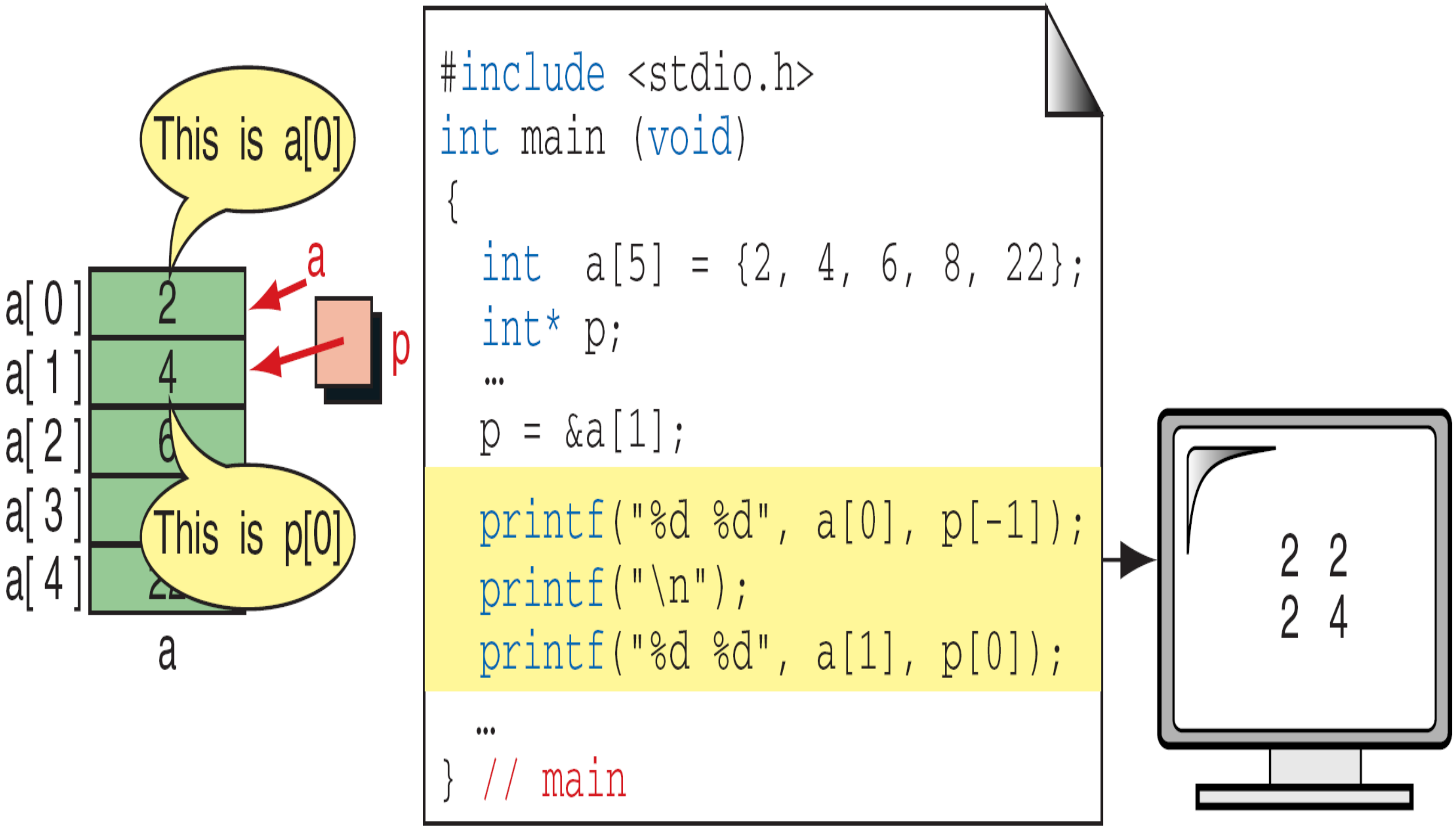
Dereference of Array Name



```
#include <stdio.h>
int main (void)
{
int  a[5] = {2, 4, 6, 8, 22};
int* p    =  a;
  …
  printf("%d %d\n", a[0], *p);
  …
  return 0;
} // main
```

```
2  2
```

Array Names as Pointers



```
#include <stdio.h>
int main (void)
{
  int  a[5] = {2, 4, 6, 8, 22};
  int* p;
  …
  p = &a[1];

  printf("%d %d", a[0], p[-1]);
  printf("\n");
  printf("%d %d", a[1], p[0]);

  …
} // main
```

```
2 2
2 4
```

Multiple Array Pointers

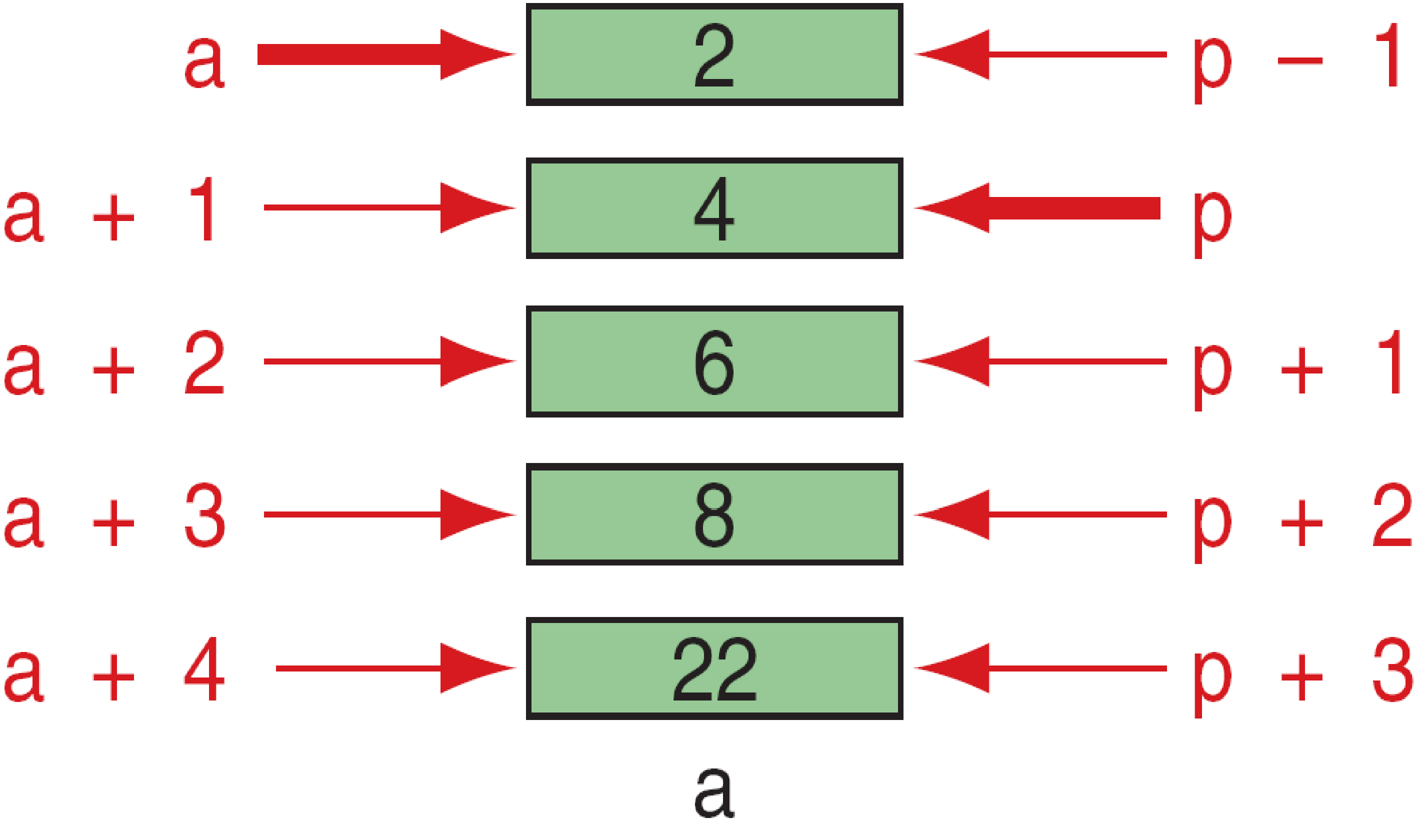
Pointer Arithmetic and Arrays

Besides indexing, programmers use another powerful method of moving through an array: **pointer arithmetic**.

Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the addresses in pointers.

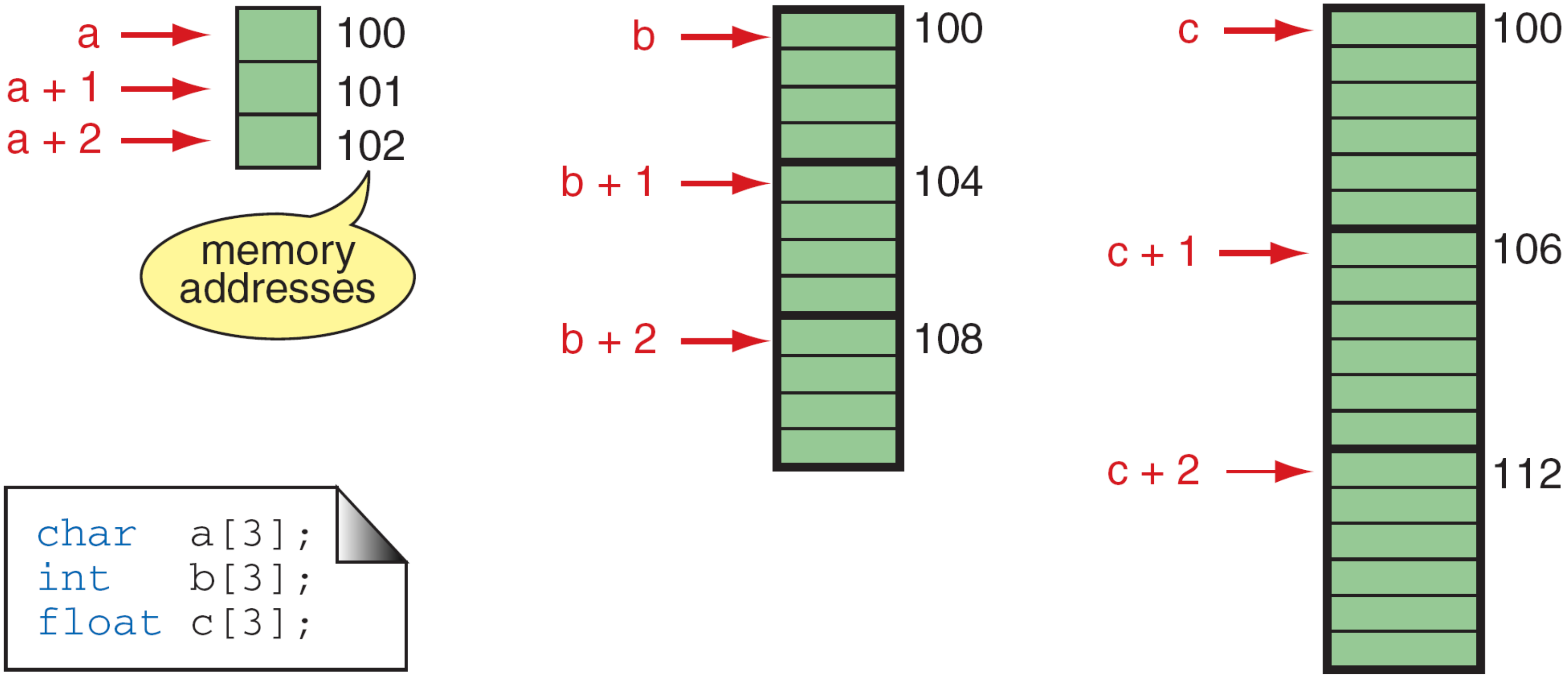
- **Pointers and One-Dimensional Arrays**
- **Arithmetic Operations on Pointers**
- **Using Pointer Arithmetic**
- **Pointers and Two-Dimensional Arrays**

Given pointer, p, p ± n is a pointer to the value n elements away.



Pointer Arithmetic

Address of a[i]= Baseaddress+ i*size of element



```
char  a[3];
int   b[3];
float c[3];
```

Pointer Arithmetic and Different Types

The following expressions are identical.
*(a + n) and a[n]

i[a] or a[i] or *(a+i)

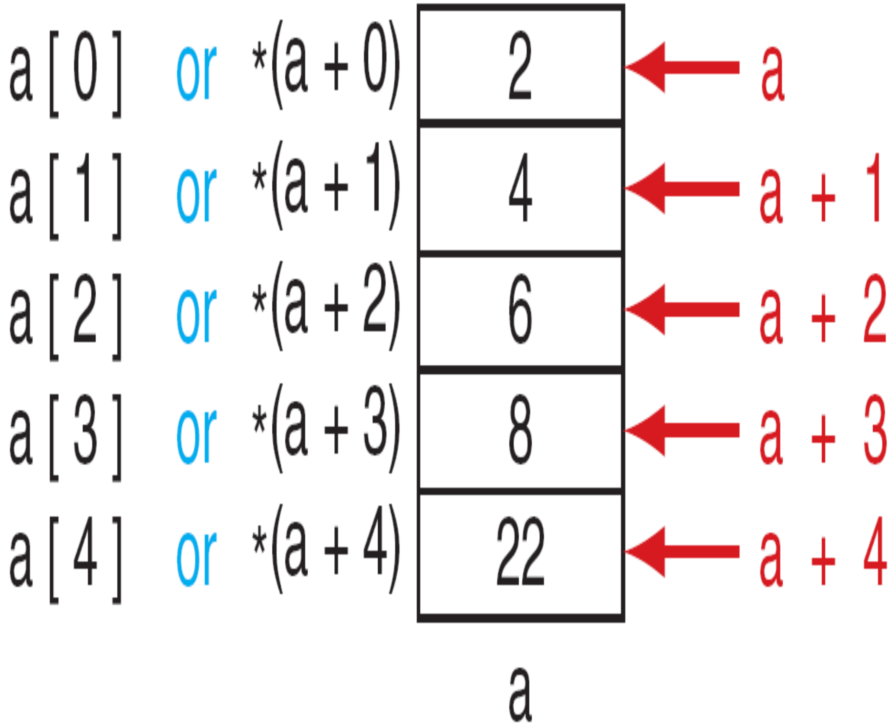
P[0] or 0[a] or

P[1] or 1[a] or

P[2] or 2[a] or

P[3] or 3[a] or

P[4] or 4[a] or



Some valid arithmetic operations on pointers:

p + 5

5 + p

p -5

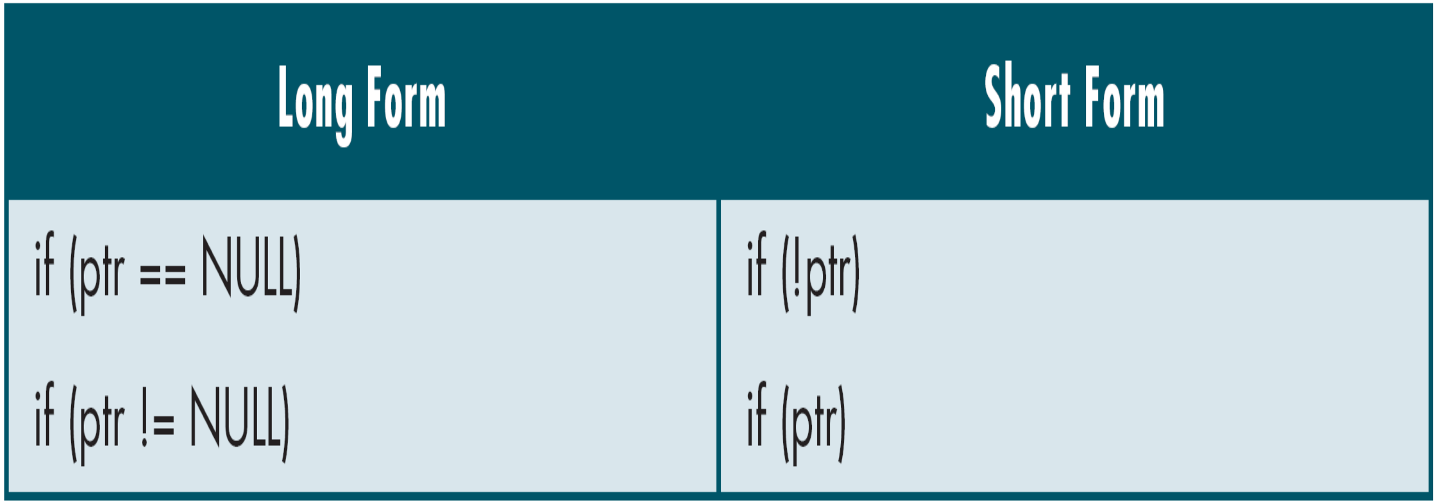
p ++

p --

p1 – p2 When one pointer is subtracted from another, the result is number of elements between two pointers

P1=&a[1] p2=&a[3] (3-1=2)

We should not use *p-1



Long Form	Short Form
if (ptr == NULL)	if (!ptr)
if (ptr != NULL)	if (ptr)

Pointers and Relational Operators

data Type	Initial Address	Operation	Address after Operations	Required Bytes
int	**4000**	++	**4002**	2
int	**4000**	- -	**3998**	2
char	**4000**	++	**4001**	1
char	**4000**	- -	**3999**	1
float	**4000**	++	**4004**	4
float	**4000**	- -	**3996**	4
long	**4000**	++	**4004**	4
long	**4000**	- -	**3996**	4

Expression	Result
Address + Number	Address
Address – Number	Address
Address – Address	Number

Address + Address = Illegal
Address * Address = Illegal
Address / Address = Illegal
Address % Address = Illegal
Address & Address = Illegal
Address | Address = Illegal
Address ^ Address = Illegal
~Address = Illegal

Accessing one dimensional array using pointer:

we can access every element of array using either array name or pointer which point to any one element of array

For example:

```
void main()
{
int  x[5]={2,4,6,8,10};
int *p;
p=&x[1];
printf("Array elements are using array name \n");
for(i =0; i < 5; i++)
printf("%4d", *(x+i));                              //It gives2 4 6 8 10
printf("Array elements are using pointer \n");
for(i =0; i < 5; i++)
printf("%4d", *(p+i));                              //It gives 4 6 8 10
}
```

Read and print array using pointer

```
#include<stdio.h>6
void main()
{
       int a[12];
       int i ,n,*p;
      printf(enter size of array”);
       scanf(“%d”,&n);
       printf(“enter values”);
       for(p=a;p<a+n;p++)
      {
        scanf(“%d”,p);
        }
      printf(“array elements are:);
      for(p=a;p<a+n;p++)
      {
        printf(“%3d”,*p);
        }
}
```

Read and print array using pointers

```
#include<stdio.h>
#define size 5
void main()
{
       int a[size];
       int*  p ;
       printf(“enter values”);
       for(p=a;p<a+size;p++)
      {
        scanf(“%d”,p);
        }
      printf(“array elements are:);
      for(p=a;p<a+size;p++)
      {
        printf(“%3d”,*p);
        }
}
```

```
#include<stdio.h>
void main()
{
      int a[15];
      int  *min,n;
      printf(“enter size of array”);
      scanf(“%d”,&n);
      printf(“enter values”);
      for(p=a;p<a+n;p++)
     {
       scanf(“%d”,p);
      }
    min=a;
    for(p=a+1;p<a+n;p++)
     {
        if(*p <*min)
        *min=*p;
     }
       printf(“minimum=%3d”,*min);
 }
```

Passing an Array to a Function

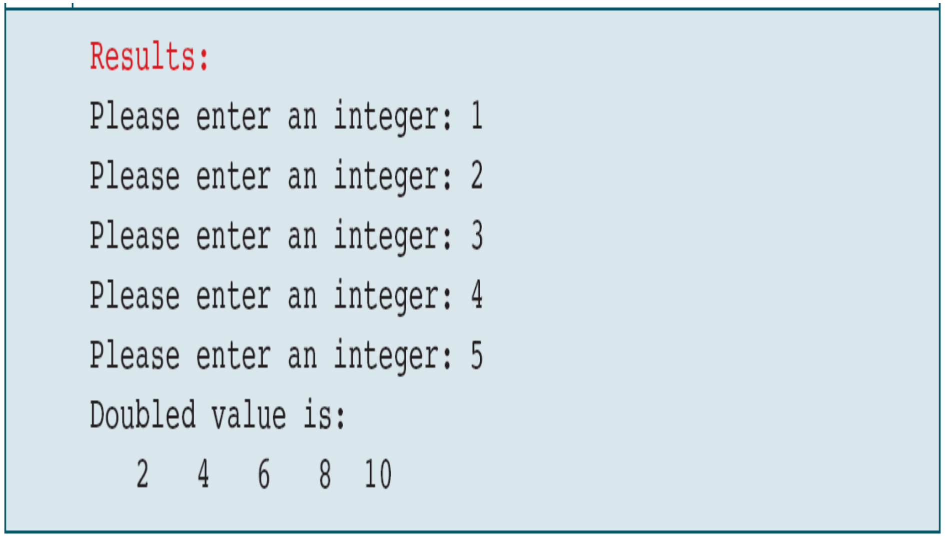
Now that we have discovered that the name of an array is actually a pointer to the first element, we can send the array name to a function for processing. When we pass the array, we do not use the address operator.

Remember, the array name is a pointer constant, so the name is already the address of the first element in the array.

Passing an Array to a Function

```
#include<stdio.h>
void multiply(int *pa, int n)
{
int *j;
for(j=pa; j<pa+n ;j++)
*j=*j * 2;
}
```

```
int main()
{
int a[15];
int *i, n;
printf(“enter n”);
scanf(“%d”,&n);
printf(“enter array elements”);
for(i=a; i< a+n; i++)
scanf("%d", i);  }
multiply(a, n) ;
printf(" Doubled value is:\n");
For(i=a; i<(a+n);i++)
          {  printf("%2d", *i);  }
return 0;
}
```



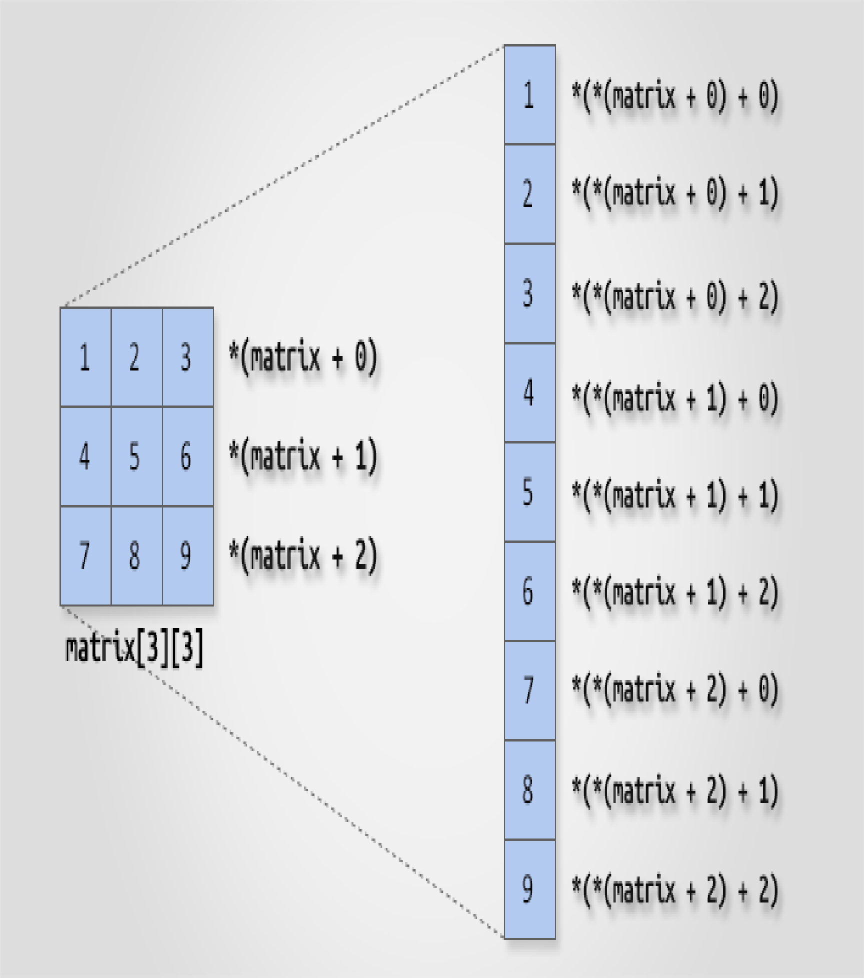
```
Results:
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 4
Please enter an integer: 5
Doubled value is:
   2   4   6   8  10
```

Pointers and two dimensional arrays:

Let us consider a two dimensional array **A[i][j]** then array elements can be accessed through pointer index notation using ***(*(A + i) + j)**

For example:

```
void main()
{
int  A[3][4]={2,11,24,42,31,19,72,4,6,9,1,5};
printf("Array elements are\n");
for(i =0; i < 3; i++)
{
     for(j =0; j < 4; j++)
    {
          printf("%d", *(*(A + i) + j));
     }
}
```



We recommend index notation for two-dimensional arrays.

Array of Pointers

An **array of pointers** is an indexed set of variables in which the variables are pointers .This structure is especially helpful when the number of elements in the array is **variable.**

Syntax:int *p[10]

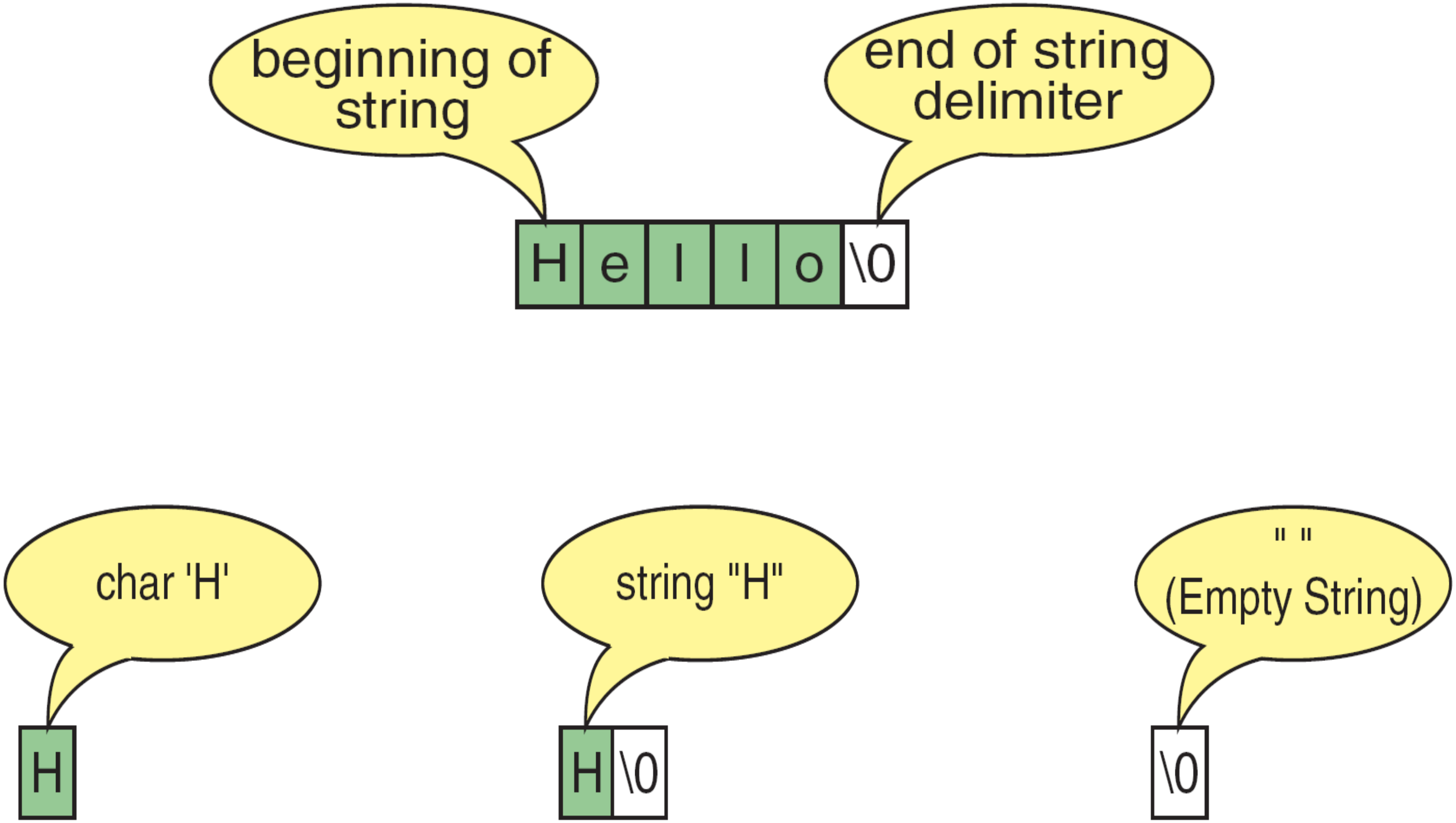
```
 void main ()
 {
int var[] = {10, 100, 200};
 int i, *ptr[3];
for ( i = 0; i < 3; i++)
{
ptr[i] = &var[i]; /* assign the address of integer. */
}
for ( i = 0; i < 3; i++)
{ printf("Value of var[%d] = %d\n", i, *ptr[i] );
 }
 }
```

C Strings

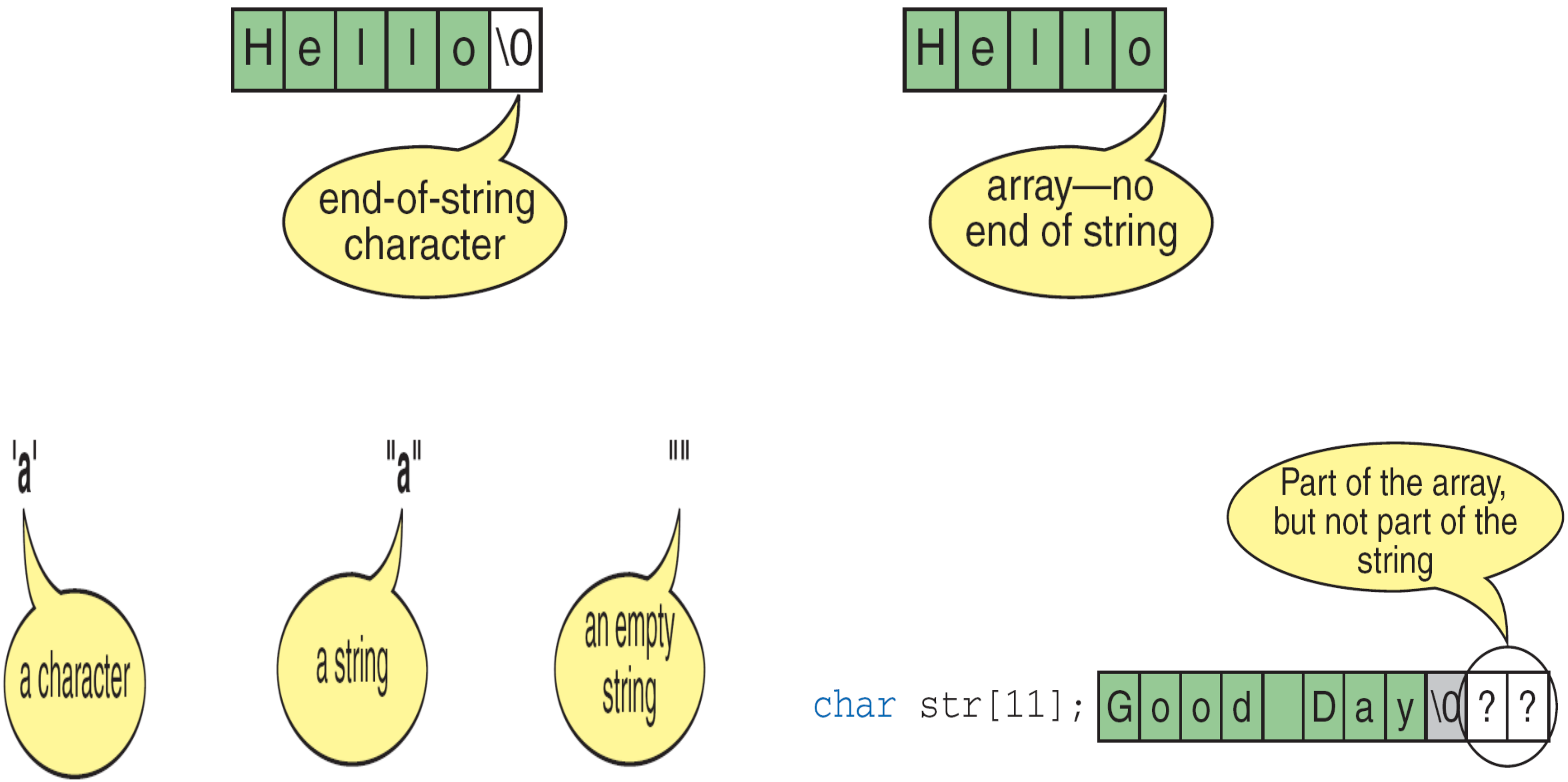
A C **string** is a variable-length array of characters that is delimited by the null character.

A string is a sequence of characters.

A string literal is enclosed in double quotes.

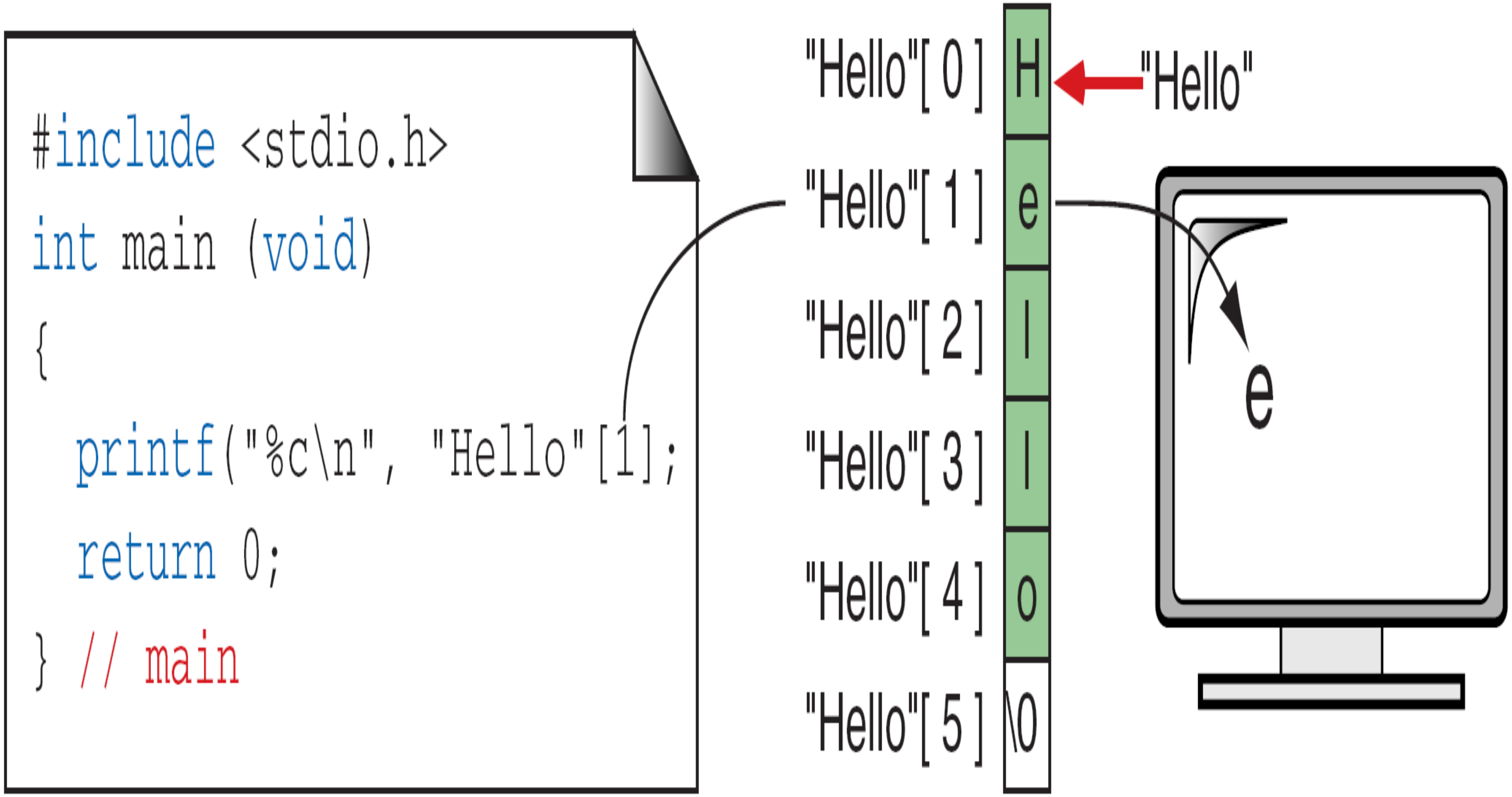


Storing Strings and Characters



```
char str[11];
```

Differences Between Strings and Character Arrays



```
#include <stdio.h>
int main (void)
{
  printf("%c\n", "Hello"[1];
  return 0;
} // main
```

String Literal References

DECLARING AND INITIALIZING STRING VARIABLES

Declaring a String:

A string variable is a valid C variable name and always declared as an array.

The general form of declaration of a string variable is,

char string_name [size];

The size determines the number of characters in the string_name.

When the compiler assigns a character string to a character array, it automatically supplies a null character('\0') at the end of the string.

The **size** should be equal to the maximum number of characters in the string **plus one.**

DECLARING AND INITIALIZING STRING VARIABLES

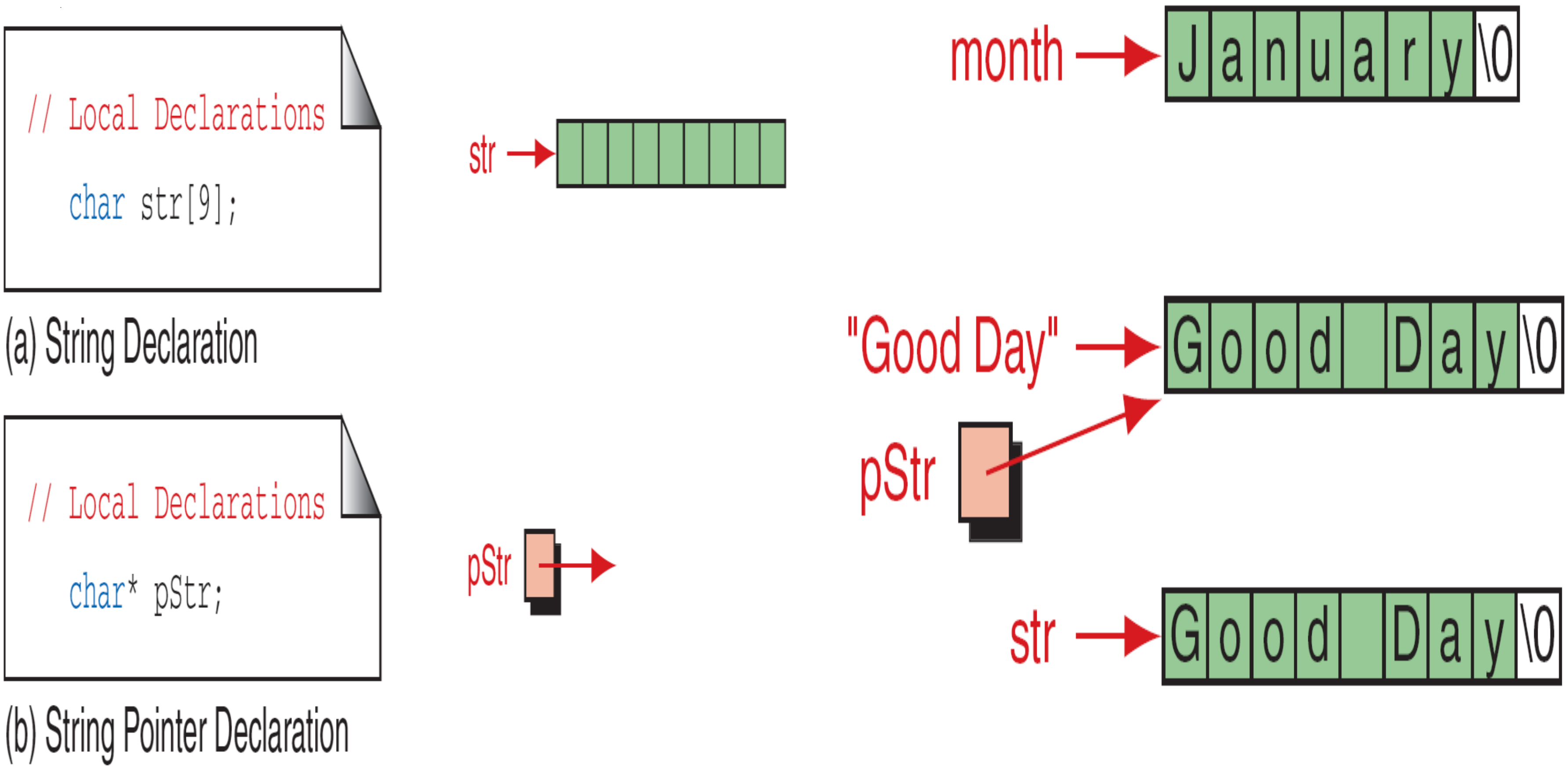
Initializing a String: This can be done in two ways.

1. char str1[7]=“Welcome”;

2. char str2[8]={‘W’,’e’,’l’,’c’,’o’,’m’,’e’,’\0’};

Without size:

3. char str[]=“welcome”;



```
// Local Declarations

   char str[9];
```

(a) String Declaration

```
// Local Declarations

   char* pStr;
```

(b) String Pointer Declaration

Memory for strings must be allocated before the string can be used.

Accessing Strings using pointer

```
void main()
{
  char s[15],r[15];
   char *sp,*rp
  printf("Enter a String : ");
  gets(s);
  sp=&s[2];
   rp=s;
  printf(“ String     : %s\n  pointer=%s",s,sp);
  printf(“\n %c“,*sp);
  printf(“enter string”)
  scanf(“%s”,rp);
 printf(“%s”,s);
}
```

String Input/Output Functions

C provides two basic ways to read and write strings.

First, we can read and write strings with the **formatted** input/output functions, **scanf/fscanf** and **printf/fprintf**.

Second, we can use a special set of string-only functions, get string (**gets/fgets**) and put string (**puts/fputs**).

Formatted input and output functions: **scanf () and printf()**

The string can be read using the scanf function with the format specifier **%s.** Syntax for reading string using scanf function is

scanf("%s", string_name);

Disadvantages:

The termination of reading of data through scanf function occurs, after finding first white space through keyboard. White space may be new line (\n), blank character, tab(\t).

For example if the input string through keyword is "hello world" then only "hello" is stored in the specified string.

The various options associated with printf ()

1. Field width specification
2. Precision specifier
3. Left Justification

Field Width Specification

Syntax: **%ws**

Here, **w** is the specified field width and s indicates that it is a string.

NOTE:

If the string to be printed is larger than the field width w, the entire string will be printed.

If the string to be printed is smaller than the field width w, then appropriate numbers of blank spaces are padded at the beginning of the string so as to ensure that field width w is reached.

Precision Specifier:

Syntax: **%w.ns**

Here, w is the specified field width, n indicates that first n characters have to be displayed and s indicates that the string is being used.

The string is printed right justification by default.

Left justification:

Syntax: **%-w.ns**

Here, - just before w indicates that string is printed using left justification.
w is the field with and s indicates that the string is being printed.

```
include<stdio.h>
void main()
{
  char s[20],s1[20];
printf("enter string");
  scanf("%s",s);
printf(‘enter another string
scanf(“%s”,s1);
  printf("%s",s);
printf("%2s",s);
printf("%10s",s);
printf("%-10s",s);
printf("%-10.5s",s);
printf(%15.2s”,s1);
printf(%-15.2s”,s1);
}
```

```
Output:
Enter string welcome

Enter another string snistengclg

welcome
welcome
  welcome
welcome
welco
     sn
sn
```

Character I/O from keyboard:

To read characters from the keyboard and write to the screen, it takes the following form:

```
        char c = getchar( );     //reads one character from the keyboard
                 putchar(c);     // display the character on the monitor
```

Un-formatted input functions: **gets () and puts()**

C provides easy approach to read a string of characters using gets() function.

Syntax: **gets (string_name);**

The function accepts string from the keyboard. The string entered includes the white spaces. The input will terminate only after pressing <Enter Key>. Once the <Enter key > is pressed, a null character(\0) appended at the end of the string.

Advantage: It is capable of reading multiple words from the keyword.

To display the string on the screen we use a function **puts() .**

Syntax: **puts(str);**

Where str is a string variable containing a string value.

Source Code to Find the Frequency of Characters

```
#include <stdio.h>
 int main()
{
char s[100],ch;
int i,count=0;
printf("Enter a string: ");
gets(s);
printf("Enter a character to find frequency: ");
scanf("%c",&ch);
for(i=0;s[i]!='\0';++i)
{
if(ch==s[i])
++count;
}
printf("Frequency of %c = %d", ch, count);
return 0;
}
```

```
Enter a string:
This program is awesome.
Enter a character to find frequency:  s
Frequency of e = 3
```

```
/*  copy one string to another string */
#include<stdio.h>

int main()
{
 char str1[20],str2[20];
 printf("Enter string : ");
 gets(str1);
 for(i=0; str1[i]!=NULL; i++)
 {
   str2[i] =str1[i];
}
str2[i]=‘\0’;
 printf("\n str2=%s”,str2);

 return 0;
}
```

```
/*  reverse of  string */
#include<stdio.h>

int main()
{
 char str[20],rev[20];
int  i,  j, n=0;
 printf("Enter string : ");
 gets(str);
 for(i=0; str[i]!=‘\0’; i++)
              n++;
printf(“length of string=%d\n”,n);
for(i=0,j=n-1; str[i]!=‘\0’; i++,j--)
 {
   rev[i] =str[j];
}
rev[j]=‘\0’;
 printf("\n rev=%s”,rev);

 return 0;
}
```

//Write a program to count the number of letters, words, and lines in a given text.

```
#include<stdio.h>
void main() {
        int noc=0,nol=0,now=0,nos=0;
        char ch;
        printf(“enter string\n”);
        while((ch=getchar())!=EOF) {
               if(ch==‘ ‘)
               {
                              nos++;     now++;
               }
               else if(ch==‘\n’)
               {
                              nol++;     now++;
               }
               else
                              noc++;                  }
        printf("\n Number of character=%d",noc);
        printf("\n Number of line=%d",nol);
        printf("\n Number of words=%d",now);
        printf("\n Number of space=%d",nos);                         }
```

Write a c program to check given string is a palindrome or not

```
#include <stdio.h>
#include <string.h>
void main()
{
    char str[25];
    int i, n, f =1,j;
    printf("Enter a string \n");
    gets(str);
   n=strlen(str);
   printf("The length of the string %s  is %d\n", str, n);
        for (i=0,j=n-1; i<=n/2 ; i++,j--)
    {
                if (str[i] != str[j])
                  {  f = 0;
                      break;
                   }
    }
    if (f == 1)
       printf ("%s is a palindrome \n", str);
    else
       printf("%s is not a palindrome \n", str);
}
```

String Manipulation Functions

The C Library provides a rich set of string handling functions that are placed under the header file **<string.h> and <ctype.h>**.

Some of the string handling functions are (**string.h**):

strlen() strcat() strcpy() strrchr()
strcmp() strstr() strchr() strrev()

Some of the string conversion functions are (**ctype.h**):

toupper() tolower() toascii()

All I/O functions are available in **stdio.h**

scanf() printf() gets() puts()
getchar() putchar()

strlen () function:

This function counts and returns the number of characters in a string. It takes the form

Syantax: **int n=strlen(string);**

Where n is an integer variable, which receives the value of the length of the string. The counting ends at the first null character.

strcat () function:

The strcat function joins two strings together.

It takes of the following form:

strcat(string1,string2);

string1 and string2 are character arrays.

When the function strcat is executed, string2 is appended to string1.

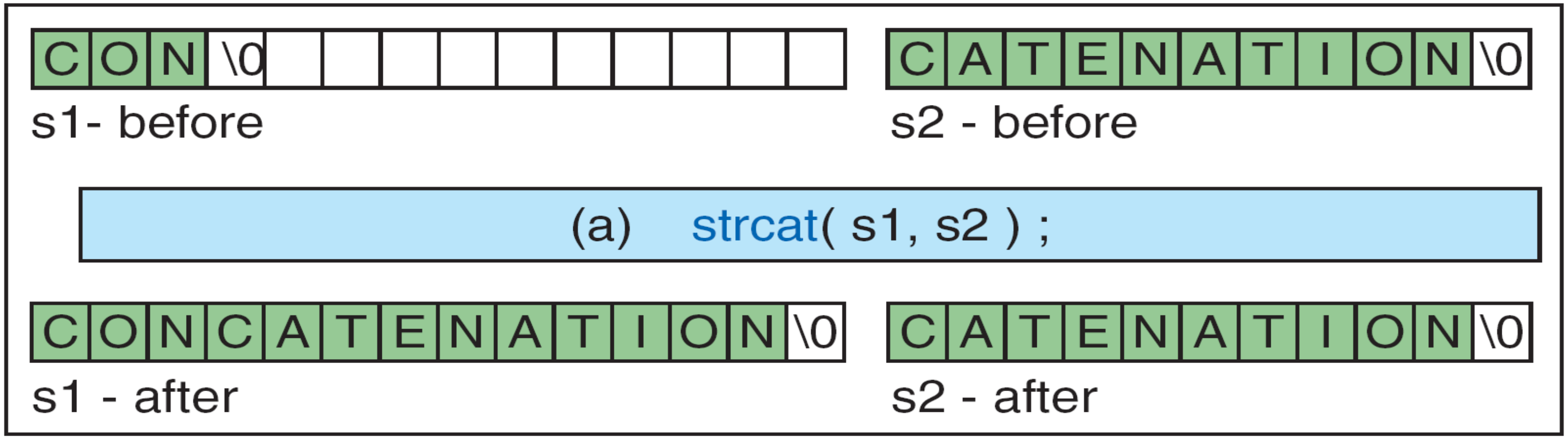
It does so by removing the null character at the end of string1 and placing string2 from there.

strcat function may also append a string constant to a string variable. The following is valid.

strcat(part1,"Good");

C permits nesting of strcat functions. Example:

strcat(strcat(string1,string2),string3);



C	O	N	\0										

s1- before

C	A	T	E	N	A	T	I	O	N	\0

s2 - before

(a) strcat(s1, s2) ;

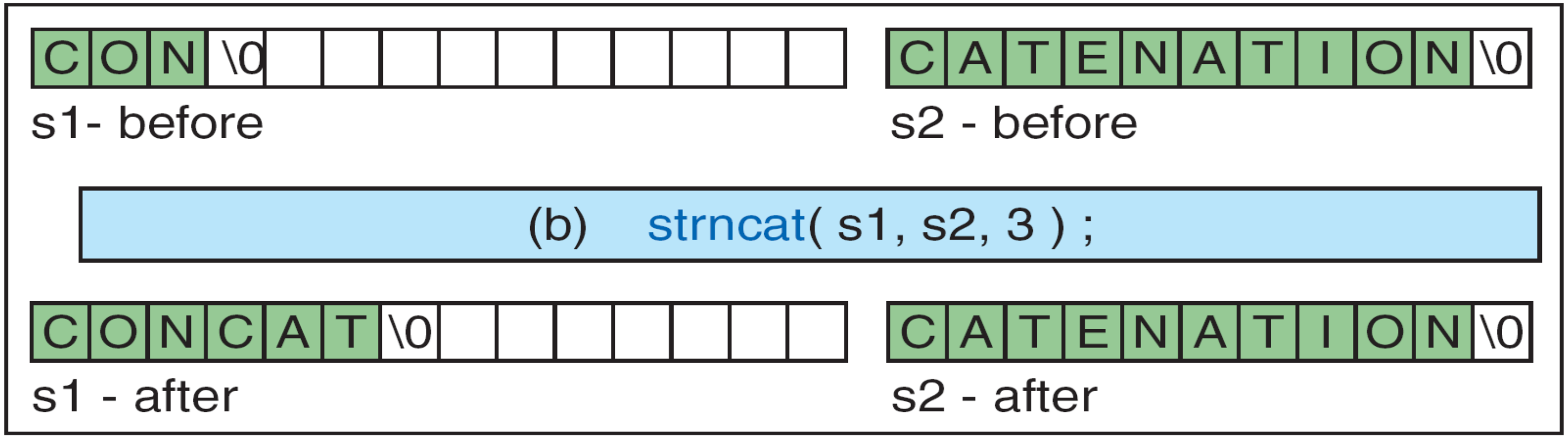
C	O	N	C	A	T	E	N	A	T	I	O	N	\0

s1 - after

C	A	T	E	N	A	T	I	O	N	\0

s2 - after

String Concatenate



C	O	N	\0										

s1- before

C	A	T	E	N	A	T	I	O	N	\0

s2 - before

(b) strncat(s1, s2, 3) ;

C	O	N	C	A	T	\0							

s1 - after

C	A	T	E	N	A	T	I	O	N	\0

s2 - after

String N Concatenate

String Concatenation

STRING HANDLING FUNCTIONS(Contd…)

strcmp () function:

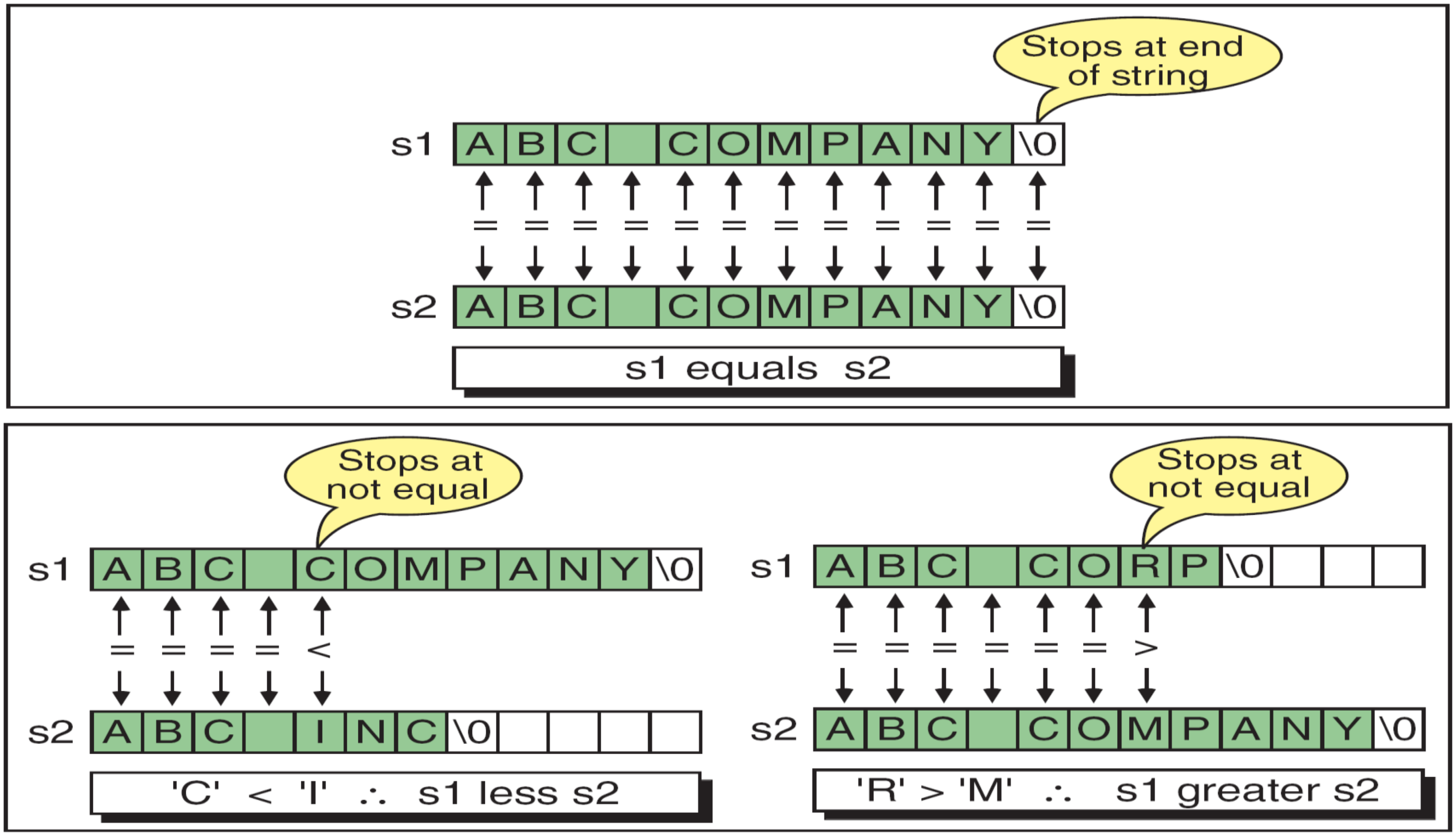
The strcmp function compares two strings, it returns the value 0 if they are equal.
If they are not equal, it returns the numeric difference between the first non matching characters in the strings.

It takes the following form:

strcmp(str1,str2);

returning value less than 0 means "str1" is less than "str2'‘
returning value 0 means "str1" is equal to "str2'‘
returning value greater than 0 means "str1" is greater than "str2"

string1 and string2 may be string variables or string constants.
Example: strcmp(name1,name2);
strcmp(name1,”John”);
strcmp(“their” ,”there”);



Stops at end of string

s1 A B C   C O M P A N Y \0

= = = = = = = = = = = =

s2 A B C   C O M P A N Y \0

s1 equals s2

Stops at not equal

s1 A B C   C O M P A N Y \0

= = = = <

s2 A B C   I N C \0

'C' < 'I' ∴ s1 less s2

Stops at not equal

s1 A B C   C O R P \0

= = = = = = >

s2 A B C   C O M P A N Y \0

'R' > 'M' ∴ s1 greater s2

strcmp (s1, s2)

String Compares

STRING HANDLING FUNCTIONS(Contd…)

strcpy () function: It copies the contents of one string to another string. It takes the following form:

strcpy(string1,string2);

The above function assign the contents of string2 to string1.
string2 may be a character array variable or a string constant. Example:

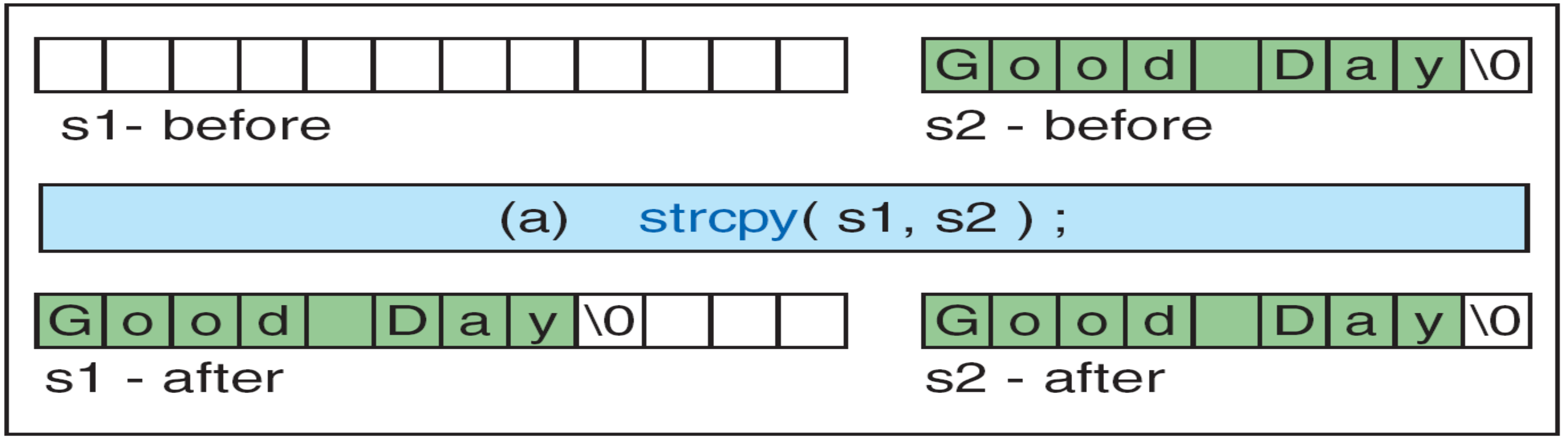
```
strcpy(city ,”Delhi”);
strcpy(city1,city2);
```

strrev() function: Reverses the contents of the string.
It takes of the form

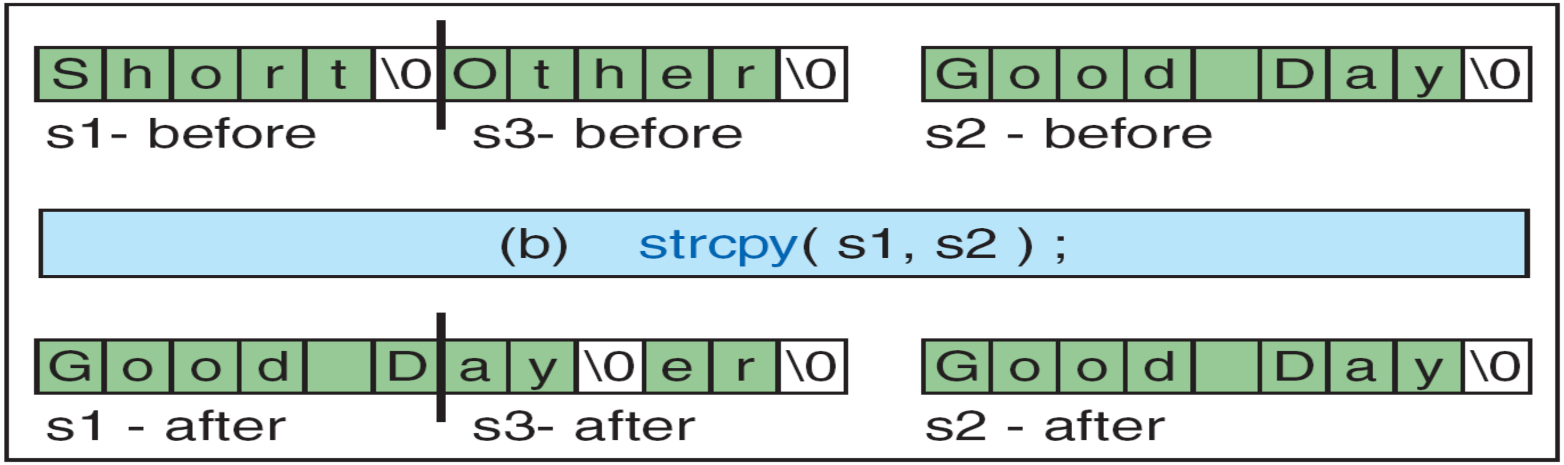
strrev(string);

Example:

```
#include<stdio.h>
#include<string.h>
void main(){
        char s[]=”hello”;
        strrev(s);
        puts(s);
}
```



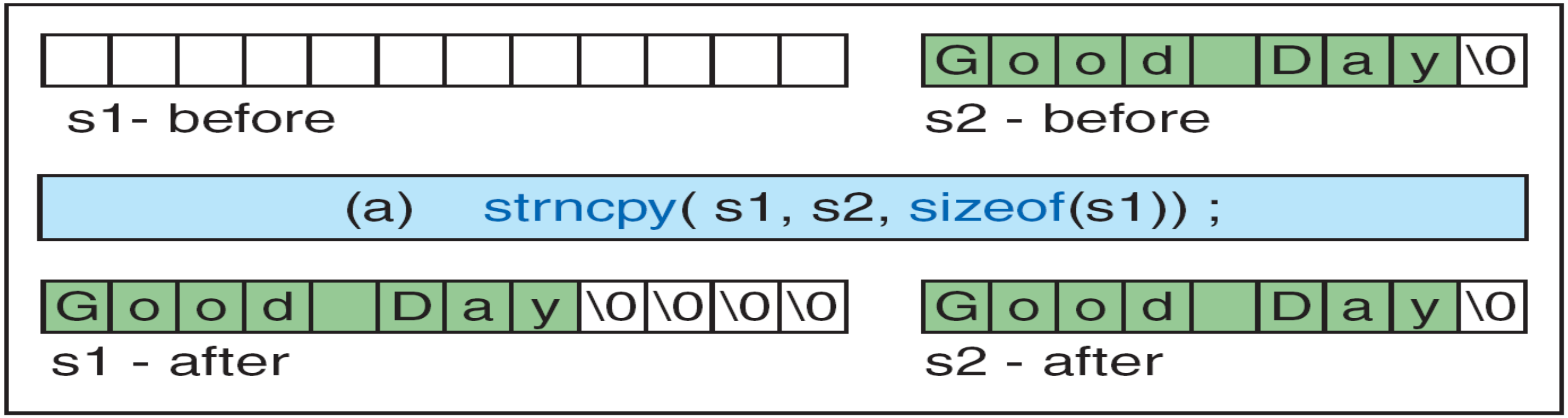
Copying Strings



Copying Long Strings

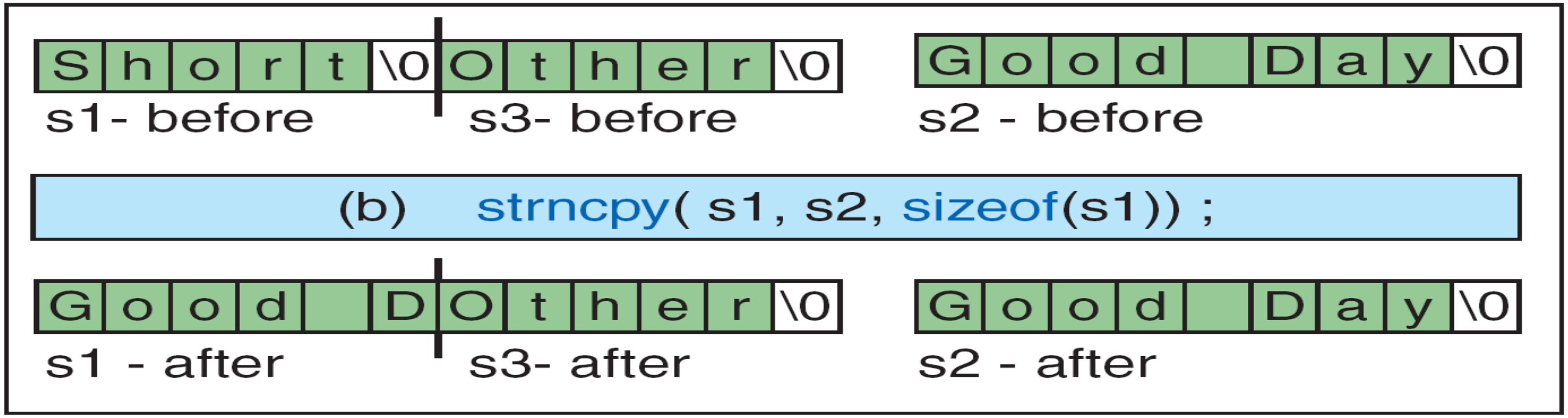
String Copy

Always use strncpy to copy one string to another.



```
strncpy( s1, s2, sizeof(s1)) ;
```

Copying Strings



```
strncpy( s1, s2, sizeof(s1)) ;
```

Copying Long Strings

STRING HANDLING FUNCTIONS(Contd…)

strstr () function:

It is a two-parameter function that can be used to locate a sub-string in a string. It takes the form:

strstr (s1, s2);

Example: strstr (s1,”ABC”);

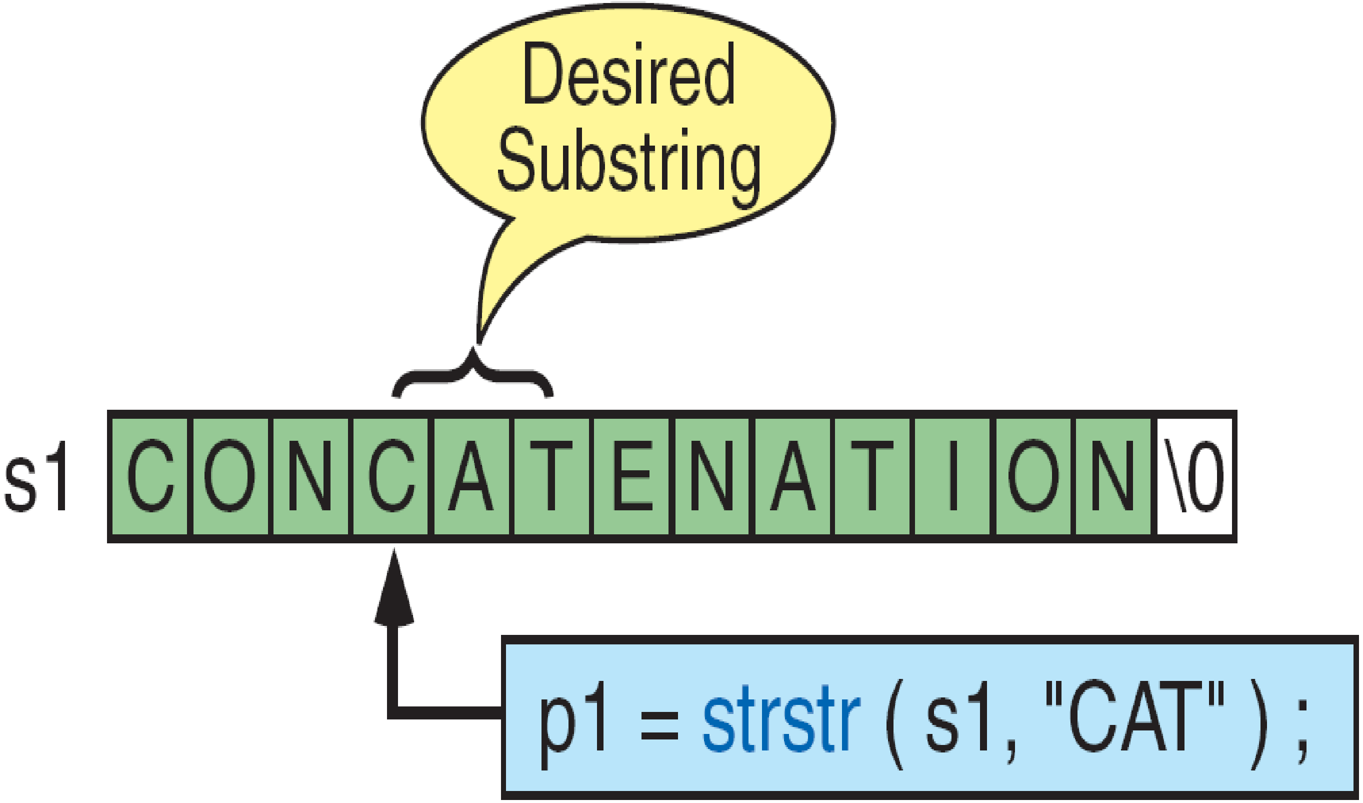
The function strstr searches the string s1 to see whether the string s2 is contained in s1.If yes, the function returns the address of the first occurrence of the sub-string. Otherwise, it returns a NULL pointer.

strchr() function:

It is used to determine the existence of a character in a string.

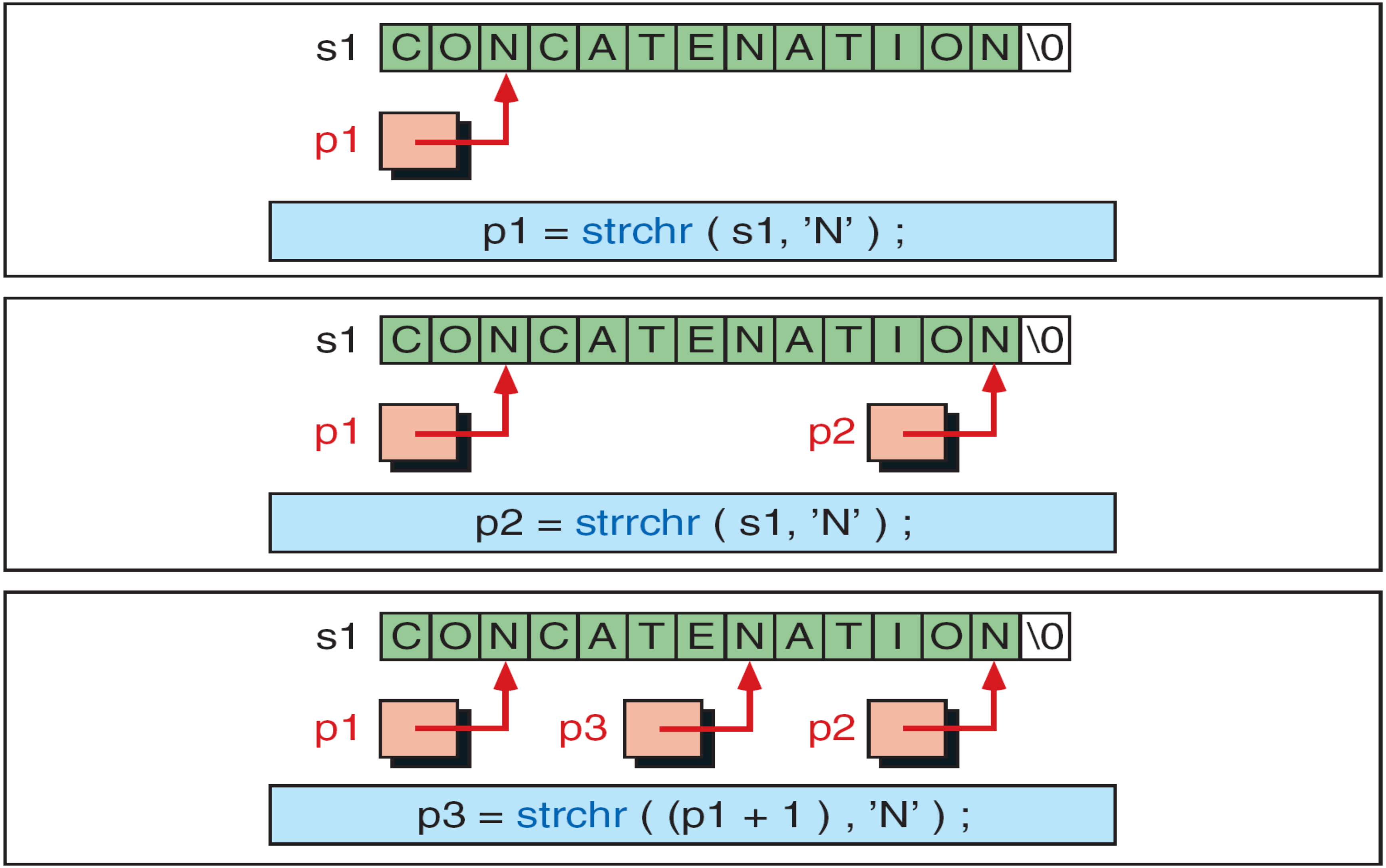
Example: **strchr (s1,’m’);** //It locates the first occurrence of the character ‘m’.

Example: **strrchr(s2,’m’);** //It locates the last occurrence of the character ‘m’.



```
p1 = strstr ( s1, "CAT" ) ;
```

String in String



```
p1 = strchr ( s1, ’N’ ) ;
```

```
p2 = strrchr ( s1, ’N’ ) ;
```

```
p3 = strchr ( (p1 + 1 ) , ’N’ ) ;
```

Character in String (*strchr*)

The basic string span function, strspn, searches the string, spanning characters that are in the set and stopping at the first character that is not in the set.

They return the number of characters that matched those in the set.

If no characters match those in the set, they return zero.

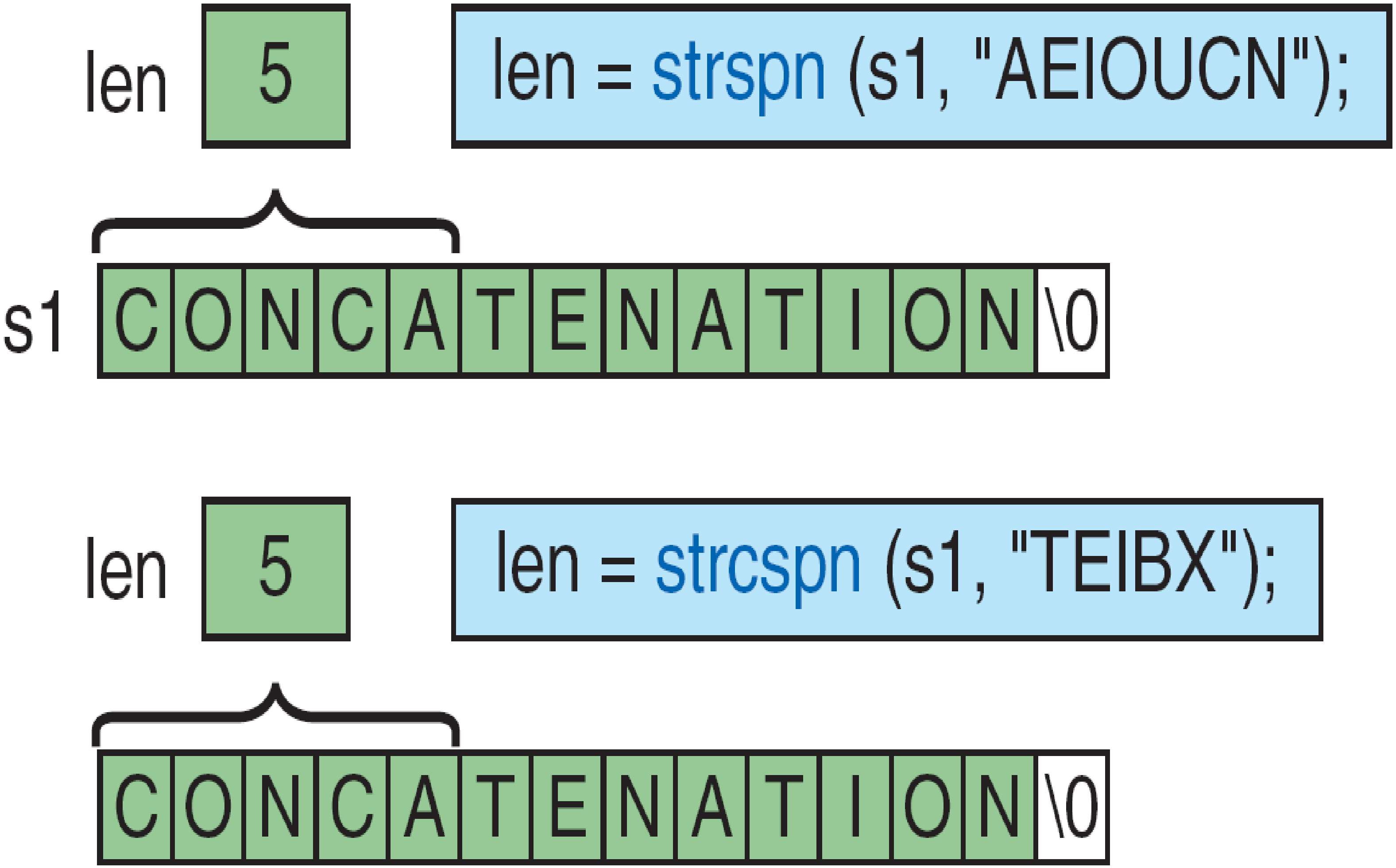
The function declaration is shown below:

int strspn(const char* str, const char* set);

The second function, strcspn, is string complement span; its functions stop at the first character that matches one of the characters in the set.

int strcspn(const char* str, const char* ste);

```
#include <stdio.h>
#include <string.h>
int main ()
 { int len;
 char str1[] = "ABCDEFG019874";
 char str2[] = "ABCD";
 len = strspn(str1, str2);
printf("Length of initial segment matching %d\n", len );
 return(0); }
```



```
len = strspn (s1, "AEIOUCN");
```

```
len = strcspn (s1, "TEIBX");
```

String Span

```
/*Define functions- length of a string, copy, concatenate, convert into uppercase letters, compare two strings for alphabetical order- over strings and implement in a program*/
#include<stdio.h>
 #include<string.h>
void main() {
    char str1[15],str2[15],str3[20];
    int n,c,len,i,cn;
   char *p;
    printf("\n Enter the string1 ");
    gets(str1);
    puts(str1);
    printf("\n Enter the string2 ");
    scanf("%s",str2);
    printf("%s",str2);
    printf("\nMENU");
    printf("\n 1. String Length  2. String Copy   3. String Concatenation");
    printf("4. String Comparison  5.Reverse of string  6 .Search substring  7.strspn");
    printf("\n Enter the choice u want to perform");
    scanf("%d",&n);
```

```
switch(n)
{
     case 1: len=strlen(str1);
             printf("\n The length of the string entered is %d",len);
             break;

     case 2: strcpy(str1,str2);
             printf("\n 1st string =%s,2nd string=%s\n",str1,str2);
             break;

     case 3:  strcat(str1,str2);
              printf("\n after concatenation string1=%s and string2=%s",str1,str2);
              break;

     case 4:  c=strcmp(str1,str2);
              if(c==0)
              printf("\n Both are  equal");
              else
              printf("\n Both are different");
              break;
case 5:   printf("\reverse string is: %s",strrev(str1));
          break;

case 6:   p=strstr(str1,”is”);

          if (p==NULL)
                    printf(“substing not found”);
          else
                    printf(“substring found at %d location”,p-str1);
          break;
case 7:   cn=strspn(str1,”aeioucn”)

           printf(“strspn=%d”,cn);

          cn =strcspn(str1,”teib”);

          printf(“strcspn=%d”,cn);

default:  printf("\n Enter correct choice");

          }

}
```

Arrays of Strings

Another common application of two dimensional arrays is to store an array of strings

A string is an array of characters; so, an array of strings is an array of arrays of characters

```
char names[MAX][SIZE];

int main()
{
        char colour[4][10] = {"Blue", "Red", "Orange", "Yellow"};
         for (int i = 0; i < 4; i++)
        puts(colour[i]);
         return 0;
}
```