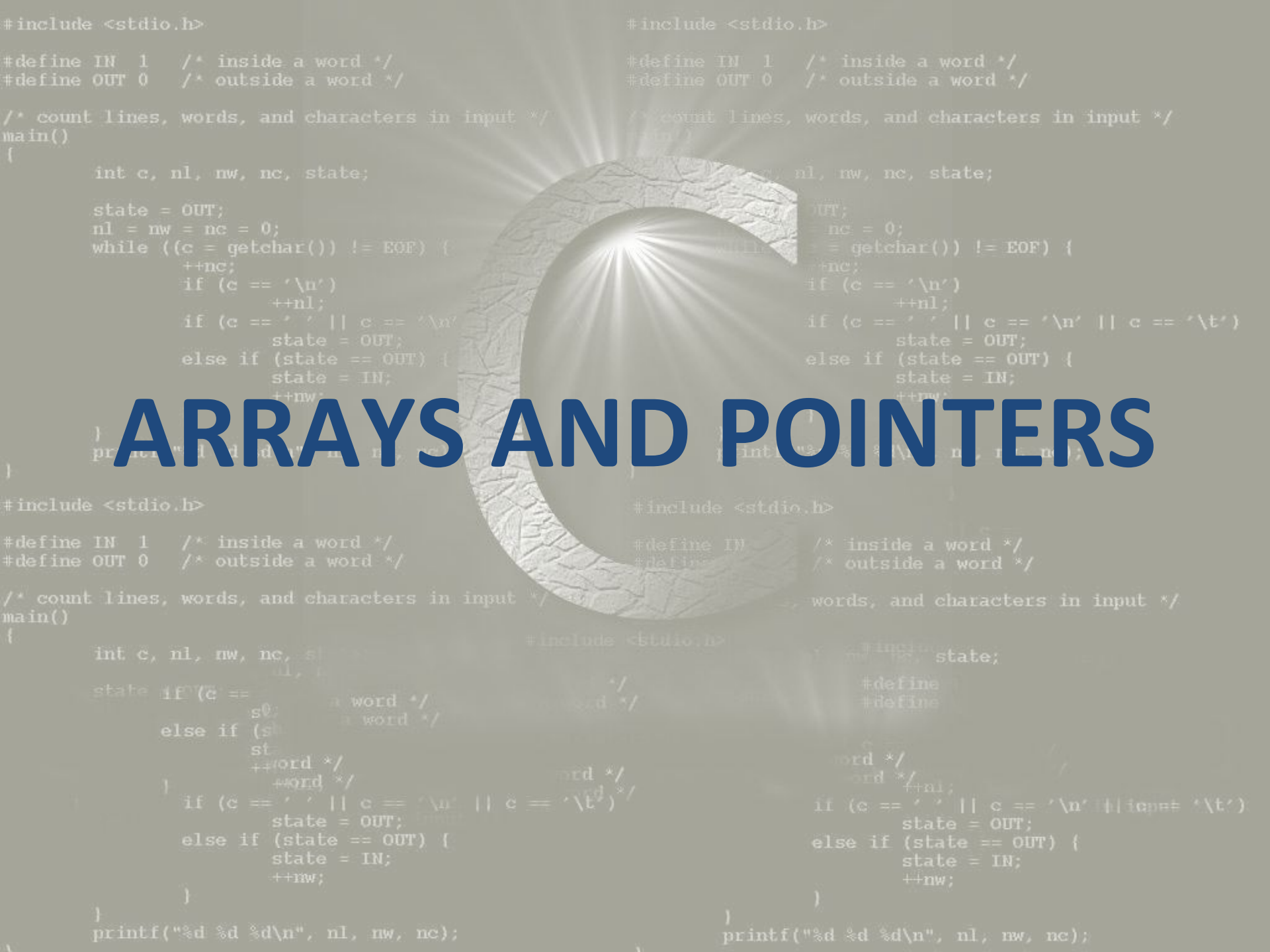



# ARRAYS AND POINTERS

# INTRODUCTION

Consider a problem to find the average of marks secured by five students in a course. The following piece of code is written for it:

```
//Comment: Average of marks secured by students
#include<stdio.h>
main()
{
int marks1=10, marks2=12, marks3=9, marks4=11,marks5=17;
int sum; float average;
sum=marks1+marks2+marks3+marks4+marks5;
average=sum/5.0;
printf("Average marks secured is %f ",average);
}
```

**NOW, SUPPOSE THERE ARE TWO HUNDRED STUDENTS IN A COURSE!**

Consider another real time problem that requires storing and processing names like ‘Sam’ entered by the user. There is no basic data type available in C that provides this flexibility.

- **The derived ARRAY type provides a solution to this problem. Array enables the user to store the characters of the entered name in a contiguous set of memory locations, all of which can be accessed by only one name i.e. array name.**

# ARRAYS

- An **array** is a data structure that is used for the storage of homogeneous data i.e. data of the same type. The figure shown below depicts arrays of four different types:
- Array1


[0] [1] [2] [3] [4] Indices or subscripts

(a) Character Array

- Array2


(b) Integer Array

- Array3


(c) Float Array

- Array4


(d) Array of user defined type

# Important Points about Arrays:

- An **array** is a collection of elements of the same data type. The data type of an element is called **element type**.
- **The individual elements of an array are not named.** All the elements of array share a common name i.e. the **array name**. For example in figure (a), all the elements of array i.e. ‘A’, ‘r’, ‘r’, ‘a’ and ‘y’ have a common name i.e. array1.
- **The individual elements of an array are distinguished and referred to or accessed according to their positions in an array.** The position of an element in an array is specified with an integer value known as **index** or **subscript**. Because arrays use indices or subscripts to access their elements, they are also known as **indexed variables** or **subscripted variables**.

- **The array index in C starts with 0** i.e. index of the first element of an array is 0.
- The memory space required by an array can be computed as **(size of element type)×(Number of elements in an array)**. For example array1 takes 1×5 i.e. 5 bytes in the memory, array2 takes 10 bytes (if an integer occupies 2 bytes), array3 takes 28 bytes and array4 takes 15 bytes (if an integer takes 2 bytes) in the memory.
- **Arrays are always stored in contiguous (i.e. continuous) memory locations.** For example in figure (a), if the first element of array1 is stored at memory location 2000, then the successive elements of the array will be stored at the memory locations 2001, 2002, 2003 and 2004.

# Classification of Arrays

1. Single Dimensional Arrays
2. Multi Dimensional Arrays

# 1.

- Single Dimensional Arrays

# Single Dimensional Arrays

- **Single-dimensional or one-dimensional array** consists of a fixed number of elements of the same data type organized as a simple linear sequence.
- The elements of a single-dimensional array can be accessed by using a single subscript, thus they are also known as **single-subscripted variables**. The other common names of single-dimensional arrays are **linear arrays** and **vectors**.
- For Example, all the arrays array1, array2, array3, array4 are single dimensional arrays.

# declaration of single-dimensional array

- The general form of **single-dimensional array declaration** is:

<storage_class_specifier><type_qualifier><type_modifier>

**Type identifier[**<size_specifier>**]**<=initialization_list>;

The following are the important points about single dimensional array declaration:

1. The terms enclosed within <> (i.e. angle brackets) are optional and the terms shown in **bold** are the **mandatory** parts of single-dimensional array declaration.

2. Single-dimensional array declaration consists of a type specifier (i.e. **element type**), an identifier (i.e. **name of array**) and a size specifier (i.e. **number of elements in the array**) enclosed within the square brackets (i.e. []).

The following declarations of single-dimensional arrays are valid:

- int array1[8];

//array1 is an array of 8 integers (Integer array)

- float array2[5];

//array2 is an array of 5floating point numbers(Floating point array)

- char array3[6];

//array3 is an array of 6 characters (Character array)

# 3. The size specifier specifies the number of elements in an array.

The following are the syntactic rules about the **size specifier:**

a) <u>It should be a compile time constant expression of integral type.</u>

**<u>Reasons:</u>**

i) The memory space to an array is allocated at the compile time.

ii) The size of array cannot be expanded or squeezed at the run-time.

- The following declarations of single-dimensional arrays are valid:

1. int array1[3+5];

//3+5 is a compile time constant expression of int type

2. float array2[size];

//where size is a qualified constant of integral type

3. char array3[size];

// where size is a symbolic constant of integral type

- The following declarations of single-dimensional arrays are not valid:

1. int array1[j];

//j is variable and not a constant

2. int array2[3.5];

//It is not possible to create an array of 3.5 locations

b) It should be greater than or equal to one.

**Reason:**

It is not possible to create an array of size zero

i.e. having no element.

- The following declarations of single-dimensional arrays are not valid:

1. int array1[-1];

//⍰ It is not possible to create an array of -1 locations

2. char array2[0];

//⍰ It is not possible to create an array of 0 locations

c) <u>The size specifier is mandatory if array is not explicitly initialized i.e. if initialization list is not present.</u>

**Reason:**

If initialization list is present, it is possible to determine the size of array from the number of initializers in the initialization list. In that case, the size specification becomes optional)

- The following declaration of single-dimensional array is not valid:

1. int array1[];

//Here, it is not possible to determine the size of array.

// Hence, the amount of memory to be allocated can

// not be determined.

# 4.Initializing elements of a single-dimensional array: Syntactic Rules

a) Comma separated list of initialzers called **Initialization list**

b) An **initializer** is an expression that determines the initial value of an element of the array.

c) If the **type** of initializers is not the same as the element type of an array, **implicit type casting** will be done, if the types are **compatible**. If types are **not compatible,** there will be a **compilation error**.

**The following program segment illustrates this fact:**

```
//Comment: Initializers of compatible but different types
#include<stdio.h>
main()
{
int arr1[]={2.3, 4.5, 6.9};
float arr2[]={‘A’,’B’,’C’};
printf(“Elements of arrays are initialized with\n”);
printf(“arr1: %d %d %d\n”,arr1[0],arr1[1],arr1[2]);
printf(“arr2: %f %f %f\n”,arr2[0],arr2[1],arr2[2]);
}
```

# Output Window:

```
Elements of arrays are initialized with
arr1: 2 4 6
arr2: 65.000000 66.000000 67.000000
```

## Remarks:

- The element types of the arrays are different from the types of initializers but the **types are compatible**.
- **Float initializers are demoted** and then elements of arr1 are initialized.
- **Char initializers are promoted** before initializing the elements of arr2. ASCII values of characters are used.

d) The number of initializers in the initialization list should be **less than or at most equal to the value of size specifier**, if it is present.

e) If the **number of initializers** in the initialization list is **less** than the value of size specifier, the **leading array locations** get initialized with the **values of initializers**. The **rest** of the array locations gets initialized to:

-> **0** (if it is an **integer** array),

->**0.0** (if case of **floating point** array)

->**'\0'**(i.e. null character if array is of **character** type).

The following declarations of single-dimensional arrays are valid:

1. int array1[]={1,2,3,4,5};

//Initialization list {1,2,3,4,5} present

2. int array2[]={2+3,a+5};

//Initializers are 2+3 and a+5, where a is an int variable

3. char array3[6]={'A','r','r','a','y'};

//Number of initializers is less than value of size specifier

The following declaration of single-dimensional array is not valid:

1. int array1[2]={1,2,3,4,5};

// Number of initializers cannot be more than the value of size specifier

# …when number of initialisers is less than the size of the array

Array1

| ‘A’ | ‘r’ | ‘r’ | ‘\0’ | ‘\0’ |
|---|---|---|---|---|

(a) char Array1[5]={‘A’,’r’,’r’};

Array2

| 1 | 5 | 8 | 12 | 7 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

(b) int Array2[8]={1,5,8,12,7};

Array3

| 1.2 | 5.1 | 8.3 | 12.9 | 7.5 | 0.0 | 0.0 | 0.0 |
|---|---|---|---|---|---|---|---|

(c) float Array3[8]={1.2,5.1,8.3,12.9,7.5};

# Usage of single dimensional array:

**The elements of a single-dimensional array can be accessed by using a subscript operator (i.e. []) and a subscript.**

**The following are the important points about the usage of single-dimensional arrays:**

- For accessing the elements of one-dimensional array, the **general form of expression is E1[E2]**, where E1 and E2 are sub-expressions and [] is subscript operator. **One of the sub-expressions** E1 or E2 must be of **array type or pointer type** and the **other sub-expression** must be of **integral type**.

- **The sub-expression of integral type (i.e. the subscript) must evaluate to a value greater than or equal to 0.**

- **The array subscript in C starts with 0 i.e. the subscript of the first element of an array is 0.** Thus, if the size of an array is n, the valid subscripts are from 0 to n-1.

However, if array index greater than n-1 is used while accessing an element of the array, there will be NO COMPILATION ERROR. This is due to the fact that C language does not provide compile-time or run-time **array index out-of-bound check**. However, using an out-of-bound index may lead to **RUN-TIME ERROR** or **EXCEPTIONS**. Thus, care must be taken to ensure that the array indices are within bounds i.e. from 0 to n-1.

# REMEMBER:

An **array type** is one of the derived data types. It is said to be derived from an element type and if element type is T (where T is a generic term and can be int, float, char or any other type), the array type is called "**array ofTs**". The construction of an array type from an element type is called **"array type derivation"**. Consider the declaration statement **int array[5];** the array type derived from an element type **int** is **int[5]**.

## This program snippet illustrates the use of subscript operator :

```
//Comment: Use of single-dimensional array
#include<stdio.h>
main()
{
int a[3]={10,20,30};
printf("Elements of array are:\n");
printf("%d %d %d",a[0],a[1],a[2]);
}
```

**Output Window:**

```
Element of array are:
10 20 30
```

**Remarks:**

- a is of array type.
- The expression a[0] refers to the first element, a[1] refers to the second element and a[2] refers to the third element of the array.

# reading, storing & accessing elements of 1-D array:

- An iteration statement (i.e. loop) is used for storing and reading the elements of one-dimensional array.
- The next program snippet illustrates the method to read, store and access the elements of single-dimensional array.

```
//Comment: Use of
   single-dimensional array
#include<stdio.h>
main()
{
int marks[200], lc, studs,
    sum=0;
float average;
printf("Enter the number of
    students in class\t");
scanf("%d",&studs);
printf("Enter marks of
    students\n\n");
for(lc=0;lc<studs;lc++)
{
printf("Enter marks of student
    %d\t",lc+1);
//Reading and storing
    elements in a 1-D array
scanf("%d",&marks[lc]);
}
for(lc=0;lc<studs;lc++)
//Accessing elements stored in
    the 1-D array
sum=sum+marks[lc];
average=(float)sum/studs;
printf("\nAverage marks of the
    class is %f",average);
}
```

## Output window:

```
Enter the number of students in class     5
Enter marks of students

Enter marks of student 1      10
Enter marks of student 2      12
Enter marks of student 3      9
Enter marks of student 4      11
Enter marks of student 5      17

Average marks of the class is 11.800000
```

## Remarks:

- **The elements of the array can be accessed in general way by writing marks[lc], where lc {0….199}.**
- Although at the runtime marks of only 5 students are entered, **the size of array is kept 200** to accommodate the worst case (i.e. 200 students).
- **195 locations are not used**. Hence, 195*2=390 bytes of memory got wasted.
- In line no. 19, integer variable sum is **explicitly type casted to float**.

# memory representation of single-dimensional array:

- The elements of array are **stored in contiguous** (i.e. continuous) memory locations.
- array1 char array1[]={'A','r','r','a','y'};


2000 2001 2002 2003 2004

- array2 int array2[]={1,5,8,12};


2000 2002 2004 2006

- array3 float array3[]={1.2,5.1,8.3,12.9};


2000 2004 2008 2012

# operations on single-dimensional array

- The only operation allowed on arrays is **subscripting**. Subscripting is an operation that selects an element from an array. To perform subscripting in C language, subscript operator (i.e. []) is used. The rules for subscripting have already been discussed.

# assigning an array to another array:

- A variable can be assigned to or initialized with another variable **but an array cannot be assigned to or initialized with another array**. The following statement is not valid and leads to a compilation error:
- array1=array2;

//where array1 and array2 are arrays of the same type and size.

**Reason:**

**In C language, the name of array refers to the address of the first element of array and is a constant object**. It does not have a modifiable l-value. Since it does not have a modifiable l-value, it cannot be placed on the left side of assignment operator.

# To assign an array to another array, each element must be assigned individually:

```
//Comment: Assignment of an
    array to another array
#include<stdio.h>
main()
{
int a[3], b[3]={10,20,30};
printf("Assigning an array to an
    array :\n");
a=b;
printf("Elements of array a
    are:\n");
printf("%d %d %d",a[0],a[1],a[2]);
}
```

**Output Window:**

Compilation error "L-value required in function main"

**Reason:**

- The name of the array a refers to the address of the array and is a constant object with a non-modifiable l-value. **Hence, it cannot be placed on the left side of assignment operator.**

**What to do?**

- Making use of a loop, assign individual elements of array b to the elements of array a by writing a[i]=b[i], where i{0,1,2}

# equating an array with another array:

- When the operands of an equality operator are of array type, it always evaluates to false.

**REASON:**

In C language, **the name of an array refers to the address of the first element of the array and the addresses of first elements of two arrays can never be the same**. Hence, when the operands of an equality operator are of array type, it always evaluates to false

# A program to illustrate the behavior of equality operator on arrays

```
//Comment: Equality operator & arrays
#include<stdio.h>
main()
{
int a[3]={10,20,30},
b[3]={10,20,30};
if(a==b)
   printf("Arrays are equal");
 else
   printf("Arrays are not equal");
}
```

**Memory contents**

(a)


2000 2002 2004

(b)


4000 4002 4004

## Output Window:

```
Arrays are not equal
```

## Reason:

- The name of arrays a and b refers to the addresses of their first elements i.e. 2000 and 4000 respectively.
- Since the addresses are different, the equality operator evaluates to false, although the contents of the arrays are same.

## What to do?

- For checking equality, check the equality of all individual elements.

# POINTERS

A **pointer** is a variable that holds the address of a variable or a function. Pointer is a powerful feature that adds enormous power and flexibility to C language.

A pointer variable can be declared as:

[storage_class_specifier][type_qualifier][type_modifier]

**type * identifier**[=l-value];

The important points about pointers:

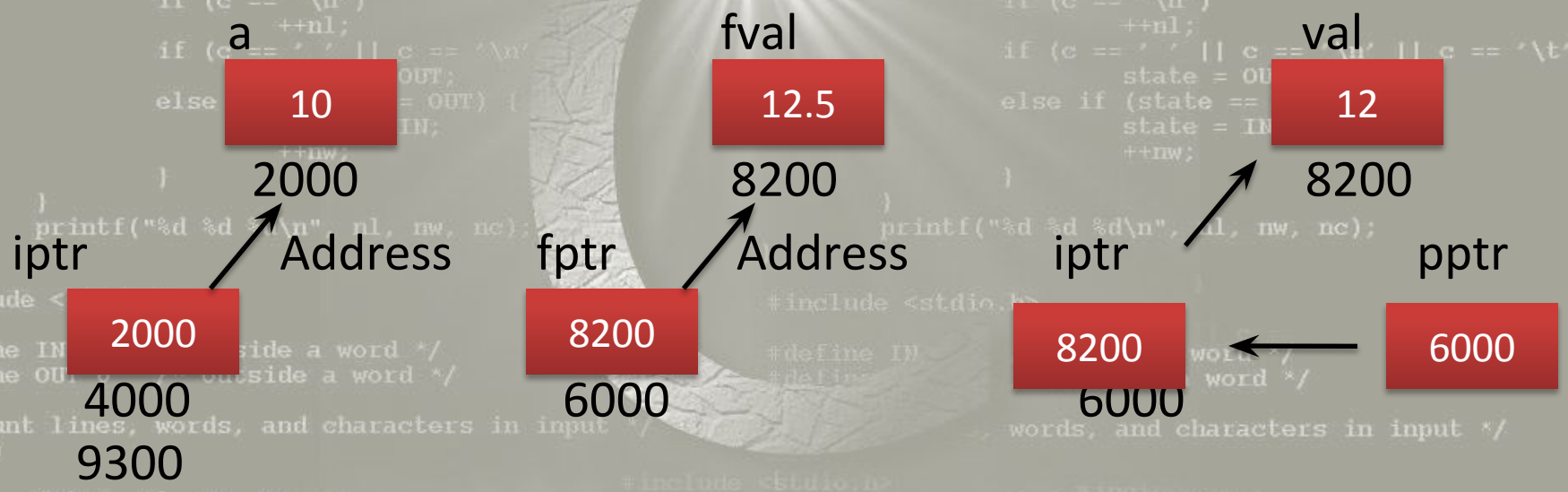
1. **The terms enclosed within [] (i.e. square brackets) are optional** and the terms shown in **bold** are the **mandatory** parts of pointer variable declaration.
2. A pointer variable declaration consists of a type specifier (i.e. **referenced type**), **punctuator *** and an identifier (i.e. **name of pointer variable**).

The following declarations are valid:

- int *iptr; //iptr is pointer to an integer
- float *fptr; //fptr is pointer to float
- char *cptr; //cptr is pointer to character
- const int *ptric //ptric is pointer to constant integer
- unsigned int *ptrui //ptrui is pointer to unsigned integer

3. **Pointer variable declarations are read from the right side.** The punctuator * is read as **“pointer to”**. So the declaration statement **int *iptr;** is read as **“iptr is a pointer to an integer”**.

4. **A pointer variable can only hold the address of a variable or a function .** In figure 4.5 a) iptr is an integer pointer and holds the address of an integer variable a. In figure 4.5 c) pptr is pointer to pointer to an integer and holds the address of an integer pointer iptr, which in turn holds the address of an integer variable val.




**int *ptr=&a**

**float fval=12.5;**

**float *fptr;**

**int val=10;**

**int *iptr=&val;**

**fptr=&fval;**

**int *pptr=&iptr;**

The concept of pointer declaration is scalable. It is possible to declare a pointer to a variable, which itself is a pointer variable. Such a pointer is known as **pointer to a pointer**. The declaration statement **int **pptr;** declares a pointer to a pointer and is read as **"pptr is a pointer to a pointer to an integer"**.

5. The value of pointer variable is printed with **%p format specifier**. Since the pointer variables hold addresses, which are unsigned integers, %u format specifier can also be used for printing pointer values. However, the use of %p format specifier is recommended over the use of %u specifier.

# 6. Every pointer variable takes same amount of memory space irrespective of whether it is a pointer to int, float, char or any other type.

```
//Size of pointer variables
#include<stdio.h>
main()
{
char *cptr;
int *iptr;
float *fptr;
printf("Pointer to character takes
    %d bytes\n",sizeof(cptr));
printf("Pointer to integer takes
    %d bytes\n",sizeof(iptr));
printf("Pointer to float takes %d
    bytes\n",sizeof(fptr));
}
```

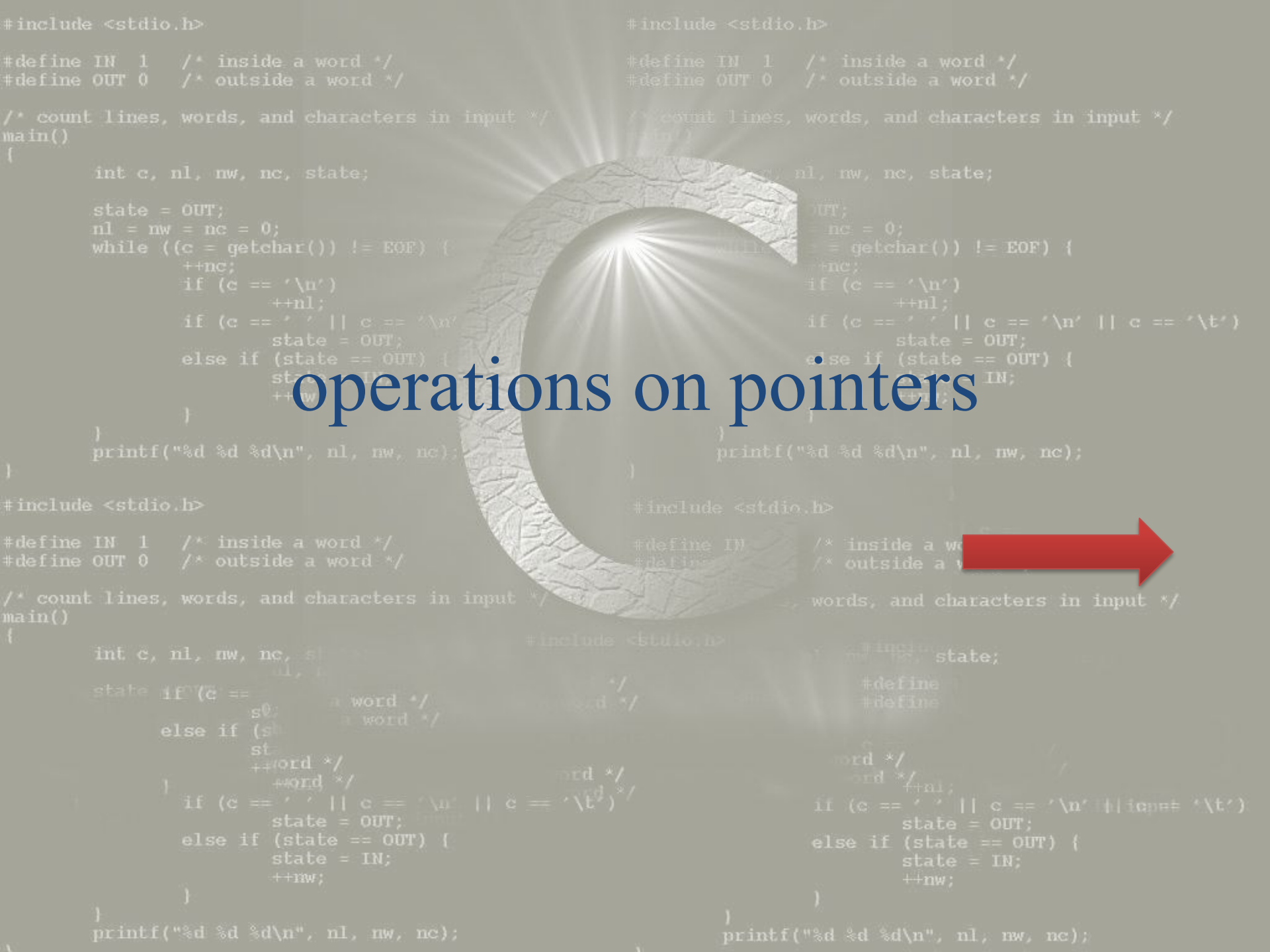
**Output Window:**

```
Pointer to character takes 2 bytes
Pointer to integer takes 2 bytes
Pointer to float takes 2 bytes
```

**Remarks:**

- The above output is result of execution using Borland Turbo C 3.0 IDE.
- In Borland Turbo C 4.5 or MS-VC++ 6.0 each type of pointer variable takes 4 bytes.

# operations on pointers



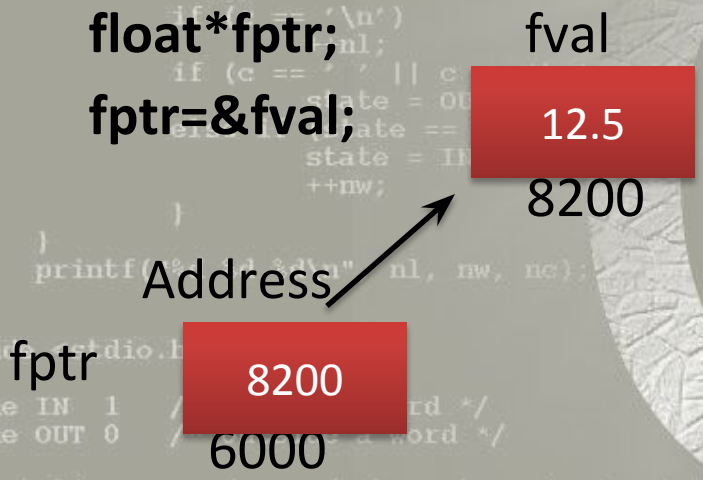
# referencing operation

In **referencing operation**, a pointer variable is made to refer to an object. The reference to an object can be created with the help of **reference operator (i.e. &).** The following are the important points about the reference operator:

- The reference operator i.e. **& is an unary operator** and should appear on the **left side of its operand.**
- The **operand** of reference operator should be a variable of **arithmetic type or pointer type** . The operand of reference operator can also be a function designator i.e. name of a function.
- The reference operator is also known as **address-of operator**.

# …A float pointer referencing a float variable

```
float fval=12.5;
float*fptr;
fptr=&fval;
```




```
//fval is a floating point variable
//initialized with 12.5
//fptr is a pointer to float type
// The address-of fval is assigned to
//fptr. fval is known as referenced
//object and fptr is known as
//referencing object and references
fval.
```

Integral and floating types are collectively called **arithmetic types**.
A **pointer type** describes an object, whose value provides reference to an object of type T. T is a generic term and will be known as **reference type**. It can be int, float, char or any other type. A pointer type derived from the reference type T is called **“pointer to T”**. The construction of pointer type is called **“pointer type derivation”**.

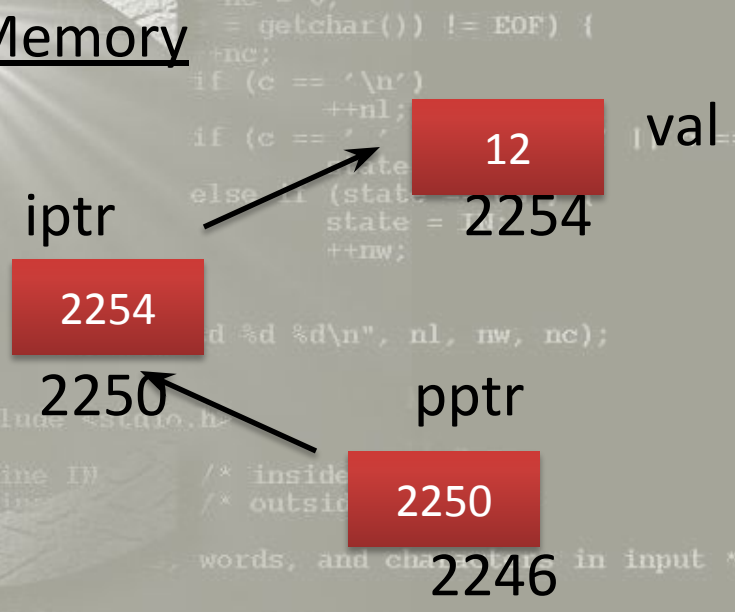
# Dereferencing a pointer:

**The object pointed to or referenced by a pointer can be indirectly accessed by dereferencing the pointer.** A dereferencing operation allows a pointer to be followed to the data object to which it points. **A pointer can be de-referenced by using a dereference operator (i.e. *).** The following are the important points about the dereference operator:

- The dereference operator (i.e. *) is a **unary operator** and should appear on the **left side of its operand**.
- The **operand** of dereference operator should be of **pointer type**.
- The dereference operator is also known as **indirection operator** or **value-at operator**.

# ...a program to illustrate dereferencing operation

```
//Comment: Dereferencing pointers
#include<stdio.h>
main()
{    int val=12;
     int *iptr=&val;
     int **pptr=&iptr;
     printf("Value is %d\n",val);
     printf("Value by dereferencing
     iptr is %d\n",*iptr);
     printf("Value by dereferencing
     pptr is %d\n",**pptr);
     printf("Value of iptr is
     %p\n",iptr);
     printf("value of pptr is
     %p\n",pptr); }
```



## Output Window:

```
Value is 12
Value by dereferencing iptr is 12
Value by dereferencing pptr is 12
Value of iptr is 2407:2254
Value of pptr is 2407:2250
```

## Remarks:

- The printed addresses are in the form of **segment address: offset address**.
- The segment address and the offset address are in **hexadecimal number system.**
- If the memory is assumed to be analogous a city, segment address is analogous to a sector number and the offset address is analogous to a house number.
- The addresses that you get in the output may be different from the mentioned addresses as the memory allocation is purely random.
- **val=12, iptr=2254 and pptr=2250**
- ***iptr=value-at(iptr)=value-at(2254)=12**
- ****pptr=value-at(value-at(pptr))=value-at(value-at(2250))=value-at(2254)=12**

# assigning to a pointer

A pointer can be assigned or initialized with the address of an object. A pointer variable **cannot hold non-address value** and thus can only be assigned or initialized with l-values.

**A program to illustrate that a pointer variable cannot hold a non-address value:**

| // **Invalid assignment to pointer variable**<br>#include<stdio.h><br>main()<br>{<br>int val=10;<br>int *ptr=val;<br>printf("Value of variable is %d\n",val);<br>printf("Pointer holds %p\n", ptr);<br>} | **Memory Contents**<br>  <br> | **Output Window**<br>Compilation error "Cannot convert int to int*"<br>**Reason:**<br>Pointer variables can only hold addresses.<br>A pointer variable ptr cannot hold an integer value val.<br>**What to do?**<br>Initialize ptr with the address of variable val, by writing&val |
|---|---|---|

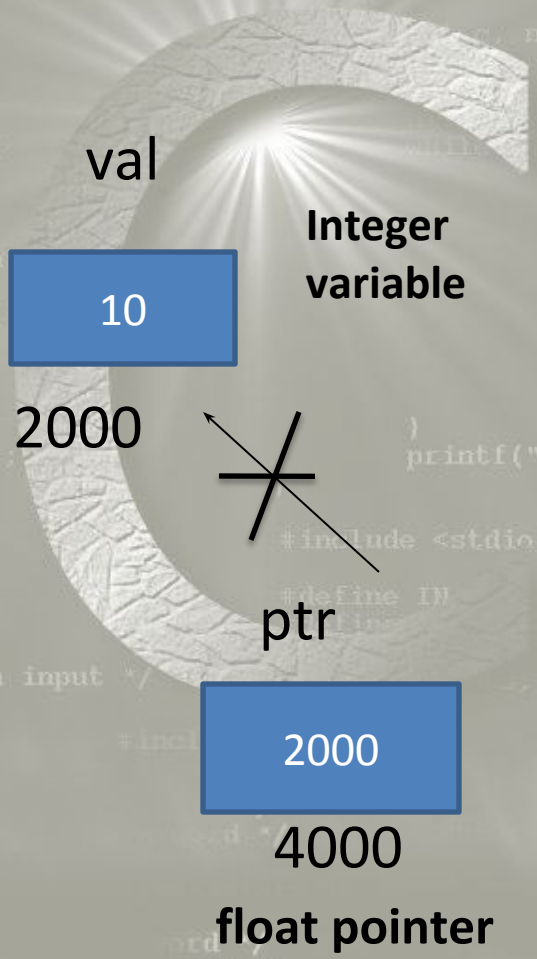
1. **There is an exception to this rule**. The constant zero can be assigned to a pointer. **For example int *iptr=0;** is valid. Assignment or initialization with zero makes pointer a special pointer known as **null pointer**.
2. **A pointer to a type cannot be initialized or assigned the address of an object of another type.**
3. **A pointer can be assigned or initialized with another pointer of the same type**. However, it is not possible to assign a pointer of one type to a pointer of another type without explicit type casting.

There is an exception to rule 2 and 3. A pointer to any type of object can be assigned to a pointer of type void but vice-versa is not true. A **void pointer** cannot be assigned to pointer to a type without explicit type casting.

**// Invalid assignment to pointer variable**

```
#include<stdio.h>
main()
{
 int val=10;
 float *ptr=&val;
 printf("Value of variable is %d\n",val);
 printf("Pointer holds %p\n", ptr);
}
```

**Memory contents**




**Output window**

Compilation error "Cannot convert int* to float*"

**Reason:**

**Pointer variables can only be assigned address of the object of the same type.**

A pointer variable ptr (of type float*) cannot hold the address of an integer variable (i.e. int*).

**What can be done?**

Explicitly type cast int* to float* by using type cast operator. Write **float*ptr=(float*)&val;**

**Remarks:**

Explicit type casting may give unexpected results and is not recommended.

# arithmetic operations (pointer arithmetic)

1. • Addition operation
2. • Increment operation
3. • Subtraction operation
4. • Decrement operation
5. • Relational (comparison) operations

# addition operation

1. **An expression of integer type can be added to an expression of pointer type.** The result of such operation would have the same type as pointer type operand. If ptr is a pointer to an object, then “adding 1 to pointer” (i.e. ptr+1) points to the next object. Similarly, ptr+i would point to the $i^{th}$ object beyond the one, the ptr currently points to.

2. **Addition of two pointers is not allowed.**

3. The addition of a pointer and an integer is **commutative** i.e. ptr+1 is same as 1+ptr.

Addition operation on pointers…

| Sr. No. | operator | Type of operand 1 | Type of operand 2 | Resultant type | e.g | Initial value | Final value | How to determine ? |
|---|---|---|---|---|---|---|---|---|
| **1.** | Addition operator (+) | Pointer to type T | **Int** | Pointer to type T | | | | Result = initial value of pointer +integer operand*size of ( the reference type T) |
| | Example1: | float* | Int | float* | ptr=ptr+1 | 2000 | 2004 | 2000+1*(4)= 2004 as sizeof(float)= 4 |
| | Example2: | int* | Int | int* | ptr=ptr+5 | 2000 | 2010 | 2000+5*(2)= 2010, if sizeof(int)=2 |
| **2.** | Addition operator (+) | Pointer | **Pointer** | **Not Allowed** | | | | |

# increment operation

- The increment operator can be applied to an operand of pointer type.

| SNo. | Operator | Type of operand | Resultant type | e.g. | Initial values | Final values |
|---|---|---|---|---|---|---|
| 1. | Increment operator (++) | Pointer to type T | Pointer to type T | | | |
| Example1: | Post-increment | float* | float* | ftr=ptr++ | ftr=? ptr=2000 | ftr=2000 ptr=2004 |
| Example2: | Pre-increment | float* | float* | ftr=++ptr | ftr=? ptr=2000 | ftr=2004 ptr=2004 |

# REMEMBER:

**Post- increment:**

Result=initial value of pointer

**Pre-increment:**

Result = initial value of pointer+ sizeof (the reference type T)

**In both the cases:**

Value of pointer=Value of pointer + sizeof (the reference type T)

# subtraction operation

1. **A pointer and an integer can be subtracted.**

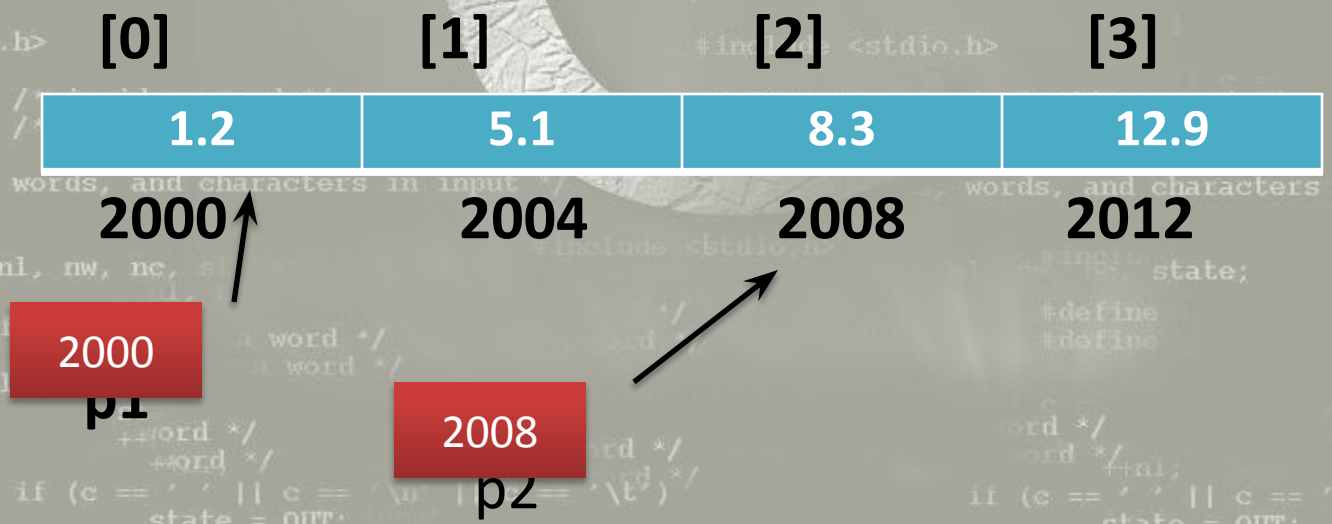
| SNo. | Operator | Type of operand 1 | Type of operand 2 | Resultant type | e.g. | Initial value(s) | Final value | How to determine? |
|---|---|---|---|---|---|---|---|---|
| 1. | Subtraction operator (-) | Pointer to type T | **int** | Pointer to type T | | | | **Result = initial value of pointer - integer operand*sizeof ( the reference type T)** |
| | **Example:** | float* | int | float* | ptr= ptr-1 | ptr= 2000 | 1996 | 2000-1*(4) =1996 as sizeof(float)=4 |

2. Subtraction of integer and pointer is **not commutative** i.e. ptr-1 is not same as 1-ptr. The operation 1-ptr is illegal.

3. **Two pointers can also be subtracted**. Pointer subtraction is meaningful only if both the pointers point to the elements of the same array. The result of the operation is the difference of subscripts of two array elements.

**Pointer subtracted from a pointer:**

**Array** float array3[]={1.2,5.1,8.3,12.9};




**p1-p2=2 i.e. the difference between the subscripts**

| SNo. | Operator | Type of operand 1 | Type of operand 2 | Resultant type | e.g. | Initial value(s) | Final value | How to determine? |
|---|---|---|---|---|---|---|---|---|
| 1. | Subtraction operator (-) | Pointer to type T | **Pointer to type T** | int | | | | **Result= (operand1-operand2)/ sizeof (the reference type T)** |
| | **Example:** | float* | float* | int | a=p2-p1 | p1=2000 p2=2008 | 2 | (2008-2000) / sizeof (float) (2008-2000) /4=2 |

# decrement operation

The decrement operator can be applied to an operand of **pointer type.**

| SNo. | Operator | Type of operand | Resultant type | e.g. | Initial values | Final values |
|---|---|---|---|---|---|---|
| 1. | Decrement operator (--) | **Pointer to type T** | Pointer to type T | | | |
| Example1: | Post-decrement | float* | float* | ftr=ptr-- | ftr=? ptr=2000 | ftr=2000 ptr=1996 |
| Example2: | Pre-decrement | float* | float* | ftr=--ptr | ftr=? ptr=2000 | ftr=1996 ptr=1996 |

# REMEMBER:

**Post- decrement:**

Result=initial value of pointer

**Pre-decrement:**

Result =initial value of pointer-sizeof (the reference type T)

**In both the cases:**

Value of pointer=Value of pointer -sizeof (the reference type T)

# relational (comparison) operations

- A pointer can be compared with a pointer of the same type or with zero. **Comparison of pointers is meaningful only when they point to the elements of the same array.**

| Operator | Type of operand 1 | Type of operand 2 | Resultant type | e.g. | Initial values | Final value |
|---|---|---|---|---|---|---|
| Operators (==, !=, <, <=, >, >=) | Pointer to type T | Pointer to type T | int (0 i.e. false or 1 i.e. true) | | | |
| Example1: | float* | float* | int | r=p1!=p2 | p1=2000 p2=2008 | 1 |
| Example2: | float* | float* | int | r=p1<p2 | p1=2000 p2=2008 | 1 |
| Example3: | float* | float* | int | r=p2>=p1 | p1=2000 p2=2008 | 1 |
| Example4: | float* | float* | int | r=p2==p1 | p1=2000 p2=2008 | 0 |

# illegal pointer operations

**The following operations on pointers are not allowed:**

- **Addition of two pointers** is not allowed.
- **Only integers can be added to pointers**. It is not valid to add float or double to a pointer.
- **Multiplication and division operators cannot be applied** on pointers.
- **Bitwise operators cannot be applied** on pointers.
- A pointer of one type cannot be assigned to a pointer of another type (except void*).
- A pointer variable cannot be assigned a non-address value (except zero).

The pointer arithmetic discussed above is not applicable on void pointers.

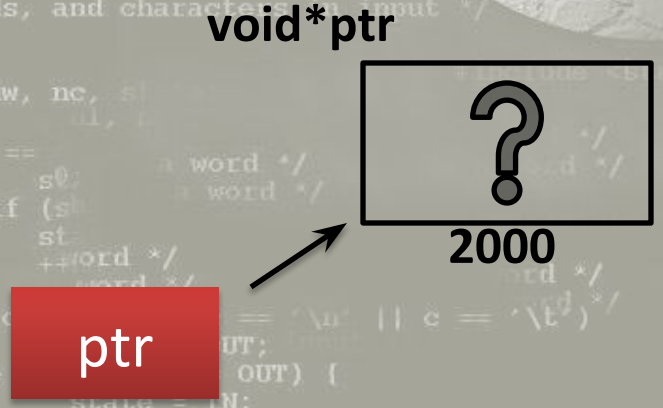
But what are actually VOID POINTERS?

# void pointer

- **void is one of the basic data types available in C language.** void means nothing or not known. **It is not possible to create an object of type void**. For example the following declaration statement is not valid and leads to "Size of var unknown or zero" compilation error:

```
void var;
```

- Although an object of type void cannot be created, it is **possible to create a pointer to void**. Such a pointer is known as **void pointer** and has type void*. **Void pointer** is a generic pointer and can point to any type of object




**void pointer can point to any type of data. The type of data inside the block can be char, int, float or any other type.**

# operations on void pointer

**The following operations on void pointer are allowed:**

1. **A pointer to any type of object can be assigned to a void pointer.** This is a standard conversion and compiler will do it implicitly without any explicit type casting.

2. **void pointers can be compared for equality and inequality.**

## …a program to illustrate that pointer to any type implicitly gets converted to void*

```
//Assigning a pointer to a void pointer
#include<stdio.h>
main()
{
int a=10;
int *iptr=&a;
void *vptr=iptr;
printf("int* is implicitly converted to void*");
}
```

**Output Window:**

```
int* is implicitly converted to void*
```

**Remarks:**

- iptr is of int* type
- vptr is of void* type
- iptr implicitly gets converted to vptr in line no. 5

# operations NOT ALLOWED ON void pointers

1. **A void pointer cannot be dereferenced.**
2. **Pointer arithmetic is not allowed** on void pointers.

**Reason:**

void pointer cannot be dereferenced and pointer arithmetic is not applied on it because the compiler does not know what kind of object the void pointer is really pointing to. Hence, the precise number of bytes to which the pointer refers to is not known.

- Before the application of dereference operator or arithmetic operator on a void pointer, it must be explicitly type casted to a pointer to a specific type.

# Null pointer

1. A **null pointer** is a special pointer that **does not point anywhere**. It does not hold the address of any object or a function. It has numeric value 0. The following declaration statement declares nptr as null pointer:

   **int *nptr=0;**

2. **The macro or symbolic constant NULL defined in <stdio.h>, <stddef.h> and other header files can also be used for the creation of a null pointer.** The following declaration statement is equivalent to the declaration statement mentioned above:

   **int *nptr=NULL;**

# Important points about null pointers

- When a null pointer is compared with a pointer to any object or a function, the result of comparison is always false.
- **Two null pointers always compare equal**.
- **Dereferencing a null pointer leads to a runtime error**.

# RELATION BETWEEN

# 1. The name of an array refers to the address of the first element of the array i.e. an expression of array type decomposes to pointer type.

```
//Arrays and pointers //relationship-I

#include<stdio.h>
main()
{
int arr[3]={10,15,20};
printf("First element of array is at %p\n",arr);
printf("Second element of array is at
%p\n",arr+1);
printf("Third element of array is at
%p\n",arr+2);
}
```

**Output Window:**

```
First element of array is at 24D7:2242
Second element of array is at
24D7:2244
Third element of array is at 24D7:2246
```

**Remarks:**

- The name of the array (i.e.arr) refers to the address of the first element of the array and is a constant object.
- The expression arr+1 decomposes to pointer type.
- Thus, in expression arr+1, the arithmetic involved is pointer arithmetic.

**Note that ++arr cannot be written instead of arr+1 as array is a constant object**.

- **The name of the array refers to the address of the first element of the array but there are two exceptions to this rule:**

a) When **array name is operand of sizeof operator** it does not decompose to the address of its first element.

```
//sizeof operator and arrays

#include<stdio.h>
main()
{
 int array[5]={10,15,20,25,30};
 printf("The result of sizeof
operator is %d\n",sizeof(array));
}
```

**Output Window**

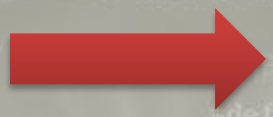
```
The result of sizeof operator is 10
```

**Remarks:**

- The result of sizeof operator is the size of the complete array (i.e. 5 elements * 2 bytes each = 10 bytes).
- This example clearly indicates that name of array is not decomposed into pointer type.
- **If it would have been decomposed into pointer type, the result would have been 2 as integer pointer takes 2 bytes in the memory.**

b) When the **array name is an operand of reference or address-of operator** it does not decompose to the address of its first element.

2. In C language, any operation that involves array subscripting is done by using pointers. The expression of form E1[E2] is automatically converted into an equivalent expression of form *(E1+E2).



## //Arrays and pointers relationship-II

```
#include<stdio.h>
main()
{
int array[3]={10,15,20};
printf("Elements are %d %d %d\n",array[0], array[1],array[2] );
printf("Elements are %d %d%d\n",*(array+0),*(array+1),*(array+2));
printf("Elements are %d %d %d\n",0[array],1[array],2[array]);
}
```

## Output window

```
Elements are 10 15 20
Elements are 10 15 20
Elements are 10 15 20
```

## Remarks:

- E1[E2] is the usual way of subscripting (used in line no. 6).
- **E1[E2] gets converted to *(E1+E2)**. The transformed way of subscripting is used in line no. 7.
- 0[array] used in line no. 8 is also valid because 0[array] will automatically be converted to *(0+array), which is equivalent to *(array+0), + being a commutative operation.
- *(array+0) is equivalent to array[0]. Hence, 0[array] is equivalent to array[0].

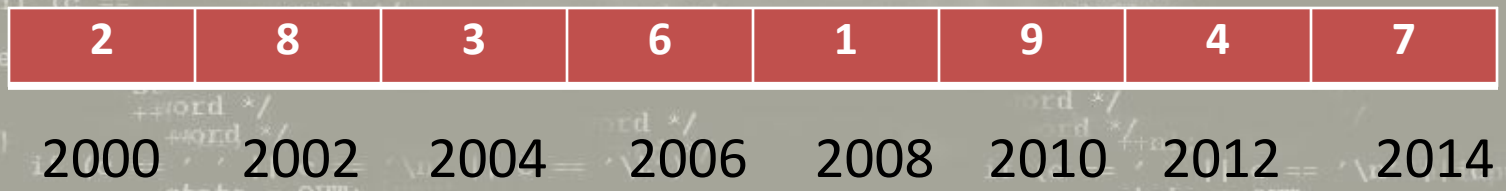
# column major order of storage

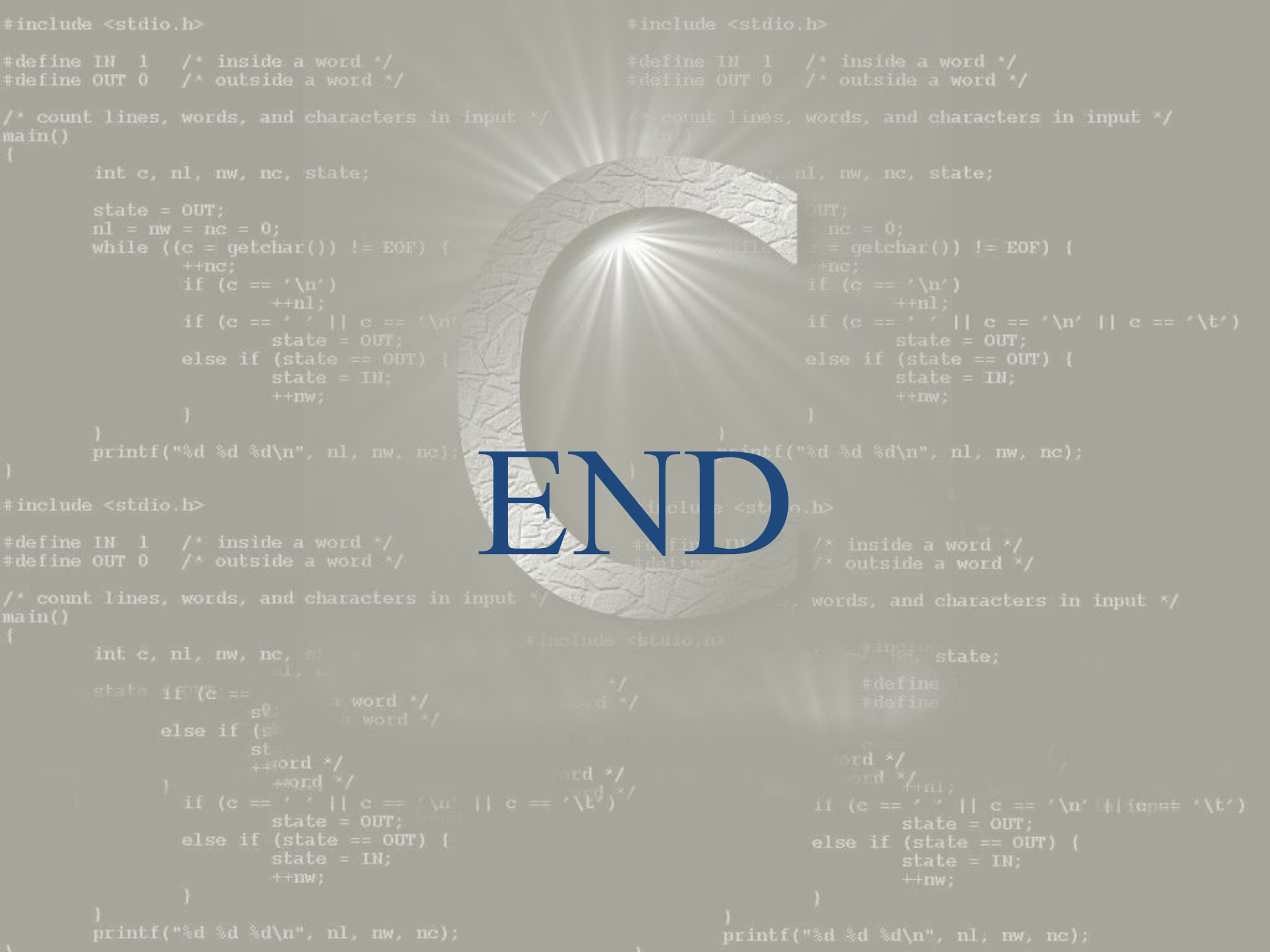
- In **column major order of storage**, the elements of an array are stored column-wise. Column major order of array storage is used in the languages like FORTRAN, MATLAB etc.

E.g.

| | | | |
|---|---|---|---|
| 2 | 3 | 1 | 4 |
| 8 | 6 | 9 | 7 |

will be stored as




END