Pointers
&
Dynamic Memory Allocation

What is pointer?

- Pointer is a **derived data type**
- It is built from one of the fundamental data types available in c
- **Pointer** is a variable that used to store address of another variable
- It is **location** in the computer memory where data is stored
- Pointer can be used to **access** and **manipulate** the data stored in memory

Benefits of pointers

- Pointer are more **efficient** in handling arrays
- Pointers can be used to return **multiple values** from **functions**
- The use of pointer arrays to character strings results in **saving of data storage space** in memory
- It allow to support **dynamic** memory management **(malloc() and calloc() available in <alloc.h>)**
- It is provide efficient way to manipulate **link list, structure , queue , stacks and trees**
- It **reduce** execution time

- **Memory** is collection of storage cell
- Each cell commonly known as **byte** and has **address** associated with it
- Addresses are **number** consecutively
- Whenever variable is declare the system allocates memory at appropriate location to hold the value

e.g int x = 100;

x → variable

100 → value

5000 → address

Declaring pointer variables

- **data_type * pointername;**
- ***** Indicate the variable is pointer variable
- **pointername** needs a memory location

- **pointername** points to variable of type data_type.

  int *p or float *y;

p points to the integer datatype or int variable

y points to the floating point data type

Remark: Any pointer variable occupies 2 Bytes

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int *p;
        float *q;
        double *r;
        char *x;
        printf("\n the size of integer pointer is %d", sizeof(p));
        printf("\n the size of float pointer is %d", sizeof(q));
        printf("\n the size of double pointer is %d", sizeof(r));
        printf("\n the size of character pointer is %d", sizeof(char *));
//      printf("\n the size of character pointer is %d", sizeof(x));
        return 0;
        getch();
}
/* the size of integer pointer is 2
 the size of float pointer is 2
 the size of double pointer is 2
 the size of character pointer is 2
*/
```

- During the execution **variable name x** always associate with address **5000**
- Since the **memory** is simply **number** so they can be assigned to some variables
- That can be stored in memory like other **variable** → called **pointer variable**

Variable	value	address
x	100	5000
p	5000	5048

P is **pointer type variable** and **Value of p** is **address of variable x**

Accessing the address of a variable

- The address of variable is determine by **operator &**
- It is used in **scanf (“%d”, &x);**
- **&** immediately preceding a variable returns the address of the variable returns the address of the variable associated with it

  p = &x;

  address of x is assigned to p

& can be used with simple variable or array of element.

- **&100** → pointing at constants

  int x[10];

  &x → pointing at array name and gives starting address of an array (first element)

  &(x+y) → pointing at an expression

  &x[0] → first element of array

  &x[i+3] → i+3th element of array

Initialization of pointer variable

- process of all assigning the address of a variable to a pointer variable is known as initialization
- All uninitialization has some unknown value

int x;

int *p; → **declaration of pointer**

p = &x → **initialization**

int *p = &x ; → **declare and initialization**

To print address of variables

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i = 5;
    int *ptr = &i;
    printf("\n the address of i using &num is %p %u", &i, &i);
    printf("\n the address of i using ptr is %p %u", ptr, ptr);
}
/*Output
 the address of i using &num is FFF4 65524
 the address of i using ptr is FFF4 65524
*/
```

- int x , *p = &x; → declare and initialized

float a , b;

int *p;

p = &a ; → **not allowed** (**a is float** and **p is pointing to integer datatype**)

int *p = 0 ; // **pointer variable with initial value 0**

int *p = NULL ;

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int *p; /* a pointer to an integer */
    //int *p=NULL; // assignment to pointer variable
    clrscr();
    printf("%d \n",*p);
    return 0;
}
```

Accessing a variable through its pointer

- ***** is used to access the value with the pointer
- It is known as **indirection operator**

e.g.

```
int x , *p;
p = &x;
int n ;  n = *p;
```

n = *p is equivalent to

n = *&x;   **or**   n = x;

POINTER ARITHMETIC

- If p is declared as a pointer variable of any type and it has been initialized properly, then, just like a simple variable, any operation can be performed with *p.
- Because * implies value at address, working with *p means working with the variable whose address is currently held by p.
- Any expression, whether relational, arithmetic, or logical, can be written, which is valid for a simple value variable.
- But with only p, operations are restricted as in each case address arithmetic has to be performed. The only valid operations on pointers are as follows.
  - Assignment of pointers to the same type of pointers: the assignment of pointers is done symbolically. Hence no integer constant except 0 can be assigned to a pointer.
  - Adding or subtracting a pointer and an integer.

POINTER ARITHMETIC

  - Subtracting or comparing two pointers (within array limits) that point to the elements of an array.
  - Incrementing or decrementing the pointers that point to the elements of an array. When a pointer to an integer is incremented by one, the address is incremented by two (as two bytes are used for int). Such scaling factors necessary for the pointer arithmetic are taken care of automatically by the compiler.
  - Assigning the value 0 to the pointer variable and comparing 0 with the pointer. The pointer with address 0 points to no where at all.
- Do not attempt the following arithmetic operations on pointers. They will not work.
  - Addition of two pointers
  - Multiplying a pointer with a number
  - Dividing a pointer with a number

To perform addition, read the two values using pointer variables

```
//Use of pointers without using the variable directly
void main()
{  int a, b, c, *pa, *pb, *pc;
   pa = &a;   pb = &b;     pc= &c;
   printf("\n Enter the first number:");
   scanf("%d",pa);
   printf("\n Enter the Second number:");
   scanf("%d",pb);
   *pc = *pa + *pb;
   printf("\n sum is = %d",*pc);
}
```

Output

```
 Enter the first number:10
 Enter the Second number:20
 sum is = 30
```

Generic Pointer

- A void pointer is a special type of pointer.
- It can point to any data type, from an integer value or a float to a string of characters.
- Its sole limitation is that the pointed data cannot be referenced directly (the asterisk * operator cannot be used on them) since its length is always undetermined.
- Therefore, *type casting or assignment* must be used to turn the void pointer to a pointer of a concrete data type to which we can refer.
- Take a look at the following example.

Generic Pointer

```
//void pointer
void main()
{
    int a = 5;
    double b=3.14;
    void *vp;
    vp = &a;
    printf("\n a = %d",*((int *)vp));
    vp = &b;
    printf("\n b = %lf",*((double *)vp));
}
/*Output
 a = 5
 b = 3.140000
*/
```

NULL POINTER

- Suppose a variable, e.g., a, is declared without initialization.
- int a;
- If this is made outside of any function, ANSI-compliant compilers will initialize it to zero.
- Similarly, an uninitialized pointer variable is initialized to a value guaranteed in such a way that it is certain not to point to any C object or function.
- A pointer initialized in this manner is called a *null pointer*.

NULL POINTER

- A *null pointer is a special pointer that points no where*.
- That is, no other valid pointer to any other variable or array cell or anything else will ever be equal to a null pointer.
- The most straightforward way to get a null pointer in the program is by using the predefined constant NULL, which is defined by several standard header files, including <stdio.h>, <stdlib.h>, and <string.h>.
- To initialize a pointer to a null pointer, code such as the following can be used.

```
#include <stdio.h>
int *ip = NULL;
```

NULL POINTER

- To test it for a null pointer before inspecting the value
  pointed to, code such as the following can be used.

  if(ip != NULL)

  printf(“%d\n”, *ip);
- It is also possible to refer to the null pointer using a constant 0, and to set null pointers by simply saying
- int *ip = 0;
- If it is too early in the code to know which address to assign to the pointer, then the pointer can be assigned to NULL, which is a constant with a value of zero defined in several standard libraries, including stdio.h.

NULL POINTER

```
#include <stdio.h>
int main()
{
    int *p;
    p = NULL;
    printf(“\n The value of p is %u”, p);
    return 0;
}
```

- **Output:**

```
The value of p is 0
```

Pointer to Pointer

- So far in the discussion, pointers have been pointing directly to data. C allows the use of pointers that point to pointers, and these, in turn, point to data.
- For pointers to do that, we only need to add an asterisk (*) for each level of reference. Consider the following declaration.

```
int a=5;
int *p; ¨ pointer to an integer
int **q; ¨ pointer to a pointer to an integer
p=&a;
q=&p;
```

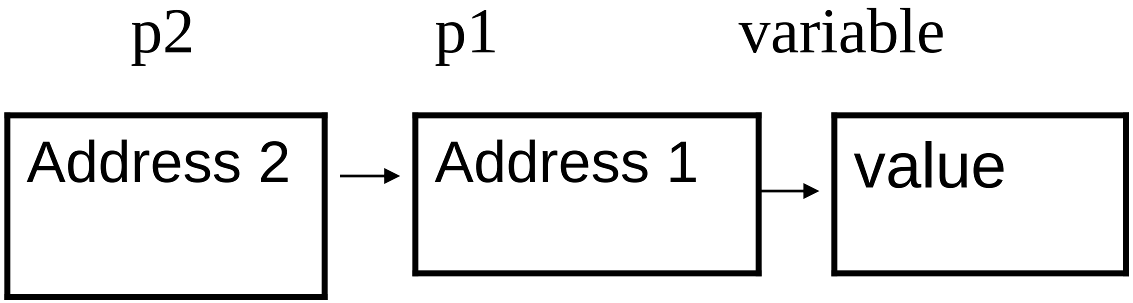
Pointer to Pointer

- To refer to a using pointer p, dereference it once, that is, *p.
- To refer to a using q, dereference it twice because here are two levels of indirection involved.
- If q is dereference once, actually p is referenced which is a pointer to an integer. It may be represented diagrammatically as follows.

- So, *p and **q print 5 if they are printed with a printf statement.
- q is a variable of type (int **) with a value of 65550
- *q is a variable of type (int *) with a value of 65540
- **q is a variable of type (int) with a value of 5

Pointer to Pointer

- pointer points to another pointer.



p2 p1 variable

Address 2 → Address 1 → value

```
main ()
{
    int x , *p1 , **p2;
    x = 100;
    p1 = &x;
    p2 = &p1;
    printf(“%d”, **p2);
}
```

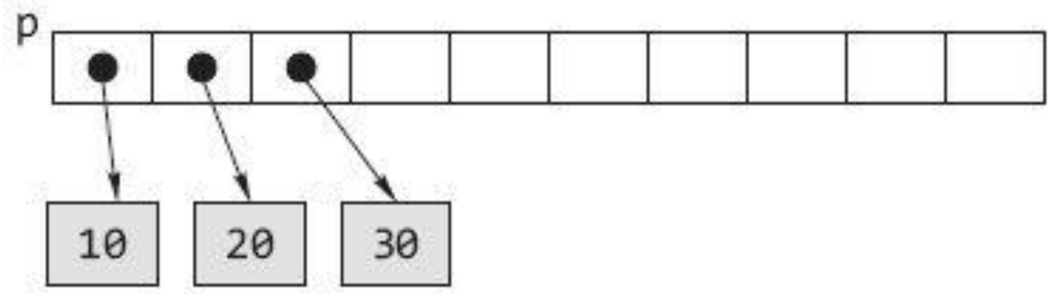
ARRAY OF POINTERS

- An array of pointers can be declared very easily. It is done thus.

```
int *p[10];
```

- This declares an array of 10 pointers, each of which points to an integer. The first pointer is called p[0], the second is p[1], and so on up to p[9]. These start off as uninitialized—they point to some unknown point in memory. We could make them point to integer variables in memory thus.

```
int* p[10];
int a = 10, b = 20, c = 30;
p[0] = &a;
p[1] = &b;
p[2] = &c;
```

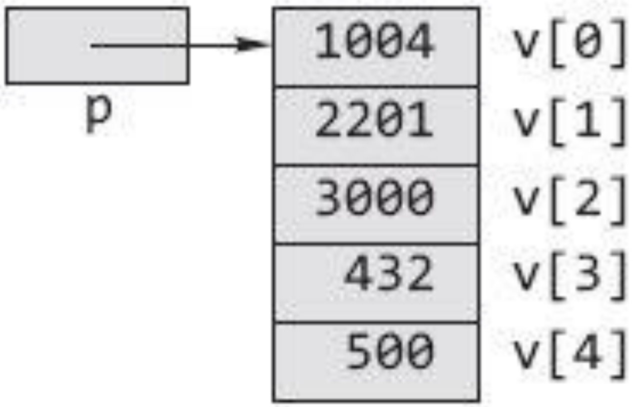


POINTERS TO AN ARRAY

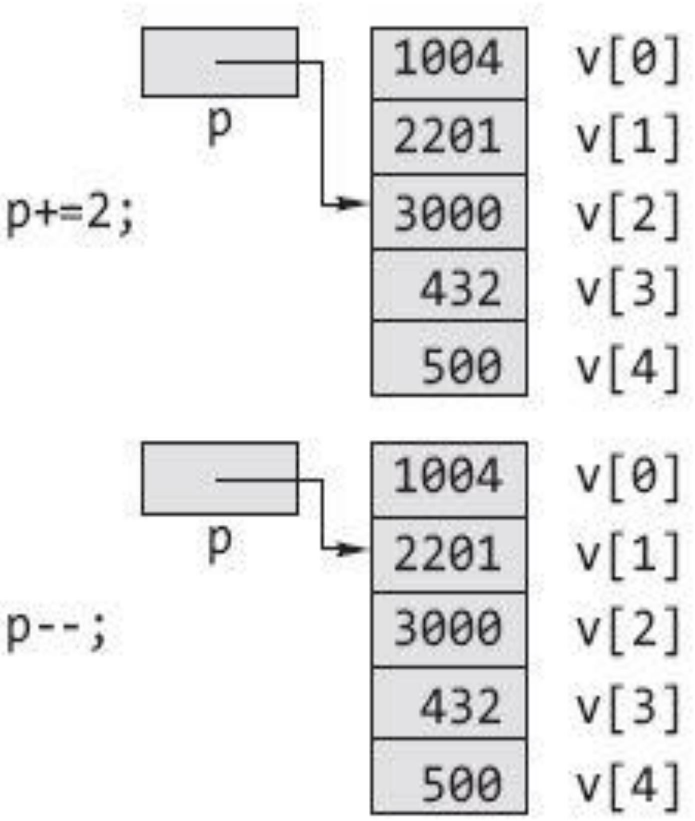
- Suppose we have an array of unsigned long values called v. We can declare a pointer to a simple integer value and make it point to the array as is done normally.

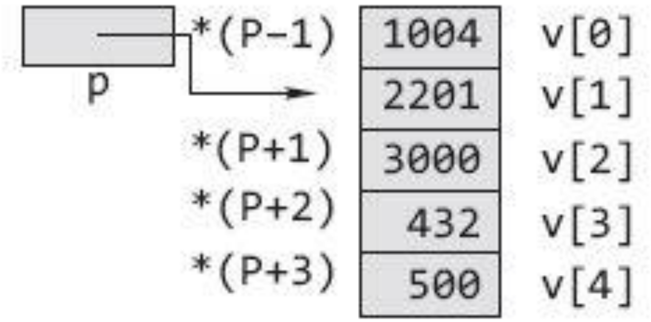
```
int v[5] = {1004, 2201, 3000, 432, 500};
int *p = v;
printf(“%d \n”, *p);
```

- This piece of code displays the number, which the pointer p points to, that is the first number in the array, namely 1004.



POINTERS TO AN ARRAY





POINTERS TO AN ARRAY

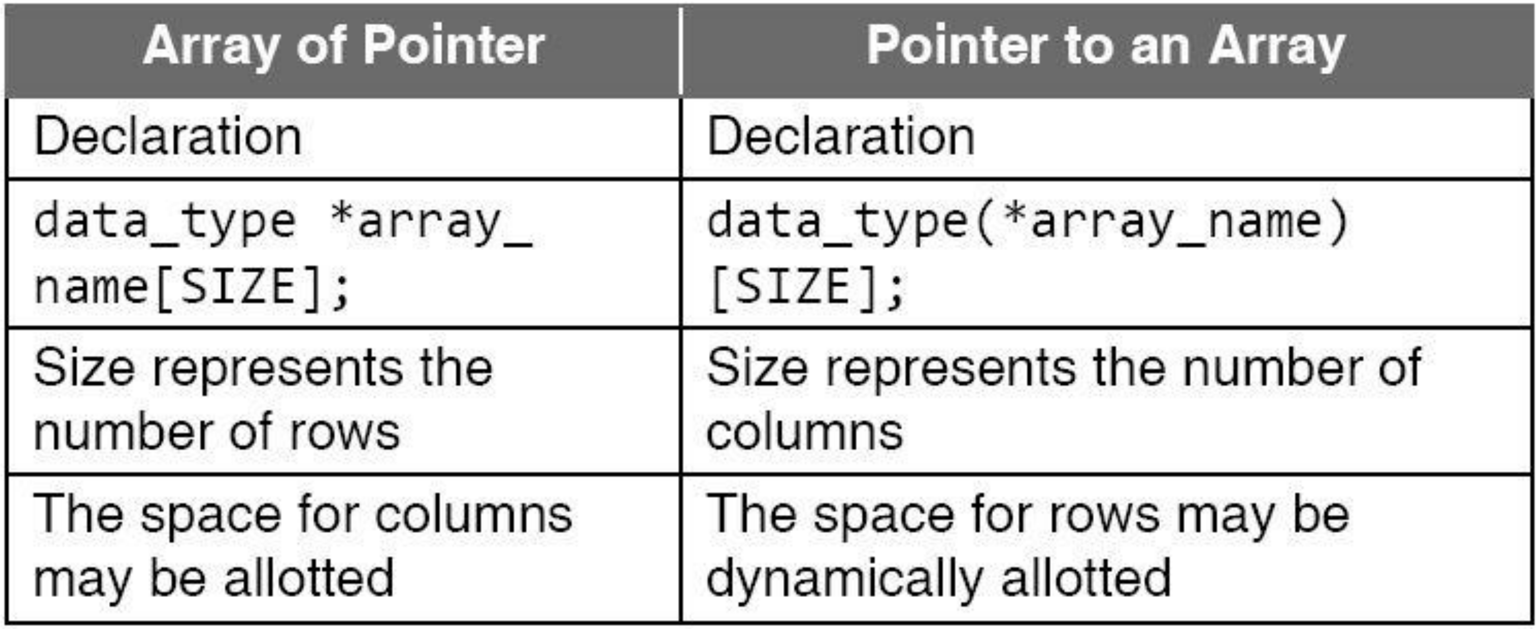
```
int main()
{
        int a[2][3]={{3,4,5},{6,7,8}};
        int i; int(*pa)[3];
        pa=a;
        for(i=0;i<3;++i)
             printf("%d\t",(*pa)[i]);
        printf("\n");
        pa++;
        for(i=0;i<3;++i)
             printf("%d\t",(*pa)[i]);
        return 0;
}
```

Output:

```
3 4 5
6 7 8
```

POINTERS TO AN ARRAY

- Difference between an array of pointers and a pointer to an array.



Array of Pointer	Pointer to an Array
Declaration	Declaration
`data_type *array_name[SIZE];`	`data_type(*array_name)[SIZE];`
Size represents the number of rows	Size represents the number of columns
The space for columns may be allotted	The space for rows may be dynamically allotted

Pointer to an Array

- **Compiler** allocate the **base address** and **memory storage** to an array.
- **Base address is the location of the first element.**
- **pointer** is declared which points to an array in number of way

  p = x or p = &x[0]

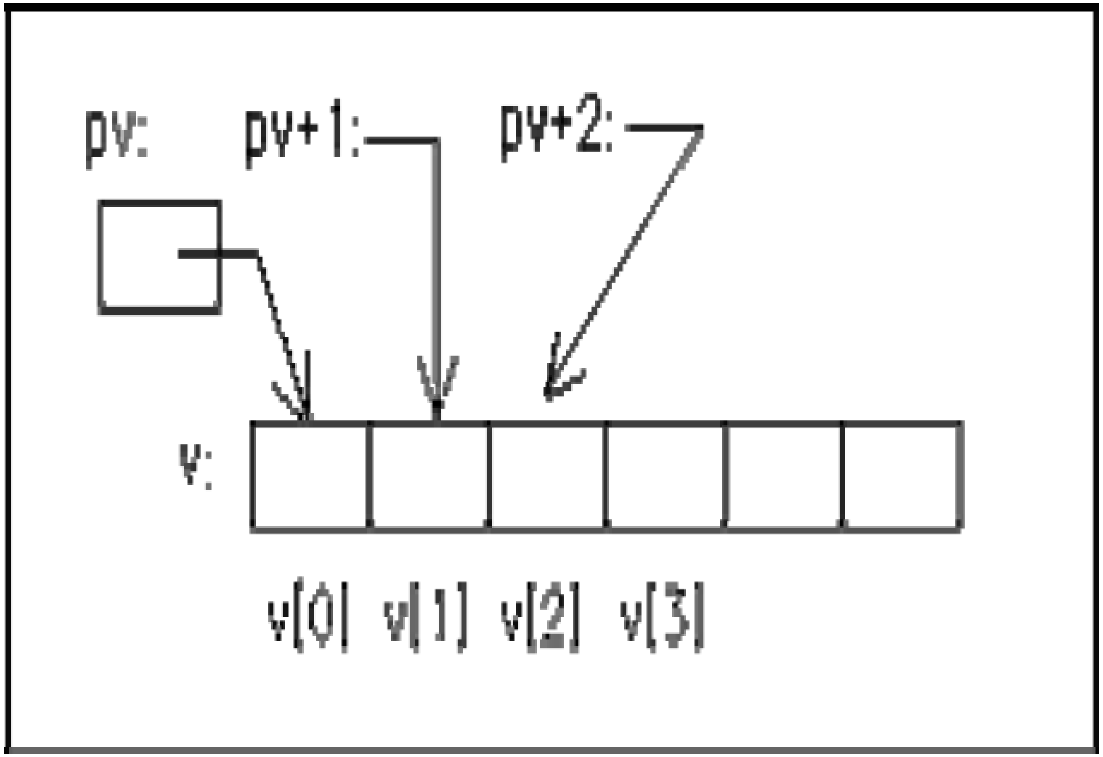
If p = &x[0];

p+1 = &x[1];

p+2 = &x[2];

p +3 = &x[3];

- Here pv is simply indication pointer v



Value of Element of array is accessed by : *(p+i)

To print an array elements using pointer variable

```
void main()
{
    int a[5]= {10, 20, 30, 40, 50};
    int *p,i;

    p = a;          //p = &a          // p = &a[0];
    for(i=0;i<5;i++)
    {
          printf(" %d ",*p);
          p++;
    }
}
```

To perform summation of an array elements using pointer

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a[5]= {10, 20, 30, 40, 50};
    int *p,i , sum = 0;
    clrscr();
    p = a;       //p = &a       // p = &a[0];
```

To perform summation of an array elements using pointer

```
    for(i=0; i<5; i++)
    {    printf("%d ",*p);
         sum = sum + *p;
         p++;  }
    printf("\n Sum = %d",sum);
    getch();
}
/*  OUTPUT
10 20 30 40 50
Sum = 150
*/
```

Passing an array to a function

- An array may be passed to a function, and the elements of that array may be modified without having to worry about referencing and dereferencing.
- Since arrays may transform immediately into pointers, all the difficult stuff gets done automatically.
- A function that expects to be passed with an array can declare that formal parameter in one of the two ways.
- **int a[] or int *a**
- When passing an array name as argument to a function, the address of the zeroth element of the array is copied to the local pointer variable in the function.

Passing an array to a function

- The values of the elements are *not copied. The corresponding local variable* is considered as a pointer variable, having all the properties of pointer arithmetic and dereferencing.
- It is *not an address* constant.
- This is illustrated with an example. The relevant function calls in main() and the corresponding function headers are shown as follows for easy reference.

Passing an array to a function

```
#define MAX 50
int main( )
{
        int arr[MAX],n;
        int getdata(int *, int);
        void show(int *, int);
        n = getdata(arr, MAX);
        show(arr, n);
        return 0;
}
void show(int *a, int n)
{
        int i;
        for(i=0;i<n;++i)
        printf(“\n %d”, *(a+i));
}
```

Passing an array to a function

```
/* Function reads scores in an array. */
int getdata(int *a, int n)
{
    int x, i = 0;
    printf(“\n Enter the array elements one by one\n”);
    while(i < n)
    {
        scanf(“%d”, &x)
        *(a + i) = x;
        i++;
    }
    return i;
}
```

Pointer to 2D array

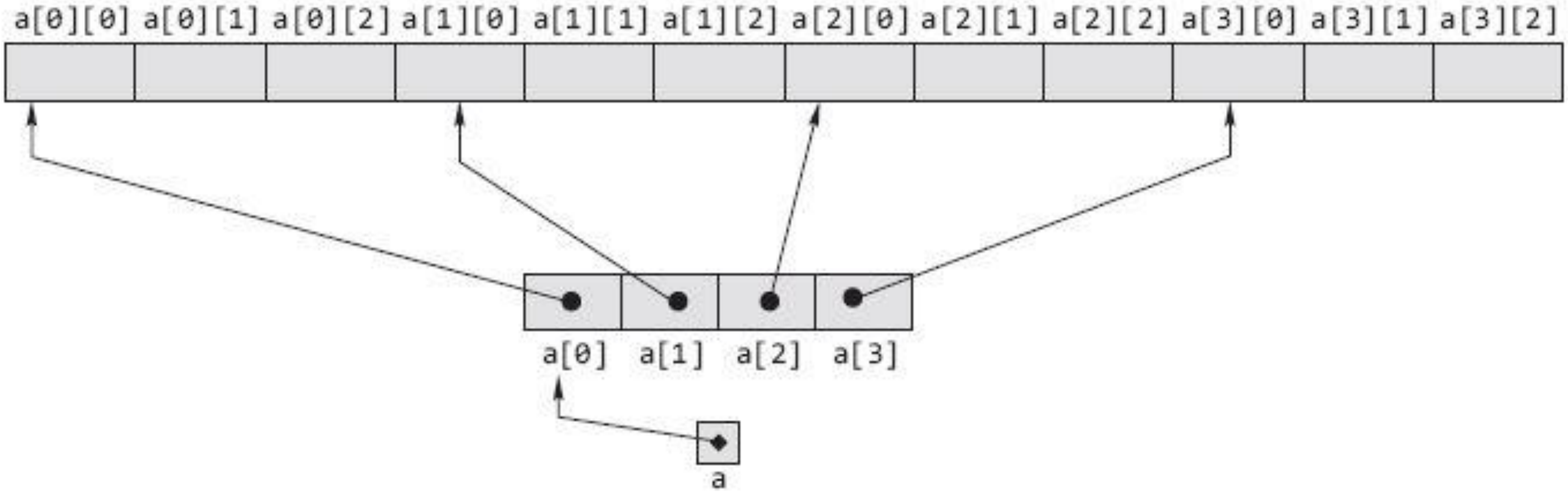


Figure 7.10 Physical representation of a two-dimensional array

Example: ptr2Darrtofunc.c
Exmaple2: ptr2Darr2func2.c

POINTERS TO FUNCTIONS

- One of the power features of C is to define pointers to functions. Function pointers are pointers, i.e., variables, which point to the address of a function.
- A running program is allocated a certain space in the main memory.
- The executable compiled program code and the used variables are both put inside this memory. Thus a function in the program code has an address.
- Like other pointer variables, function pointers can be declared, assigned values, and then used to access the functions they point to.

POINTERS TO FUNCTIONS

- **Declaration of a Pointer to a Function**
- Function pointers are declared as follows:

```
Return_type(*function_pointer_name
(argument_type1, argument_type2, ...);
```

- In the following example, a function pointer named fp is declared. It points to functions that take one float and two char and return an int.

```
int(*fp)(float, char, char);
```

- Some examples include the following.

```
int(*fp)();
double(*fptr)();
```

- Here, fp is declared as a pointer to a function that returns int type, and fptr is a pointer to a function that returns double.

POINTERS TO FUNCTIONS

- The interpretation is as follows for the first declaration: the dereferenced value of fp,
- i.e., **(*fp)**
- followed by () indicates a function that returns integer type. The parentheses are essential in the declarations. The declaration without the parentheses

int *fp();

- declares a function fp that returns an integer pointer.

POINTERS TO FUNCTIONS

- **Initialization of Function Pointers**
- Like other pointer variables, function pointers must be

initialized prior to use. It is quite easy to assign the address of a function to a function pointer. One simply uses the name of a function. It is optional to use the address operator & in front of the function's name. For example, if add() and sub() are declared as follows

int add(int, int);

and

int sub(int, int);

- the names of these functions, add and sum, are pointers to those functions. These can be assigned to pointer variables.

fpointer = add;

fpointer = sub;

POINTERS TO FUNCTIONS

- **Calling a Function using a Function Pointer**
- In C there are two ways of calling a function using a function pointer: use the name of the function pointer instead of the name of the function or explicitly dereference it.

```
result1 = fpointer(4, 5);
result2 = fpointer(6, 2);
```

- The following program illustrates the above facts.
- Example : **callfuncbyfuncptr.c**
- Example : **arroffuncptr.c**

Pointer to a char string

- char ***str** = “welcome”;
  - It create the string and stores its address to the pointer variable **str.**
  - **str** points to first character of the string.
- e.g char *str1;
- str1 = “welcome”; → its **not** a string copy.

Because the str1 is pointer variable.

- no need to use the * for printing a string using **printf or puts function**.

To print character array using pointer variable

```
#include <stdio.h>
#include <conio.h>
void main()
{
  char ch[5]= {'a', 'b', 'c', 'd', 'e'};
   char *p,
  int i;
   clrscr();
   p = ch;   //p = &ch   // p = &ch[0];

   for(i=0;i<5;i++)
   {   printf("%c %u\n",*p,p);
       p++;
    }
}
```

```
/* Output
a 65520
b 65521
c 65522
d 65523
e 65524
*/
```

To print character array using pointer

```
main()
  {     char *str;
        printf("Enter string");
        gets(str);
        while(*str!='\0')
        {
        printf("\n %c the character is stored at %u", *str,str);
        str++;
      }
}
```

```
/*Output
Enter string kanak
 k the character is stored at 858
 a the character is stored at 859
 n the character is stored at 860
 a the character is stored at 861
 k the character is stored at 862
*/
```

To reverse a string using strrev()

```
main()
{
    char *str;
    printf("Enter string");
    gets(str);
    strrev(str);
    puts(str);
}
```

```
/* Output
Enter string Nadiad
daidaN
*/
```

Pointer as function argument

- In the function call the **address of the parameter** is passed .
- Parameter receiving in function definition should be **pointer.**
- The process of calling a function using passing the address of variable is known as **call by reference.**

Pass by value

```
#include<stdio.h>
#include<conio.h>
void swap(int a,int b)
{  int t;
   t = a;
   a = b;
   b = t;
}
void main()
{
   int x = 5, y =20;
   swap(x,y);
   printf("\n X is = %d Y is = %d",x,y);
}
/*Output
X is = 5 Y is = 20   */
```

Pass by Address

```
#include<stdio.h>
#include<conio.h>
void swap(int *a,int *b)
{  int t;
   t = *a;
   *a = *b;
   *b = t;    }
void main()
{  int x = 5, y =20;
   clrscr();
   printf("\n Before Swap X is = %d Y is = %d",x,y);
   swap(&x,&y);
   printf("\n After Swap X is = %d Y is = %d",x,y);
}
```

```
/*Output
 Before Swap X is = 5 Y is = 20
 After Swap X is = 20 Y is = 5  */
```

- **ina[n][m]** notation, there are n rows and m columns.
- Array elements are stored **row by row**.
- When pointer points to 2D its need row and column calculation to access the value.

- **Program to implement user defined function which returns multiple values and this function is used to perform addition and subtraction of two given numbers.**

Function should return multiple values

```
void operation(int x,int y,int *sum, int *diff);
   //prototype
void main()
{  int x=20,y=10,s,d;
   operation(x,y,&s,&d);
   printf("\n Sum is %d Diff is %d",s,d);
}
void operation(int a,int b,int *sum, int *diff)
{  *sum = a + b;
   *diff = a - b;
}
/*Output
Sum is 30 Diff is 10
*/
```

Pointer to a structure

- When pointer to a structure is declared it use → operator for accessing of member.
- pointer can also assign to the array of structure.

Which points to first structure element of an array.

e.g

```
struct item
{
  int i;
}*it ;
```

if `struct item a[3] ;`

then `it= a;` → `it` pointer points to `a[0]`

For accessing member variable `it→ i ;`

Or

```
(*it).i
```

Pointer to a structure Program

```
struct person
{
   int age;
   float weight;
};

int main()
{   struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```

DMA

- A problem with many simple programs such as those written so far is that they tend to use fixed-size arrays, which may or may not be big enough.
- There are more problems of using arrays. Firstly, there is the possibility of overflow since C does not check array bounds.
- Secondly, there is wastage of space—if an array of 100 elements is declared and a few are used, it leads to wastage of memory space.
- How can the restrictions of fixed-size arrays be avoided?
- The answer is dynamic memory allocation.
- It is the required memory that is allocated at run-time (at the time of execution). Where fixed arrays are used, static memory allocation, or memory allocated at compile time, is used.

DMA

- Dynamic memory allocation is a way to defer the decision of how much memory is necessary until the program is actually running, or give back memory that the program no longer needs.
- The area from where the application gets dynamic memory is called heap.
- The heap starts at the end of the data segment and grows against the bottom of the stack. If both meet, the program is in trouble and will be terminated by the operating system.
- Thus, C gives programmers the standard sort of facilities to allocate and de-allocate dynamic heap memory.

DMA

- ***Static memory allocation***
  - *The compiler allocates the required* memory space for a declared variable.
  - By using the address of operator, the reserved address is obtained that maybe assigned to a pointer variable. Since most declared variables have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation.
- ***Dynamic memory allocation***
  - *A dynamic memory allocation* uses functions such as malloc() or calloc() to get memory dynamically.
  - If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run-time.

DMA

- C provides access to the heap features through library functions that any C code can call. The prototypes for these functions are in the file <stdlib.h>.
- So any code which wants to call these must #include that header file.
- **void* malloc(size_t size):**
- Request a contiguous block of memory of the given size in the heap.
- malloc() returns a pointer to the heap block or NULL if the request is not satisfied.
- The type size_t is essentially an unsigned long that indicates how large a block the caller would like measured in bytes.
- Because the block pointer returned by malloc() is a void * (i.e., it makes no claim about the type of its pointee), a cast will probably be required when storing the void pointer into a regular typed pointer

DMA

- **calloc():**
- works like malloc, but initializes the memory to zero if possible. The prototype is

  void * calloc(size_t count, size_t eltsize)
- This function allocates a block long enough to contain an array of count elements, each of size eltsize. Its contents are cleared to zero before calloc returns.

- **void free(void* block):**
- free() takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for reuse.
- After the free(), the client should not access any part of the block or assume that the block is valid memory.
- The block should not be freed a second time.

DMA

- **void* realloc(void* block, size_t size):**
- Take an existing heap block and try to reallocate it to a heap block of the given size which may be larger or smaller than the original size of the block.
- It returns a pointer to the new block, or NULL if the reallocation was unsuccessful.
- Remember to catch and examine the return value of realloc().
- It is a common error to continue to use the old block pointer.
- realloc() takes care of moving the bytes from the old block to the new block.
- realloc() exists because it can be implemented using low-level features that make it more efficient than the C code a programmer could write.

DMA

- To use these functions, either **stdlib.h** or **alloc.h** must be included as these functions are declared in these header files.
- **Point to be noted:**
- All of a program’s memory is de-allocated automatically when it exits.
- So a program only needs to use free() during execution if it is important for the program to recycle its memory while it runs, typically because it uses a lot of memory or because it runs for a long time.
- The pointer passed to free() must be the same pointer that was originally returned by malloc() or realloc(), not just a pointer into somewhere within the heap block.

DMA

- Note that if sufficient memory is not available, the malloc returns a NULL. Because malloc can return NULL instead of a usable pointer, the code should *always check the return value of* malloc to see whether it was successful.
- If it was not, and the program dereferences the resulting NULL pointer, the program will crash.
- A call to malloc, with an error check, typically looks something like this.

```
int *ip;
*ip =(int *) malloc(sizeof(int));
if(ip == NULL)
{
printf(“out of memory\n”);
exit(0); /* ‘return’ may be used*/
}
```

DMA

- ***About exit()*** In the previous example there was a case in which we could not allocate memory.
- In such cases it is often best to write an error message, and exit the program.
- The exit() function will stop the program, clean up any memory used, and will close any files that were open at the time.

```
#include <stdlib.h>
void exit(int status);
```

- When memory is allocated, the allocating function

(such as malloc() and calloc()) returns a pointer.

DMA

- With the older compiler, the type of the returned pointer is char; with the ANSI compiler it is void.
- malloc() returns a void pointer (because it does not matter to malloc what type this memory will be used for) that needs to be cast to one of the appropriate types.
- The expression (int*) in front of malloc is called a ‘cast expression’.
- **Example: mallocdemo.c**
- **Example: DMAarray.c**

DMA

- Another way is to use calloc() that allocates memory and **clears it to zero.**
- It is declared in stdlib.h.

void * calloc(size_t count, size_t eltsize)

- This function allocates a block long enough to contain a vector of count elements, each of size eltsize.
- Its contents are cleared to zero before calloc returns.

- **Example: callocdemo.c**

DMA

- calloc() can be defined using malloc() as follows.

```
void * calloc(size_t count, size_t eltsize)
{
size_t size = count * eltsize;
void *value = malloc(size);
    if(value != 0)
        memset(value, 0, size);
return value;
}
```

- But in general, it is not necessary that calloc() calls malloc() internally.
- memset sets n bytes of s to byte c where its prototype is given by
- **void *memset(void *s, int c, size_t n);**
- memset also sets the first n bytes of the array s to the character c. The following program illustrates the use of the memset function.

DMA

```
#include <string.h>
#include <stdio.h>
#include <mem.h>
int main(void)
{
    char b[] = “Hello world\n”;
    printf(“b before memset: %s\n”, b);
    memset(b, ‘*’, strlen(b) - 1);
    printf(“b after memset: %s\n”, b);
    return 0;
}
```

Output:

```
b before memset: Hello world
b after memset: ***********
```

DMA

- The malloc() function has one potential error. If malloc() is called with a zero size, the results are unpredictable.
- It may return some other pointer or it may return some other implementation-dependent value. It is recommended that malloc() never be called with a size zero.

DMA

- **Points to be Noted**
- calloc() requires two parameters, the first for the number of elements to be allocated and the second for the size of each element, whereas malloc() requires one parameters.
- calloc() initializes all the bits in the allocated space set to zero whereas malloc() does not do this. A call to calloc() is equivalent to a call to malloc() followed by one to memset().
- calloc(m, n) is essentially equivalent to

```
p = malloc(m * n);
memset(p, 0, m * n);
```

- If malloc() is called with a zero size, the results are unpredictable. It may return some other pointer or it may return some other implementation-dependent value.

DMA

- **Important difference between malloc() and calloc():**

malloc()	calloc()
Malloc() function will create a single block of memory of size specified by the user.	Calloc() function can assign multiple blocks of memory for a variable.
Malloc function contains garbage value.	The memory block allocated by a calloc function is always initialized to zero.
Number of arguments is 1.	Number of argument is 2.
Calloc is slower than malloc.	Malloc is faster than calloc.
It is not secure as compare to calloc.	It is secure to use compared to malloc.
Time efficiency is higher than calloc().	Time efficiency is lower than malloc().
Malloc() function returns only starting address and does not make it zero.	Before allocating the address, Calloc() function returns the starting address and make it zero.
It does not perform initializes of memory.	It performs memory initialization.

Freeing Memory

- Memory allocated with malloc() does not automatically get de-allocated when a function returns, as automatic duration variables do, but it does not have to remain for the entire duration of the program, either.
- In fact, many programs such as the preceding one use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they do not need that particular piece any more (when function or main() finishes).
- Because memory is not inexhaustible, it is a good idea to de-allocate (that is, release or *free)* memory that is no longer being used.
- Dynamically allocated memory is de-allocated with the free function. If p contains a pointer previously returned by malloc(), a call such as **free(p);**

Freeing Memory

- **free(p);** will ‘give the memory back’ to the stock of memory (sometimes called the ‘arena’ or ‘pool’) from which malloc requests are satisfied. When the allocated memory is deallocated with the free() function, it returns the memory block to the ‘free list’ within the heap.
- **Example: freedemo.c**
- Naturally, once some memory has been freed, it must not be used any more. After calling free(p); it is probably the case that p still points at the same memory.
- However, since it has been given back, it is now available, and a later call to malloc() might give that memory to some other part of the program.
- If the variable p is a global variable or will otherwise stick around for a while, one good way to record the fact that it is not to be used any more would be to set it to a null pointer.

Freeing Memory

- free(p);
- p = NULL;
- Now the question is why NULL should be assigned to the pointer after freeing it.
- After a pointer has been freed, the pointed-to data can no longer be used. The pointer is said to be a *dangling pointer; it does not point at anything useful.*
- If a pointer is ‘NULL out’ or ‘zero out’ immediately after freeing it, the program can no longer get in trouble by using that pointer.
- malloc() and calloc() can also be used in a similar way with strings.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *str = NULL;
   /* allocate memory for string */
   str = (char *)calloc(10, sizeof(char));
   /* copy “Hello” into string */
   strcpy(str, “Hello”);
   /* display string */
   printf(“String is %s\n”, str);
   /* free memory */
   free(str);
   str=NULL;
   return 0;
}
```

Reallocating Memory Blocks

- For example, if a series of items entered by the user has to be stored, the only way to know how many they are totally depends on the user input.
- Here malloc() will not work.
- It is the realloc() function that is required.
- For example, to point ip variable from an earlier example 200 ints instead of 100, try calling
- ip = realloc(ip, 200 * sizeof(int));
- Since each block of dynamically allocated memory needs to be contiguous (so that one can treat it as if it were an array), it may be a case where realloc cannot make the old block of memory bigger 'in place', but has to reallocate it elsewhere to find enough contiguous space for the new requested size.
- realloc() does this by returning a new pointer.

Reallocating Memory Blocks

- Finally, if realloc() cannot find enough memory to satisfy the new request at all, it returns a NULL.
- Therefore, usually the old pointer is not overwritten with realloc()'s return value until it has been tested to make sure it is not a null pointer.
- If realloc() returns something other than a null pointer, then memory reallocation has succeeded and ip might be set to what it returned. If realloc() returns a null pointer, however, the old pointer ip still points at the original 100 values.
- **Example: reallocdemo.c**

MEMORY LEAK AND MEMORY CORRUPTION

- A *memory leak occurs when a dynamically allocated area* of memory is not released or when no longer needed.
- In C, there are two common coding errors that can cause memory leaks.
- Firstly, an area can be allocated but, under certain circumstances, the control path bypasses the code that frees the area. This is particularly likely to occur if the allocation and release are handled in different functions or even in different source files.
- Secondly, the address of an area can be stored in a variable (of pointer data type) and then the address of another area stored in the same variable without releasing the area referred to the first time. The original address has now been overwritten and is completely lost.

MEMORY LEAK AND MEMORY CORRUPTION

- In a reasonably well-structured program, the second type are usually the harder to find.
- In some programming languages and environments, special facilities known as *garbage collectors are available to track down and release* unreferenced dynamically allocated blocks. But it should be noted that automatic garbage collection is not available in C.
- It is the programmer’s responsibility to deallocate the memory that was allocated through the use of malloc() or calloc(). The following sample codes will cause memory leak.

```
char *oldString = “Old String”;
char newString;
strcpy(newString, oldString);
...
free(newString);
```

MEMORY LEAK AND MEMORY CORRUPTION

- Memory leaks are another undesirable result when a function is written as follows.

```
void my_function(void)
{
int *a;
a=(int *)malloc(100*sizeof(int));
/* Do something with a*/
/* forgot to free a */
}
```

- The only problem is that every time this function is called, it allocates a small bit of memory and never gives it back. If this function is called a few times, all will be fine and the difference will not be noticed. On the other hand, if it is called often, then it will gradually eat all the memory in the computer.

- It will eventually crash the computer.

MEMORY LEAK AND MEMORY CORRUPTION

- ***Dangling pointer In C****, a pointer may be used to hold* the address of dynamically allocated memory. After this memory is freed with the free() function (in C), the pointer itself will still contain the address of the released block. This is referred to as a dangling pointer.
- Using the pointer in this state is a serious programming error.
- Pointers should be assigned 0, or NULL in C after freeing memory to avoid this bug.
- If the pointer is reassigned a new value before beingfreed, it will lead to a ‘dangling pointer’ and memory leak.

MEMORY LEAK AND MEMORY CORRUPTION

- Consider the following example.

```
char *a = malloc(128*sizeof(char));
char *b = malloc(128*sizeof(char));
b = a;
free(a);
free(b); /* will not free the pointer to the
original allocated memory.*/
```

MEMORY LEAK AND MEMORY CORRUPTION

- ***Memory corruption Memory when altered without an*** explicit assignment due to the inadvertent and unexpected altering of data held in memory or the altering of a pointer to a specific place in memory is known as memory corruption.
- The following are some examples of the causes of memory corruption that may happen.

1.Buffer overflow A case of overflow: *Overwrite beyond* allocated length

```
char *a = malloc(128*sizeof(char));
memcpy(a, data, dataLen); /* Error if dataLen is
too long. */
```

MEMORY LEAK AND MEMORY CORRUPTION

- A case of index of array out of bounds: (array index overflow—index too large/underflow—negative index)

Char *s=”Oxford University”;

ptr = (char *) malloc(strlen(s));

/* Should be (s + 1) to account */

/* for null termination.*/

strcpy(ptr, s);

/* Copies memory from string s which is one byte

longer than its destination ptr.*/

- Overflow by one byte

2.Using an address before memory is allocated and set

int *ptr;

ptr=5;

- In this case, the memory location is NULL or random

MEMORY LEAK AND MEMORY CORRUPTION

3.Using a pointer which is already freed

```
char *a = (char *)malloc(128*sizeof(char));
...
free(a);
puts(a); /* This will probably work but dangerous. */
```

4.Freeing memory that has already been freed :Freeing a pointer twice:

```
char *a = malloc(128*sizeof(char));
free(a);
... Do Something ...
free(a);
```

/* A check for NULL would indicate nothing. This memory space may be reallocated and thus one may be freeing memory. It does not intend to free or portions of another block of memory. The size of the block of memory allocated is often held just before the memory block itself..*/

MEMORY LEAK AND MEMORY CORRUPTION

5. Freeing memory which was not dynamically allocated

```
double a=6.12345, *ptr;
ptr = &a;
...
free(ptr);
```

POINTER AND CONST QUALIFIER

- A declaration involving a pointer and const has several possible orderings.
- **Pointer to Constant**
- The const keyword can be used in the declaration of the pointer when a pointer is declared to indicate that the value pointed to must not be changed.
- If a pointer is declared as follows

```
int n = 10;
const int *ptr=&n;
```

- The second declaration makes the object that it points at read-only and of course, both the object and what it points at might be constant.
- Because we have declared the value pointed to by ptr to be const, the compiler will check for any statements that attempt to modify the value pointed to by ptr and flag such statements as an error.

POINTER AND CONST QUALIFIER

- For example, the following statement will now result in an error message from the compiler:

```
*ptr= 100; /* ERROR */
```

- As the declaration asserted that what ptr points to must not be changed. But the following assignment is valid.

```
n = 50;
```

- The value pointed to has changed but here it was not tried to use the pointer to make the change. Of course, the pointer itself is not constant, so it is always legal to change what it points to:

```
int v = 100;
ptr = &v; /* OK - changing the address in ptr */
```

- This will change the address stored in ptr to point to the variable v.
- It is to be noted that the following declarations are equivalent.

```
const int *ptr=&n;
int const *ptr=&n;
```

POINTER AND CONST QUALIFIER

- **Constant Pointers**
- Constant pointers ensure that the address stored in a pointer cannot be changed.
- Consider the following statements -

```
int n = 10;
int *const ptr = &n; /* Defines a constant */
```

- Here’s how one could ensure that a pointer always points to the same object; the second statement declares and initializes ptr and indicates that the address stored must not be changed.
- Any attempt to change what the pointer points to elsewhere in the program will result in an error message when you compile:

```
int v = 5;
ptr = &v; /* Error - attempt to change a constant
pointer */
```

POINTER AND CONST QUALIFIER

- It is still legitimate to change the value that ptr points to using ptr though:

```
*ptr = 100; /* OK - changes the value of v */
```

- This statement alters the value stored in v through the pointer and changes its value to 100.
- You can create a constant pointer that points to a value that is also constant:

```
int n = 25;
const int *const ptr = &n;
```

- ptr is a constant pointer to a constant so everything is fixed.
- It is not legal to change the address stored in ptr as well as ptr cannot be used to modify what it points to.