Chapter 1

Pointers in C

Learning Objectives

- Understand memory addresses
- Understand the concept of pointers
- Learn the use of pointer variables and understand call-by-value and call-by-address
- Get acquainted with dereferencing
- Study equivalence among arrays and pointers—treating pointers as arrays
- Learn about the concept and construction of array of pointers and pointers to array

Learning Objectives

- Discuss pointers and functions—parameter passing techniques, pointers as function parameters
- Get an idea about pointers to functions—functions as arguments to another function
- Understand dynamic memory allocation using pointers
- Comprehend memory leak, memory corruption, and garbage collection
- Decipher (long) pointer declarations
- Understand pointer arithmetic

Introduction

- Pointers allow new and ugly types of bugs.
- Pointer bugs can crash in random ways, which makes them more difficult to debug.
- But at the same time, pointers seem to make the code harder to understand. However, with increased power, pointers bring increased responsibility.
- The only peculiarity of C, compared to other languages, is its heavy reliance on pointers and the relatively permissive view of how they can be used.

Key Words

- **Dynamic data structures:** Those that are built up from blocks of memory allocated from the heap at run-time.

- **Memory leak:** A commonly used term indicating that a program is dynamically allocating memory but not properly deallocating it, which results in a gradual accumulation of unused memory by the program to the detriment of other programs, the operating system, and itself.

- **Heap:** This memory region is reserved for dynamically allocating memory for variables at run-time. Dynamic memory allocation is done by using the malloc() or calloc() functions.

Key Words

- **Call by address:** Facilitating the changes made to a variable in the called function to become permanently available in the function from where the function is called.

- **Call-by-value:** A particular way of implementing a function call, in which the arguments are passed by their value (i.e., their copies).

- **Dangling pointer:** A pointer pointing to a previously meaningful location that is no longer meaningful; usually a result of a pointer pointing to an object that is deallocated without resetting the value of the pointer.

Key Words

- **Dynamic memory allocation:** The process of requesting and obtaining additional memory segments during the execution of a program.
- **Function pointer:** A function has a physical location in memory that can be assigned to a pointer. Then it is called function pointer. This address is the entry point of the function and it is the address used when the function is called.
- **Garbage collection:** If only implicit dynamic allocation is allowed then deallocation must also be done by implicit means, which is often called garbage collection.

Key Words

- **NULL:** A special C constant, defined as macro in stdio.h as 0, or (void*)), that can be used as the null value for pointers.
- **Null pointer:** A null pointer is a special pointer value that points nowhere. I is initialized with value 0 or NULL.
- **Pointer:** A value or a variable with two attributes: (i) an address and (ii) a data type of what should be found at that address.

Key Words

- **Ragged array:** An array of pointers whose elements are used to point to arrays of varying sizes is called a ragged array.
- **Stack:** A data structure resembling a deck of cards; a new item can only be put on top of the deck (the push operation) or removed from the top of the deck (the pop operation).
- **Static memory allocation:** Memory layout for static data prepared by the compiler.
- **Void pointer:** A void pointer is a special type of pointer that can point to any data type.

Understanding Memory Addresses

- All computers have primary memory, also known as RAM or random access memory.
- For example, a computer may have 16, 32, 64, 128, 256, or 512 MB of RAM installed.
- RAM holds the programs that the computer is currently running along with the data they are currently manipulating.

Understanding Memory Addresses

- C uses pointers in three main ways.
  1. Pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when working with strings. There is an intimate link between arrays and pointers in C.
  2. C uses pointers to handle *variable parameters* passed to functions.
  3. They are used to create *dynamic data structures*, those that are built up from blocks of memory allocated from the heap at run-time. This is only visible through the use of pointers.

Understanding Memory Addresses

- The stack has three major functions:

1. The stack provides the storage area for local variables declared within the function.
2. The stack stores *housekeeping information involved* when function call is made.
3. The stack is needed for recursive call.

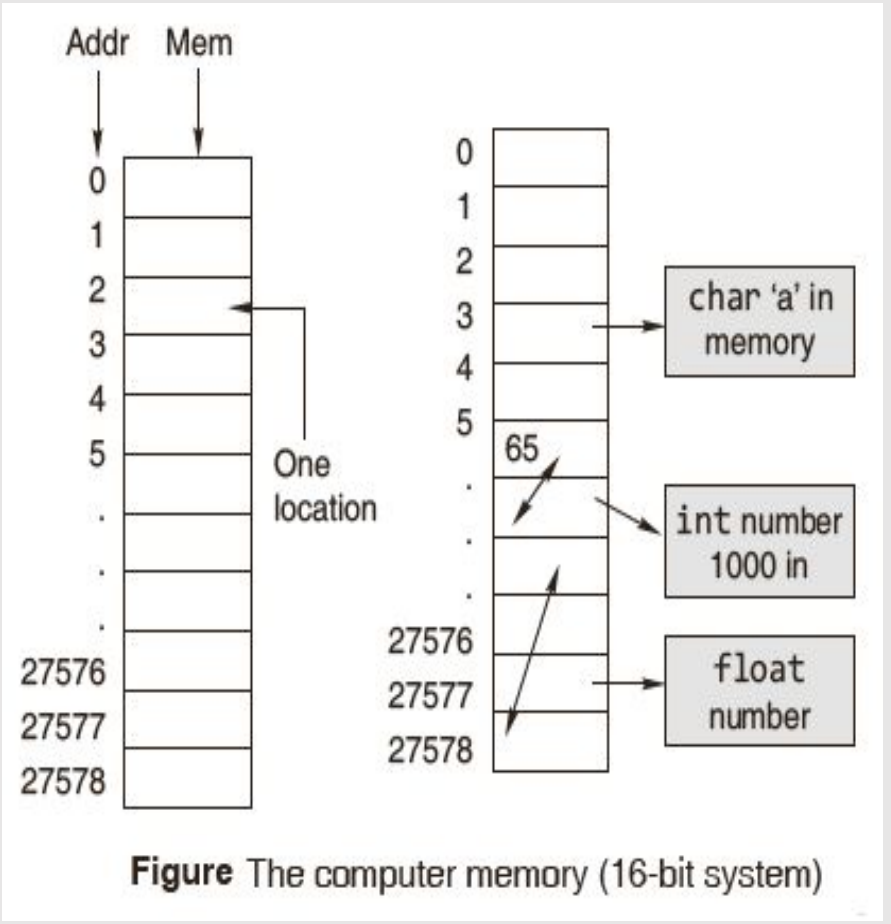


Figure The computer memory (16-bit system)

Memory Layout Summary

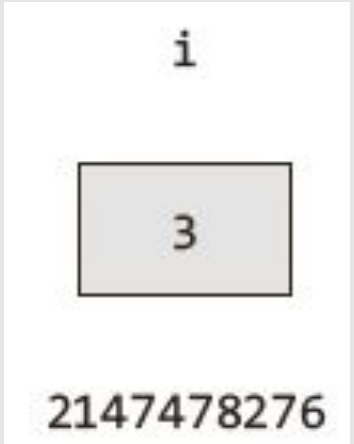
Memory Section Name	Description
Text (or the code segment)	This is the area of memory that contains the machine instructions corresponding to the compiled program. This area is READ ONLY and is shared by multiple instances of a running program.
Data	This area in the memory image of a running program contains storage for *initialized global* variables. This area is separate for each running instance of a program.
BSS	This is the memory area that contains storage for *uninitialized global* variables. It is also separate for each running instance of a program.
Stack	This region of the memory image of a running program contains storage for the automatic (local) variables of the program. It also stores context-specific information before a function call, e.g., the value of the instruction pointer (program counter) register before a function call is made. On most architectures, the stack grows from *higher memory to lower memory addresses*.
Heap	This memory region is reserved for dynamically allocating memory for variables at run-time. Dynamic memory allocation is done by using the malloc or calloc functions.
Shared libraries	This region contains the executable image of shared libraries being used by the program.

Address Operator (&)

- After declaring a variable, where the variable is to be located, is decided by the compiler and the operating system at run-time.
- After declaring a variable, if an attempt is made to output its value without assigning a value, the result is a garbage value.
- Readers might have noticed that when we call certain functions in C, the & sign is used. For example,

  scanf("%d", &i);

- This declaration tells the C compiler to
  - reserve space in memory to hold the integer value
  - associate the name i with this memory location
  - store the value 3 at this location



Pointer

- A pointer provides a way of accessing a variable without referring to the variable directly.
  - The mechanism used for this is the address of the variable.
  - A program statement can refer to a variable indirectly using the address of the variable.
  - A pointer variable is a variable that holds the memory address of another variable.
- Putting it another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable.

Pointer

- They are called pointers for the simple reason that, by storing an address, they ‘point’ to a particular point in memory.
- A pointer points to that variable by holding a copy of its address.
- Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. The address points to a value.

Pointers Can be Used to:

- Call by address, thereby facilitating the changes made to a variable in the called function to become permanently available in the function from where the function is called.
- Return more than one value from a function indirectly.
- Pass arrays and strings more conveniently from one function to another.
- Manipulate arrays more easily by moving pointers to them (or to parts of them) instead of moving the arrays themselves.

Pointers Can be Used to:

- Create complex data structures, such as linked lists and binary trees, where one data structure must contain references to other data structures.
- Communicate information about memory, as in the function malloc() which returns the location of free memory by using a pointer.
- Compile faster, more efficient code than other derived data types such as arrays.

Declaring a Pointer

- Just as any other variable in a program, a pointer has to be declared; it will have a value, a scope, a lifetime, a name; and it will occupy a certain number of memory locations. The pointer operator available in C is '*', called '*value at address'* operator.
- The syntax for declaring a pointer variable is

  datatype * pointer_variable;

Table Meaning of some pointer type variable declarations

Declaration	What it Means
`int p`	p is an integer
`int *p`	p is a pointer to an integer
`char p`	p is a character
`char *p`	p is a pointer to a character
`long p`	p is a long integer
`long *p`	p is a pointer to a long integer
`unsigned char p`	p is an unsigned character
`unsigned char *p`	p is a pointer to an unsigned character

Consider the Following Program

```
#include <stdio.h>
int main()
{
    int *p;
    float *q;
    double *r;
    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 *));
    return 0;
}
```

OUTPUT

In Turbo C:

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

In GCC:

The size of integer pointer is 4

The size of float pointer is 4

The size of double pointer is 4

The size of character pointer is 4

Drawbacks

- A programmer is likely to commit mistakes such as typographical mistakes and providing wrong offsets.
- Porting the code to other implementations would require changes, if data type sizes differ. This would lead to portability issues.
- Pointers have data types but the size of a pointer variable is always four bytes (in a 32-bit machine) whatever the data type is used in declaring it.

Where is a Pointer Stored?

- It can be defined and stored globally, or it can be defined local to a function and stored on the stack but generally not stored on the heap.
  - The size of the pointer depends on the implementation and for 32-bit operating systems, it generally requires four bytes of storage space.
  - A compiler writer can use any number of bytes desired to store a pointer.
- A pointer is like any other variable in the sense that it requires storage space somewhere in the computer's memory, but it is not like most variables because it contains no data, only an address.
  - Since it is an address, it actually contains a number referring to some memory location.
  - Dynamically allocated arrays can also be expanded during the execution of the program.

Initializing Pointers

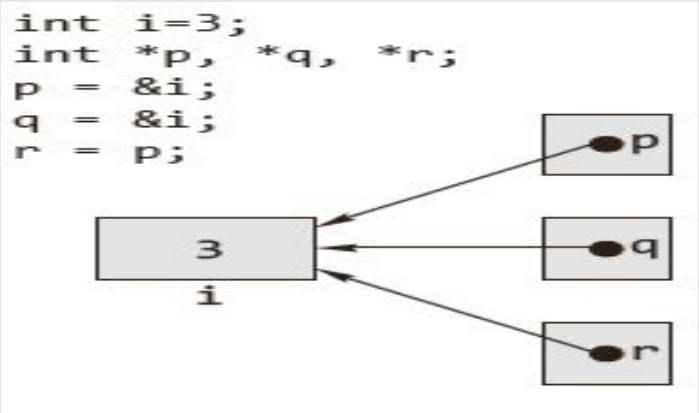
- Unlike a simple variable that stores a value, a pointer must be initialized with a specified address prior to its use.
- A pointer should be initialized with another variable's memory address, with 0, or with the keyword NULL prior to its use; otherwise the result may be a compiler error or a run-time error.
  - Consider the following program.

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

Assigning the Pointer

- A pointer is bound to a specific data type (except pointer to void) . A pointer to an int cannot hold the address of a character variable in which case a compiler error would result.
- The following pointer initializations are invalid.

```
int a=3, *ip;
float *p;
char ch='A';
p=&a;
ip=&ch:
```



- r points to the same address that p points to, which is the address of i.
- We can assign pointers to one another, and the address is copied from the right-hand side to the left-hand side during the assignment.

Printing Pointer Value

- A pointer variable contains a memory address that points to another variable. To print the memory address stored in pointers and non-pointer variables using the %p conversion specifier and to learn the use of the %p conversion specifier.
  - Addresses must always be printed using %u or %p or %x. If %p is used, the address is printed in hexadecimal form. If %u is used, address is printed in decimal form.
  - Study the following program.

```
#include <stdio.h>
int main(void)
{
int a=10, *p;
*p=&a;
printf("\n  *p = %p", *p);
return 0;
}
```

Output:

```
p = 0022FF2C
```

Is it Possible to Assign a Constant to a Pointer Variable?

- A pointer is a variable that holds the address of a memory location. That is, pointers are variables that *point to memory locations.*
- In C, pointers are not allowed to store any arbitrary memory address, but they can only store addresses of variables of a given type.
- Consider the following code:

```
int *pi;
pi= (int*)1000;
*pi = 5;
```

- Location 1000 might contain the program. Since it is a read only, the OS will throw up a segmentation fault.

Indirection Operator and Dereferencing

- The primary use of a pointer is to access and, if appropriate, change the value of the variable that the pointer is pointing to. The other pointer operator available in C is '*', called the 'value at address' operator. It returns the value stored at a particular address.
  - The value at address operator is also called indirection operator or dereferencing operator.
  - In the following program, the value of the integer variable num is changed twice.

Example

```
#include <stdio.h>
int main()
{
int num = 5;
int *iPtr = &num;
printf("\n The value of num is %d", num);
num = 10;
printf("\n The value of num after num = 10 is\
%d", num);
*iPtr = 15;
printf("\n The value of num
   after *iPtr = 15 is\
%d", num);
return 0;
}
```

Output:

```
The value of num is 5
The value of num after num =
   10 is 10
The value of num after *iPtr =
   15 is 15
```

Void 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.
  - Void pointer can point to a variable of any data type, from an integer value or a float to a string of characters.
  - The typecasting or assignment must be used to turn the void pointer to a pointer of a concrete data type to which we can refer.

Example

```
#include <stdio.h>
int main()
{
int a=5,
double b=3.1415;
void *vp;
vp=&a;
printf("\n a= %d", *((int*)vp));
vp=&b;
printf("\n a= %d", *((double *)vp));
return 0;
}
```

Output:

a= 5

b= 3.141500

Null Pointer

- A **null pointer** is a special pointer that points nowhere. That is, no other valid pointer to any other variable or array cell or anything else will ever be equal to a null pointer.

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

- 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;
- NULL is a constant that is defined in the standard library and is the equivalent of zero for a pointer. NULL is a value that is guaranteed not to point to any location in memory.

Use of Pointers

- ***Call by address:***
- One of the typical applications of pointers is to support call by reference. However, C does not support call by reference as do other programming languages such as PASCAL and FORTRAN.
  - Typically a function call is made to communicate some arguments to the function.
  - C makes use of only one mechanism to communicate arguments to a function: *call by value*.
  - This means that when a function is called, a copy of the values of the arguments is created and given to the function.

Example

```
#include <stdio.h>
void swap(int a, int b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

```
int main()
{
int x=5,y=10;
void swap(int,int);
printf("%d %d\n",x,y);
swap(x,y);
printf("%d %d\n",x,y);
return 0;
}
```

Output:

5 10

5 10 No swapping

Call by Address

- Call by reference does not exist in C, but it can be simulated through the use of pointers.
- To make a function be able to modify a certain variable, the function must be provided with information about the location of the variable in memory (i.e., its address).
- If the function knows where the variable is in memory, it will be able to access that area of memory by using pointers and change its content. This is known as *call by address.*

```
#include <stdio.h>
void swap(int *a, int *b)
{
int temp;
temp = *a;
*a = *b;
*b = temp;
}
int main()
{
int x=5,y=10;
void swap(int *,int *);
printf("%d %d\n",x,y);
swap(&x, &y);
printf("%d %d\n",x,y);
return 0;
}
```

Output:

```
5 10
10 5
```

Returning More Than One Value from a Function

- Functions usually return only one value and when arguments are passed by value, the called function cannot alter the values passed and have those changes reflected in the calling function.
- Pointers allow the programmer to 'return' more than one value by allowing the arguments to be passed by address, which allows the function to alter the values pointed to, and thus 'return' more than one value from a function.

```
#include <stdio.h>
int main()
{
float r, area, perimeter;
float compute(fl oat, fl oat *);
printf("\n enter the radius of the
circle:");
scanf("%f",&r);
area=compute(r, &perimeter);
printf("\n AREA = %f", area);
printf("\n PERIMETER = %f",
perimeter);
return 0;
}
float compute(fl oat r, fl oat *p)
{
fl oat a;
a=(fl oat)3.1415 * r * r;
*p=(fl oat)3.1415 * 2 * r;
return a;
}
```

Returning Pointer from a Function

- It is also possible to return a pointer from a function.

When a pointer is returned from a function, it must point to data in the calling function or in the global variable. Consider the following program. In this program, a pointer would point an integer variable whichever is larger between two variables through a function which returns the address of the larger variable.

```
#include <stdio.h>
int *pointMax(int *, int *);
int main(void)
{
int a,b,*p;
printf("\n a = ?");
scanf("%d",&a);
printf("\n b = ?");
scanf("%d",&b);
p=pointMax(&a,&b);
printf("\n*p = %d", *p);
return 0;
}
int *pointMax(int *x, int *y)
{
if(*x>*y)
return x;
else
return y;
}
```

Output:

```
a = ?5
b = ?7
*p = 7
```

Arrays and Pointers

- **One-dimensional Arrays and Pointers**
- An array is a non-empty set of sequentially indexed elements having the same type of data.
- Each element of an array has a unique identifying index number. Changes made to one element of an array does not affect the other elements.
- An array occupies a contiguous block of memory. The array a is laid out in memory as a contiguous block, as shown.

```
int a[]={10, 20, 30, 40, 50};
```

a[0]	a[1]	a[2]	a[3]	a[4]
10	20	30	40	50
2147478270	2147478274	2147478278	2147478282	2147478286

One-dimensional Arrays and Pointers

- Array name is an *pointer constant*. It cannot be used as *lvalue*.
- That is array names cannot be used as variables on the left of an assignment operator.
- Both array and &array would give the base address of the array, but the only difference is under ANSI/ISO Standard C, &array yields a pointer, of type pointer to- array of-the data type to the entire array.

- **Example**

```
#include <stdio.h>
int main()
{
int array[]={10, 20, 30, 40, 50};
printf("%u %u", array, &array[0]);
return 0;
}
```

Output:

2147478270 2147478270

One-dimensional Arrays and Pointers

- A pointer variable (of the appropriate type) can also be used to initialize or point to the first element of the array.

```
#include <stdio.h>
int main()
{
  int a[]={10, 20, 30, 40, 50};
  int i, *p;
  p=a; /* it can also be written as p=&a[0]; */
  for(i=0;i<5;++i)
    printf("\n%d", p[i]);

  return 0;
}
```

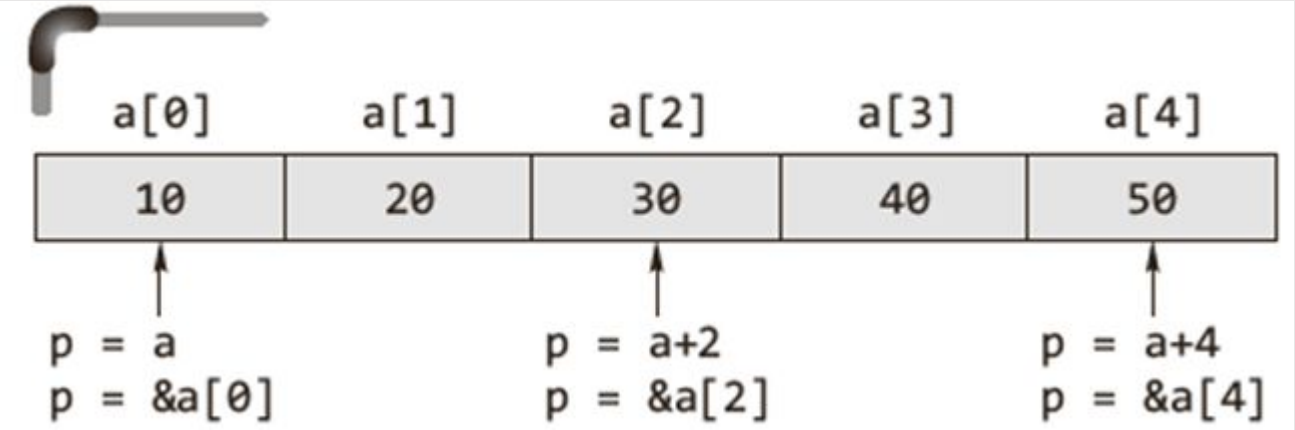
```
printf ("\n%d", *(p+i));
          OR
printf ("\n%d", *(i+p));
          OR
printf ("\n%d", i[p]);
```

Output:

```
10
20
30
40
50
```

One-dimensional Arrays and Pointers

The figure below depicts the equivalence among array notation and pointer notation.



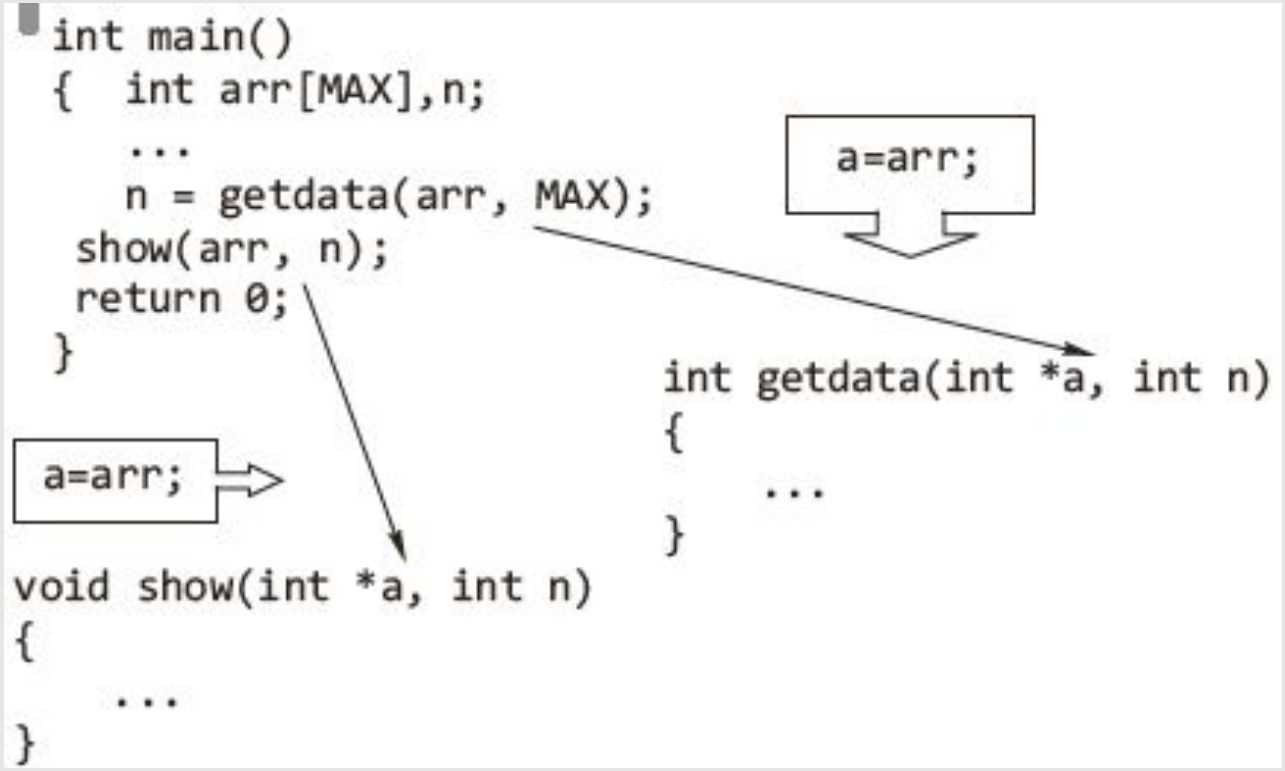
Pointer notation of array elements

For any one-dimensional array a and integer i, the following relationships are always true.

- a[i] ≡ *(a+i) ≡ *(i+a) ≡ i[a]

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.



Passing an Array to a Function

- When an array is passed to a function, it degenerates to a pointer.
- All array names that are function parameters are always converted into pointers by the compiler.
- Because when passing an array to a function, the address of the zero-th element of the array is copied to the pointer variable which is the formal parameter of the function.
- However, arrays and pointers are processed differently by the compiler, represented differently at runtime.

Differences between Array Name and Pointer

- When memory is allocated for the array, the starting address is fixed, i.e., it cannot be changed during program execution. Therefore, array name is an address constant.
- The & (address of) operator normally returns the address of the operand. However, arrays are the exception.
  - When applied to an array (which is an address), it has the same value as the array reference without the operator.
  - This is not true of the equivalent pointers, which have an independent address.
- The sizeof operator returns the size of the allocated space for arrays.
  - In case of a pointer, the sizeof operator returns two or four or more bytes of storage (machine dependent).

Differences between Pointer and Arrays

Arrays	Pointers
• Array allocates space automatically.	• It is explicitly assigned to point to an allocated space.
• It cannot be resized.	• It can be resized using `realloc()`.
• It cannot be reassigned.	• It can be reassigned.
• `sizeof(arrayname)` gives the number of bytes occupied by the array.	• `sizeof(p)` returns the number of bytes used to store the pointer variable p.

Pointers and Strings

- Strings are one-dimensional arrays of type char. By convention, a string in C is terminated by the end-of-string sentinel \0, or null character.
- A string constant is treated by the compiler as a pointer.
- For a string, the number of elements it has need not be passed to a function because it has the terminating null character.

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.

Pointer Arithmetic

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.
- Subtracting or comparing two pointers (within array limits) that point to the elements of an array.
- Pointers with the assignment operators can be used if the following conditions are met.
  - The left-hand operand is a pointer and the right-hand operand is a null pointer constant.
  - One operand is a pointer to an object of incompatible type and the other is a pointer to void.
  - Both the operands are pointers to compatible types.

Pointer Arithmetic

The following arithmetic operations on pointers are not feasible.

- Addition of two pointers
- Multiplying a pointer with a number
- Dividing a pointer with a number

Pointer Arithmetic

- Incrementing or decrementing the pointers (within array limits) 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 nowhere at all.

Pointer Arithmetic

- These valid address arithmetic are discussed below in detail. 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

Pointer Arithmetic

Operation	Condition	Example	Result

Assignment	Pointers must be of same type	`int *p,*q`
		`...`
		`p = q;`
			p points to whatever q points to

Addition of an integer		`int k,*p;`
		`...`
		`p + k`
			Address of the kth object after the one p points to

Subtraction of an integer		`int k,*p;`
		`...`
		`p - k`
			Address of the kth object before the one p points to

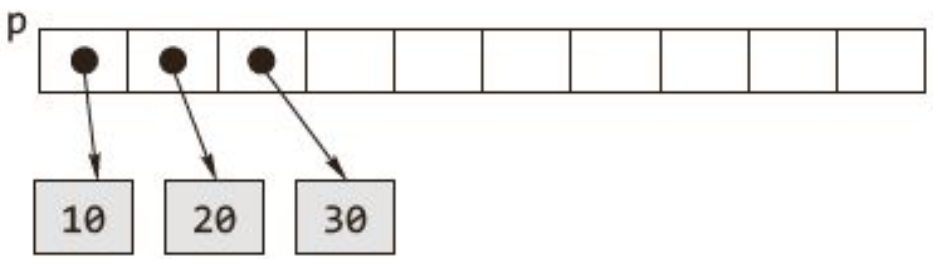
Comparison of pointers	Pointers pointing to the members of the same array	`int *p,*q;`
		`...`
		`q < p`
			Returns true (1) if q points to an earlier element of the array than p does. Return type is int

Subtraction of pointers	Pointers to members of the same array and q < p	`int *p,*q;`
		`...`
		`p - q`
			Number of elements between p & q;

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.

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

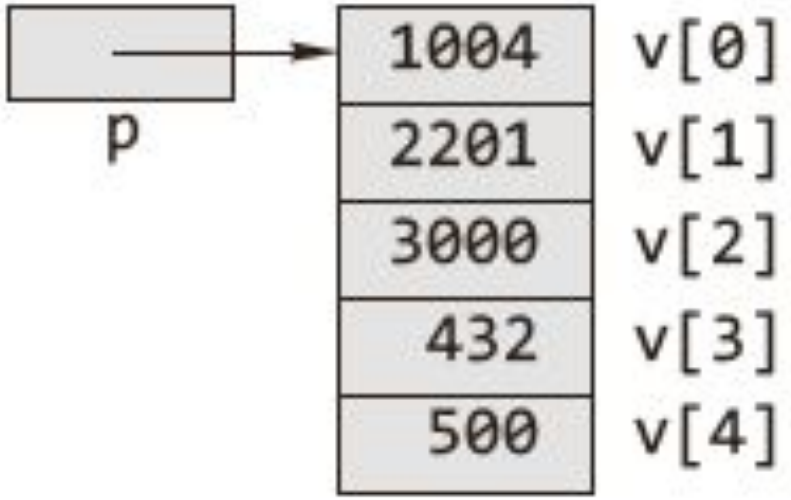


Pointers to an Array

- We can declare a pointer to a simple integer value and make it point to the array as is done normally.

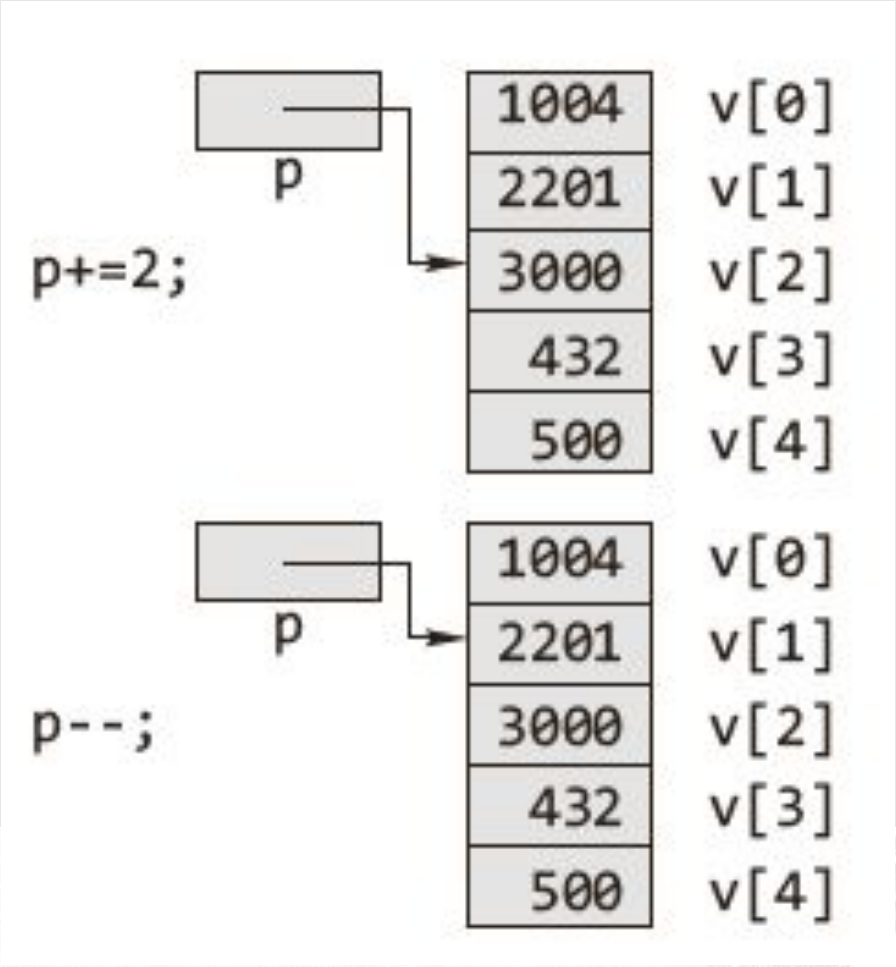
```
int v[5] = {1004, 2201, 3000, 432, 500};
int *p = v;
printf("%d \n", *p);
```

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

- We can use instructions such as += and -= to refer to different elements in the array.

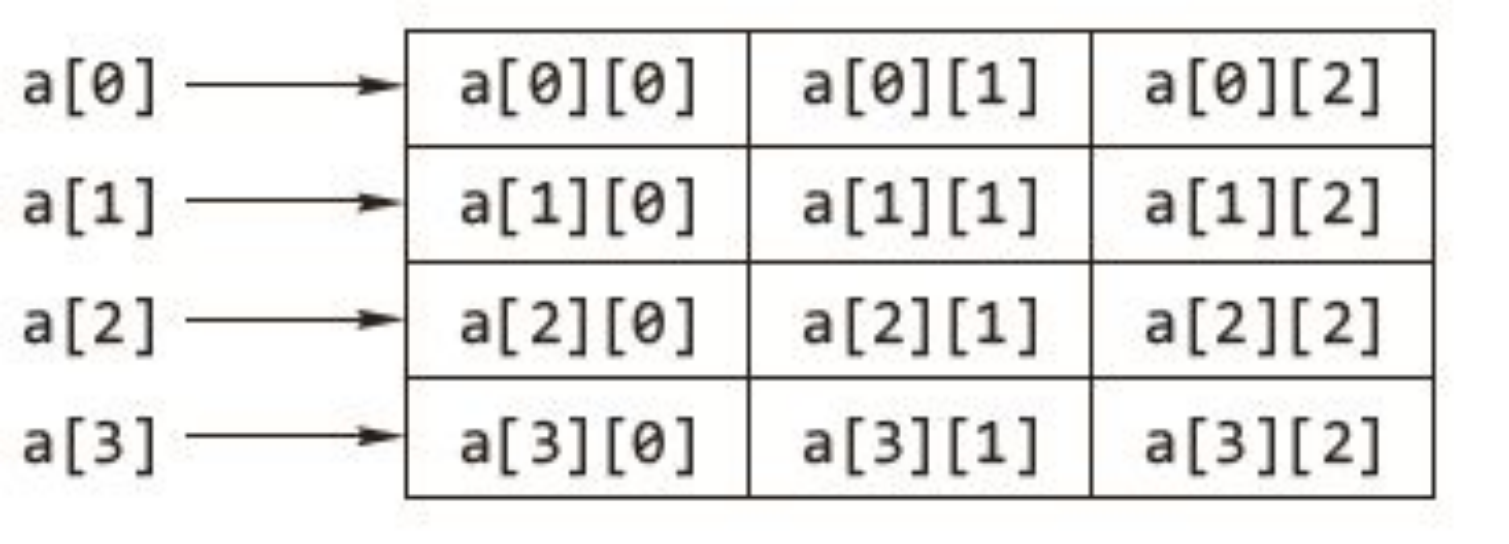


Difference between an Array of Pointers and a Pointer to an Array

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

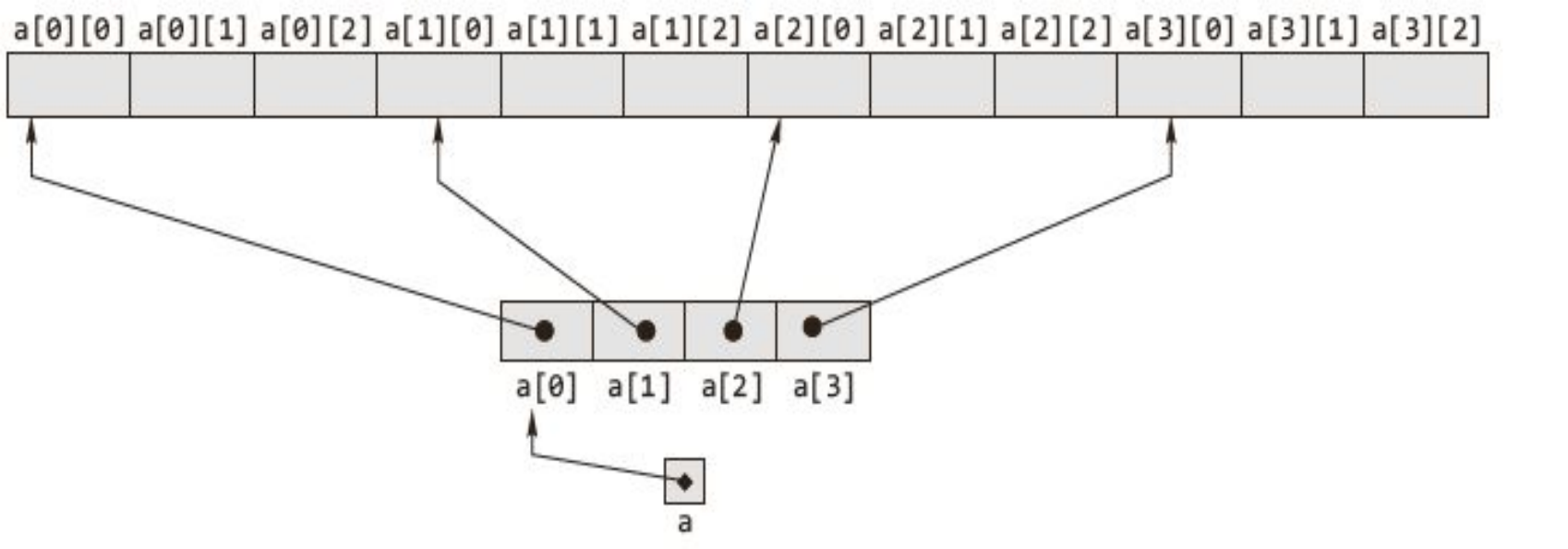
Two-dimensional Arrays and Pointers

- A two-dimensional array in C is treated as a one dimensional array whose elements are one-dimensional arrays (the rows).



Two-dimensional Arrays and Pointers

- Physical representation of a two-dimensional array



Two-dimensional Arrays and Pointers

- C does not do run-time range checking of array subscripts.
- In C the rightmost subscript of a two-dimensional array varies faster than the leftmost.
- Multi-dimensional array is stored in a ‘row major addressing’ format.
- The following expressions are equivalent for a two dimensional array
  - a[i][j]= *(a[i]+ j) = (*(a + i))[j] = *(*(a + i) + j)

Addressing

Address of A[I] = B + W * (I – LB)

I = Subset of element whose address to be found,

B = Base address,

W = Storage size of one element store in any array(in byte),

LB = Lower Limit/Lower Bound of subscript(If not specified assume zero).

Example: Given the base address of an array A[1300 …… 1900] as 1020 and the size of each element is 2 bytes in the memory, find the address of A[1700].

Example: Given the base address of an array A[1300 …… 1900] as 1020 and the size of each element is 2 bytes in the memory, find the address of A[1700].

Base address B = 1020
Lower Limit/Lower Bound of subscript LB = 1300
Storage size of one element store in any array W = 2 Byte
Subset of element whose address to be found I = 1700

Formula used:
Address of A[I] = B + W * (I – LB)

Solution:
Address of A[1700] = 1020 + 2 * (1700 – 1300)
= 1020 + 2 * (400)
= 1020 + 800
Address of A[1700] = 1820

2D

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Example: Given an array, arr[1……10][1……15] with base value 100 and the size of each element is 1 Byte in memory. Find the address of arr[8][6] with the help of row-major order.

Solution:

Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15

Formula:
Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

Solution:
Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))
= 100 + 1 * ((7) * 15 + (5))
= 100 + 1 * (110)
Address of A[I][J] = 210

3D

To find the address of the element using row-major order, use the following formula:

Address of A[i][j][k] = B + W *(M * N(i-x) + N *(j-y) + (k-z))

Here:

B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array, arr[1:9, -4:1, 5:10] with a base value of 400 and the size of each element is 2 Bytes in memory find the address of element arr[5][-1][8] with the help of row-major order?

Solution:

Given:
Row Subset of an element whose address to be found I = 5
Column Subset of an element whose address to be found J = -1
Block Subset of an element whose address to be found K = 8
Base address B = 400
Storage size of one element store in any array(in Byte) W = 2
Lower Limit of row/start row index of matrix x = 1
Lower Limit of column/start column index of matrix y = -4
Lower Limit of blocks in matrix z = 5
M = Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6
N = Upper Bound – Lower Bound + 1 = 10 – 5 + 1 = 6

Formula used:
Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))

Solution:
Address of arr[5][-1][8] = 400 + 2 * {[6 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]
= 400 + 2 * ((4 * 6 + 3) * 6 + 3)
= 400 + 2 * (165)
= 730

Two-dimensional Arrays and Pointers

- Arrays of dimension higher than two work in a similar fashion.
  - Let us describe how three-dimensional arrays work.
  - If the following is declared int a[7][9][2];
- In case of multi-dimensional arrays all sizes except the first must be specified.
- ***Caution*** These three declarations are equivalent only in a header to a function definition.

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, ...);

- Initialization of function pointers:

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

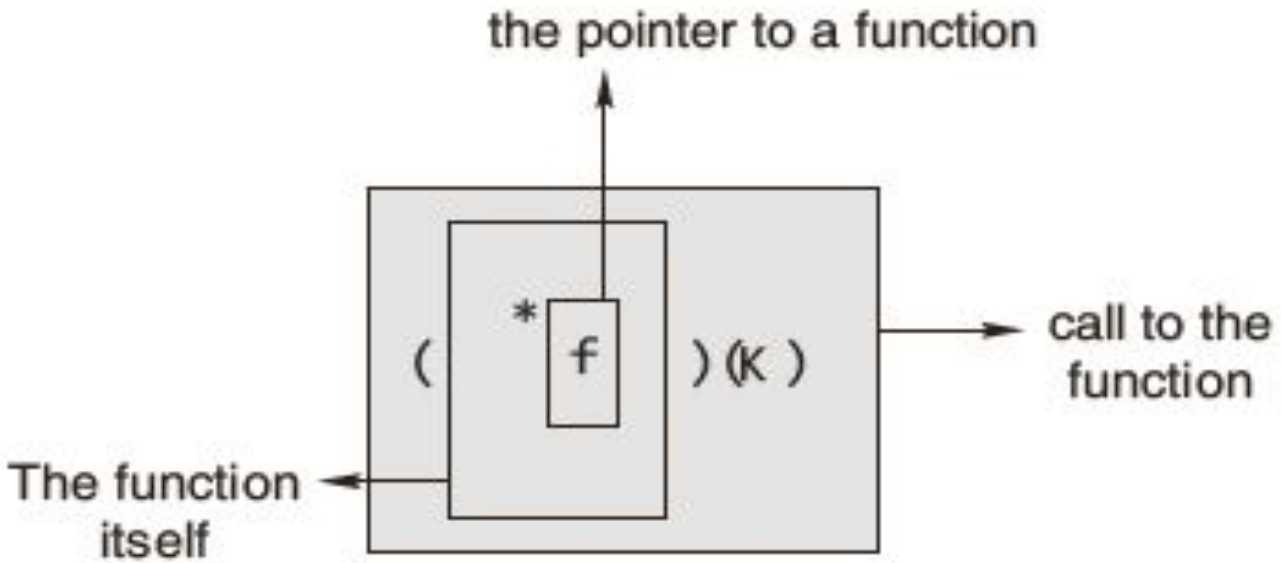
- Calling a function using a function pointer

result1 = fpointer(4, 5);

result2 = fpointer(6, 2);

- Passing a Function to another Function

A function pointer can be passed as a function's calling argument.



Pointers to Functions

- How to Return a Function Pointer

```
float(*GetPtr1(char opCode))(float, float)
{
if(opCode == '+') return &Add;
if(opCode == '-') return &Sub;
}
```

A solution using a typedef defines a pointer to a function that takes two float values and returns a float.

Example

- **Arrays of Function Pointers**:

```
int main()
{
void(*p[3])(int, int);
int i;
void Add(int, int);
void Sub(int, int);
void Mul(int, int);
p[0] = Add;
p[1] = Sub;
p[2] = Mul;
for(i = 0; i <= 2; i++)
(*p[i])(10, 5);
return 0;
}
```

```
void Add(int a, int b)
{
printf("\n Result of Addition = %d",a+b);
}
void Sub(int a, int b)
{
printf("\n Result of Subtraction = %d",a-b);
}
void Mul(int a, int b)
{
printf("\n Result of Multiplication =
%d",a*b);
}
```

Dynamic Memory Allocation

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).

Dynamic Memory Allocation

- Where fixed arrays are used, static memory allocation, or memory allocated at compile time, is used.
- 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 deallocate dynamic heap memory.

Dynamic Memory Allocation

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 may be 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.

Dynamic Memory Allocation

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>.

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.

Dynamic Memory Allocation

calloc():

- It 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.

Dynamic Memory Allocation

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.

To use these functions, either stdlib.h or alloc.h must be included as these functions are declared in these header files.

Dynamic Memory Allocation

- All of a program's memory is deallocated 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.

Dynamic Memory Allocation

- 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*/
}
```

Dynamic Memory Allocation

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);
```

Note that we need to include stdlib.h to use this function.

Dynamic Memory Allocation

- When memory is allocated, the allocating function (such as malloc() and calloc()) returns a pointer.
- The type of this pointer depends on whether one using an older K&R compiler or the newer ANSI type compiler.
- 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’.
- Although this is not mandatory in ANSI/ISO C, but it is recommended for portability of the code.

Dynamic Memory Allocation

- In dynamic memory allocation, memory is allocated at runtime heap.
  - According to ANSI compiler, the block pointer returned by allocating function is a void pointer.
  - If sufficient memory is not available, the malloc()and calloc() returns a NULL.
  - According to ANSI compiler, a cast on the void pointer returned by malloc() is not required.

Dynamic Memory Allocation

- 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);
- When dynamically allocated, arrays are no longer needed, it is recommended to free them immediately.

Dynamic Memory Allocation

- **Dynamic Allocation of Arrays:**

To allocate a one-dimensional array of length N of some particular type where N is given by the user, simply use malloc() to allocate enough memory to hold N elements of the particular type.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int N,*a,i,s=0;
printf("\n enter no. of elements of the array:");
scanf("%d",&N);
a=(int *)malloc(N*sizeof(int));
if(a==NULL)
{
printf("\n memory allocation unsuccessful...");
exit(0);
}
printf("\n enter the array elements
    one by one");
for(i=0; i<N;++i)
{
scanf("%d",&a[i])); /* equivalent
    statement
scanf("%d",(a+i));*/
s+=a[i];
}
printf("\n sum is %d ",s);
return 0;
}
```

Dynamic Memory Allocation

- **Freeing Memory :**
- 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); will ‘give the memory back’ to the stock of memory
- malloc() requires two parameters, the first for the number of elements to be allocated and the second for the size of each element, whereas calloc() requires one parameter.

Dynamic Memory Allocation

void *memset(void *s, int c, size_t n);

memset also sets the first n bytes of the array s to the character c.

```
#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;
}
```

Dynamic Memory Allocation

- 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.

Reallocating Memory Blocks

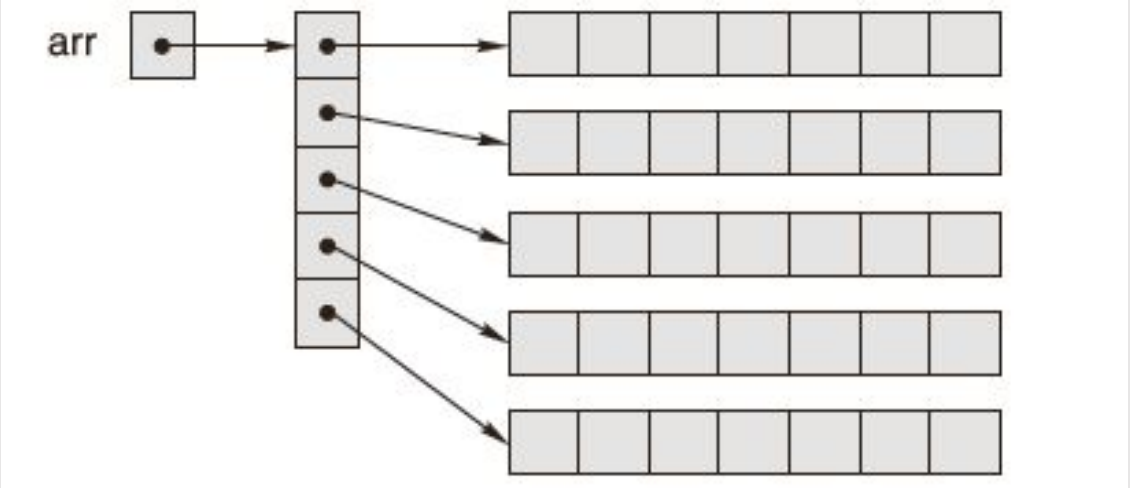
malloc() will not work. It is the realloc() function that is required

ip = realloc(ip, 200 * sizeof(int));

Dynamic Memory Allocation

Implementing Multidimensional Arrays using Pointers:

It is usually best to allocate an array of pointers, and then initialize each pointer to a dynamically allocated ‘row’.



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.

- First, 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.
- Second, 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.
  - second type is usually the harder to find.
  - 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().

Memory Leak and Memory Corruption

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

- This function will do everything it is meant to. The only problem is that every time this function is called, it allocates a small bit of memory and never gives it back.
- if it is called often, then it will gradually use all the memory in the computer.

Memory Leak and Memory Corruption

Dangling pointer

- 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.

```
char *a = malloc(128*sizeof(char));
char *b = malloc(128*sizeof(char));
b = a;
free(a);
free(b);
```

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.

- ***Buffer 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

- ***Buffer overflow***

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
```

Memory Leak and Memory Corruption

- *Using an address before memory is allocated and set*

```
int *ptr;
ptr=5;
```

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

- *Using a pointer which is already freed*

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

Memory Leak and Memory Corruption

- *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

- *Freeing memory which was not dynamically allocated*

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

Pointer and Const Qualifier

- **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.

```
int n = 10;
const int *ptr=&n;
*p = 100; /* ERROR */
n = 50;
int v = 100;
ptr = &v; /* OK - changing the address in ptr */
```

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 */
int v = 5;
ptr = &v;/* Error - attempt to change a constant pointer */
*ptr = 100;/* OK - changes the value of v */
```

You can create a constant pointer that points to a value that is also constant:

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

Pointer and Const Qualifier

- **Constant parameters**

To prevent an array argument from being altered in a function, use the const qualifier as demonstrated in the next program.

```
#include <stdio.h>
void change(char *);
int main(void)
{
char s[]="Siva";
change(s);
printf("\n The string after
calling change(): %s", s);
return 0;
}
```

```
void change(char *t)
{
   *t= 'V';
}
```

```
void change(const char *t)
{
   *t= 'V'; //Error
}
```

Thank You!