UNIT-1 Pointers and File Handling

Course- Fundamentals of Data Structures

Mrs. S.G. Dedgaonkar
suruchi.Dedgaonkar@viit.ac.in

Department of Information Technology Engineering, VIIT, Pune-48

BRACT'S, Vishwakarma Institute of Information Technology, Pune-48

(An Autonomous Institute affiliated to Savitribai Phule Pune University)
(NBA and NAAC accredited, ISO 9001:2015 certified)

Last Session Recap

Bridge Course

1. C Fundamentals: Loops, Functions etc.
2. Arrays
3. Structures

Objective/s of this session

1. To understand pointer fundamentals
2. To do pointer arithmetic

Learning Outcome/Course Outcome

1. Able to use pointers.
2. Able to perform pointer operations.

Content

Part –I

- Introduction to Pointers
- Dynamic memory allocation
- Pointer to pointer

Part-II

- Pointer to single and multidimensional arrays
- Array of pointers, String and structure manipulation using pointers
- Pointer to functions

Part-III

- Pointer to file structure
- basic operations on file
- File handling in C

UNIT I POINTERS AND FILE HANDLING

Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

What is pointer?

In a generic sense, a “pointer” is anything that tells us where something can be found.

- Addresses in the phone book
- URLs for webpages
- Road signs
- Page number in index

A pointer can contain the memory address of any variable type

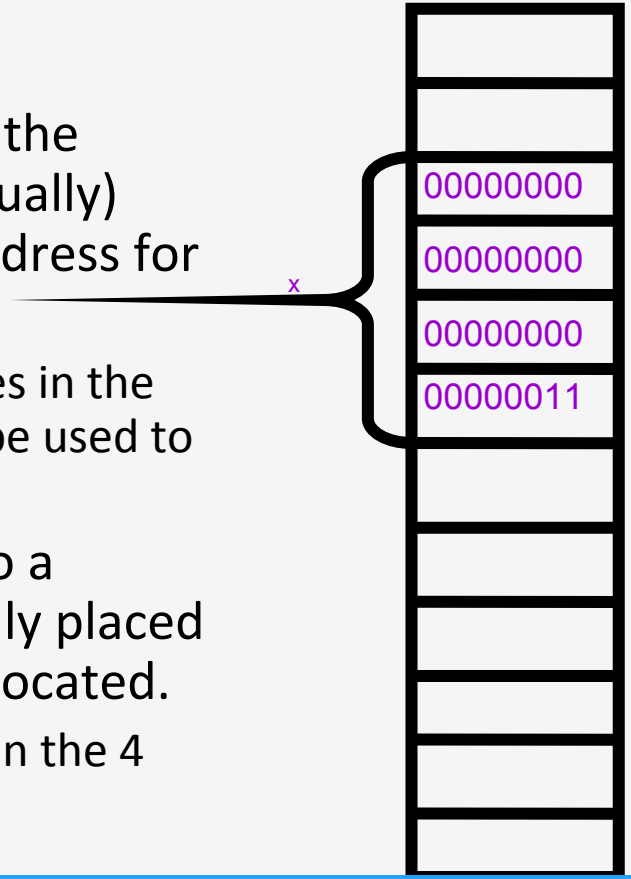
- A primitive (int, char, float)
- An array
- A struct or union
- Dynamically allocated memory
- Another pointer
- A function

Why Pointers?

- They allow you to refer to large data structures in a compact way
  - Eg- array, database (array of structure elements)
- They facilitate sharing between different parts of programs
  - Call by reference in functions
- They make it possible to get new memory dynamically as your program is running
  - Dynamic memory allocation
- They make it easy to represent relationships among data items.
  - Complex data structures like linked list, stack, queue, tree, graph etc.

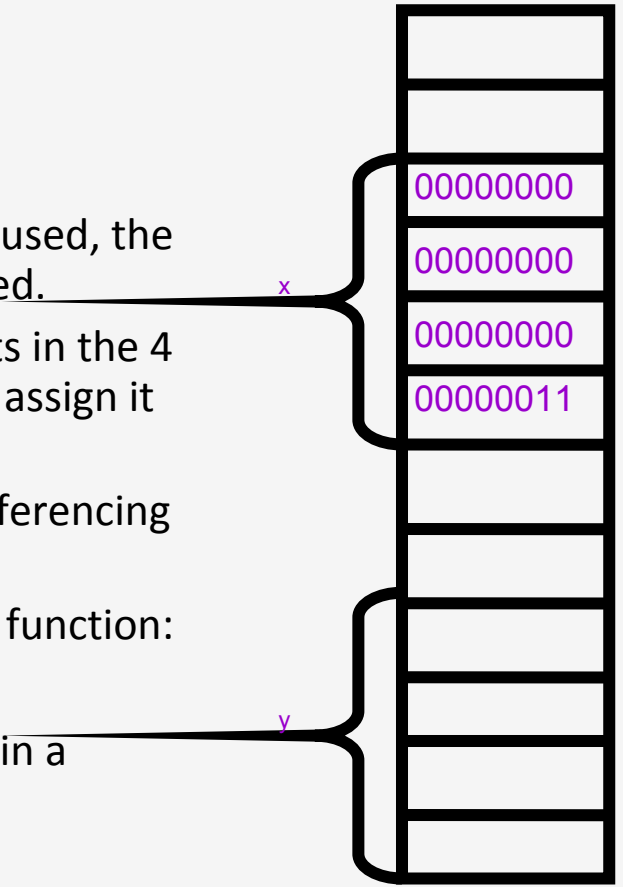
1. Pointer Fundamentals

- When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.
  - int x; will allocate 4 bytes in the main memory, which will be used to store an integer value.
- When a value is assigned to a variable, the value is actually placed to the memory that was allocated.
  - x=3; will store integer 3 in the 4 bytes of memory.



Pointers

- When the value of a variable is used, the contents in the memory are used.
  - y=x; will read the contents in the 4 bytes of memory, and then assign it to variable y.
- &x can get the address of x. (referencing operator &)
- The address can be passed to a function:
  - scanf("%d", &x);
- The address can also be stored in a variable ……



Pointers

- To declare a pointer variable

  type * pointername;

- For example:
  - int * p1; p1 is a variable that tends to point to an integer, (or p1 is a int pointer)
  - char *p2;
  - unsigned int * p3;
- p1 = &x; /* Store the address in p1 */
- scanf("%d", p1); /* i.e. scanf("%d",&x); */
- p2 = &x; /* Will get warning message */

PRIMARY OPERATORS

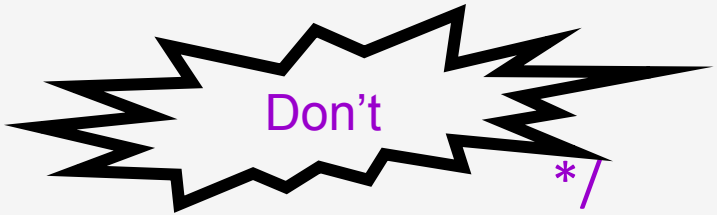
- When is & used?
  - & -- "address operator" which gives or produces the memory address of a data variable

- When is * used?
  - -- "dereferencing operator" which provides the contents in the memory location specified by a pointer

Initializing Pointers

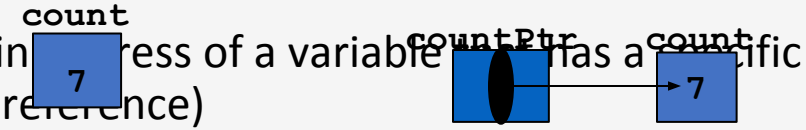
- Like other variables, always initialize pointers before using them!!!
- For example:

```
int main(){
   int x;
   int *p;
   scanf("%d",p);   /*                    */
   p = &x;
   scanf("%d",p);  /* Correct */
}
```



Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)
  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



Pointer Variable Declarations and Initialization

- Pointer declarations
  - ***** used with pointer variables

```
int *myPtr;
```

  - Declares a pointer to an **int** (pointer of type **int ***)
  - Multiple pointers require using a ***** before each variable declaration

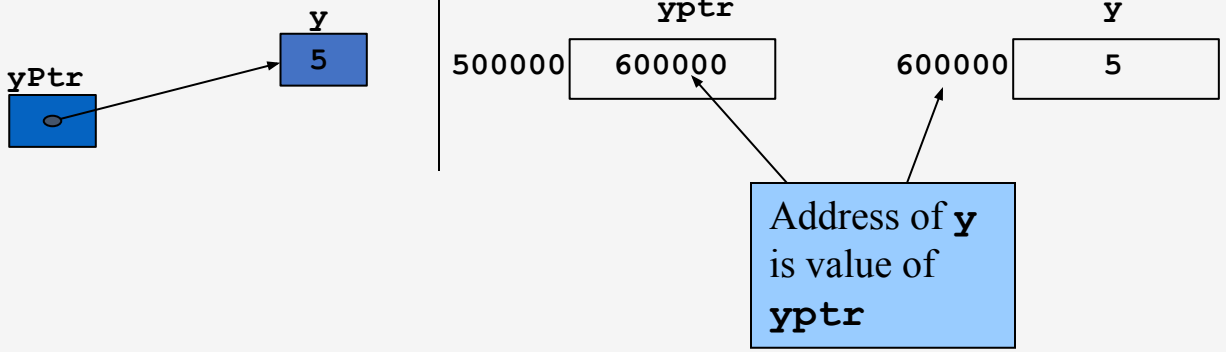
```
int *myPtr1, *myPtr2;
```

  - Can declare pointers to any data type
  - Initialize pointers to **0**, **NULL**, or an address
    - **0** or **NULL** – points to nothing (**NULL** preferred)

Pointer Operators

- **&** (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
yPtr "points to" y
```



Pointer Operators

- **`*`** (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - **`*yptr`** returns **`y`** (because **`yptr`** points to **`y`**)
  - **`*`** can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  // changes y to 7
```

- **`*`** and **`&`** are inverses
  - They cancel each other out

Computer Memory Revisited

- Computers store data in memory slots
- Each slot has an *unique address*
- Variables store their values like this:

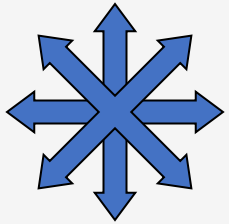
Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 37	1002	j: 46	1004	k: 58	1006	m: 74
1008	a[0]: 'a'	1009	a[1]: 'b'	1010	a[2]: 'c'	1011	a[3]: '\0'
1012	ptr: 1002	1014	…	…	…	….	….

Computer Memory Revisited

- Altering the value of a variable is indeed changing the content of the memory
  - e.g. i = 40; a[2] = ‘z’;

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1002	j: 46	1004	k: 58	1006	m: 74
1008	a[0]: ‘a’	1009	a[1]: ‘b’	1010	a[2]: ‘z’	1011	a[3]: ‘\0’
1012	ptr: 1002	1013	…	…	…	…	…

Addressing Concept



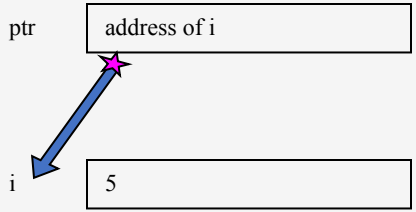
- Pointer stores the **address** of another entity
- It **refers** to a memory location

```
int i = 5;
int *ptr;                          /* declare a pointer variable */
ptr = &i;                          /* store address-of i to ptr */
printf("*ptr = %d\n", *ptr);       /* refer to referee of ptr */
```

What actually *ptr* is?

- **ptr** is a variable storing **an address**
- ptr is **NOT** storing the actual value of i

```
int i = 5;
int *ptr;
ptr = &i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```



Output:

```
i = 5
*ptr = 5
ptr = effff5e0
```

value of ptr = address of i in memory

Twin Operators

- &: Address-of operator
  - Get the *address* of an entity
    - e.g. `ptr = &j;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1002	j: 33	1004	k: 58	1006	m: 74
1008	ptr: 1002						

Twin Operators

- *: De-reference operator
  - Refer to the *content* of the referee
    - e.g. `*ptr = 99;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1002	j: 99	1004	k: 58	1006	m: 74
1008	ptr: 1002						

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table

Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

An Illustration

```
int i = 5, j = 10;
int *ptr;    /* declare a pointer-to-integer variable */
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;  /* declare a pointer-to-pointer-to-integer variable */
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	
pptr	int **	integer pointer pointer variable	
		Double Indirection	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;     /* store address-of i to ptr */
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	
*ptr	int	de-reference of ptr	5

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr; /* store address-of ptr to pptr */
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	3
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	3

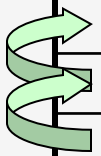
An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	7

An Illustration

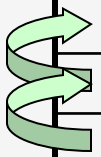
```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	10

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```



Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	9

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	-2
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	-2

Pointer Arithmetic

- What's `ptr + 1`?

 The next memory location!

- What's `ptr - 1`?

 The previous memory location!

- What's `ptr * 2` and `ptr / 2`?

 Invalid operations!!!

More Pointer Arithmetic

- What if `a` is a `float` array?
- A `float` *may* occupy more memory slots!
  - Given **`float *ptr = a;`**
  - What's **`ptr + 1`** then?

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…
1008	a[2]: 3.14	1009	…	1010	…	1011	…

More Pointer Arithmetic

- Arithmetic operators + and – *auto-adjust* the address offset
- According to the *type* of the pointer:
  - 1000 + sizeof(float) = 1000 + 4 = 1004

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	a[0]: 37.9	1001	…	1002	…	1003	…
1004	a[1]: 1.23	1005	…	1006	…	1007	…
1008	a[2]: 3.14	1009	…	1010	…	1011	…

Advice and Precaution

- Pros
  - Efficiency
  - Convenience
- Cons
  - Error-prone
  - Difficult to debug

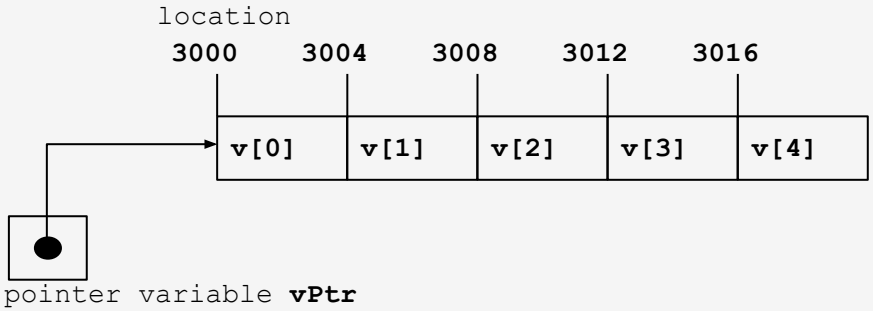
2. Pointer arithmetic

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or −−)
  - Add an integer to a pointer(+ or += , − or −=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer Expressions and Pointer Arithmetic

- 5 element **`int`** array on machine with 4 byte **`int`**s
  - **`vPtr`** points to first element **`v[ 0 ]`**
    - at location **`3000`** (**`vPtr = 3000`**)
  - **`vPtr += 2;`** sets **`vPtr`** to **`3008`**
    - **`vPtr`** points to **`v[ 2 ]`** (incremented by 2), but the machine has 4 byte **`int`**s, so it points to address **`3008`**



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
```

  - **`vPtr2 - vPtr`** would produce 2
- Pointer comparison (**<**, **==** , **>**)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to **0**

Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

Using the `const` Qualifier with Pointers

- **`const`** qualifier
  - Variable cannot be changed
  - Use **`const`** if function does not need to change a variable
  - Attempting to change a **`const`** variable produces an error
- **`const`** pointers
  - Point to a constant memory location
  - Must be initialized when declared
  - **`int *const myPtr = &x;`**
    - Type **`int *const`** – constant pointer to an **`int`**
  - **`const int *myPtr = &x;`**
    - Regular pointer to a **`const int`**
  - **`const int *const Ptr = &x;`**
    - **`const`** pointer to a **`const int`**
    - **`x`** and *Ptr can not be changed

```
1   /* Fig. 7.13: fig07_13.c
2      Attempting to modify a constant pointer to
3      non-constant data */
4
5   #include <stdio.h>
6
7   int main()
8   {
9      int x, y;
10
11     int * const ptr = &x; /* ptr is a constant
12                              integer. An integer
13                              through ptr, but ptr
14                              to the same memory
15     *ptr = 7;
16     ptr = &y;
17
18     return 0;
19  }
```

Changing ***ptr** is allowed – **x** is not a constant.

Changing **ptr** is an error – **ptr** is a constant pointer.

```
FIG07_13.c:
Error E2024 FIG07_13.c 16: Cannot modify a const object in
function main
*** 1 errors in Compile ***
```

3. Pointers and arrays

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array **`b[ 5 ]`** and a pointer **`bPtr`**
  - To set them equal to one another use:

```
bPtr = b;
```

    - The array name (**`b`**) is actually the address of first element of the array **`b[ 5 ]`**

```
bPtr = &b[ 0 ]
```

    - Explicitly assigns **`bPtr`** to address of first element of **`b`**

```
1   /* Fig. 7.7: fig
2      Cube a variab
3      with a pointe
4
5   #include <stdio.h>
6
7   void cubeByReference(
8
9   int main()
10  {
11     int number = 5;
12
13     printf( "The original value of number is %d", number );
14     cubeByReference( &number );
15     printf( "\nThe new value of number is %d\n", number );
16
17     return 0;
18  }
19
20  void cubeByReference( int *nPtr )
21  {
22     *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
23  }
```

Notice that the function prototype takes a pointer to an integer (**int ***).

Notice how the address of **number** is given - **cubeByReference** expects a pointer (an address of a variable).

Inside **cubeByReference**, ***nPtr** is used (***nPtr** is **number**).

```
The original value of number is 5
The new value of number is 125
```

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	?
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	?
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	?

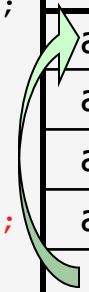
Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[3]
*ptr	float	de-reference of float pointer variable	9.0

Pointer Arithmetic and Array

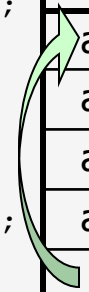
```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	?
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	?

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```



Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[0]
*ptr	float	de-reference of float pointer variable	6.0

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	3.14
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	3.14

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

Data Table			
Name	Type	Description	Value
a[0]	float	float array element (variable)	6.0
a[1]	float	float array element (variable)	?
a[2]	float	float array element (variable)	7.0
a[3]	float	float array element (variable)	9.0
ptr	float *	float pointer variable	address of a[2]
*ptr	float	de-reference of float pointer variable	7.0

Pointer Arithmetic and Array

```
float a[4];
float *ptr;
ptr = &(a[2]);
*ptr = 3.14;
ptr++;
*ptr = 9.0;
ptr = ptr - 3;
*ptr = 6.0;
ptr += 2;
*ptr = 7.0;
```

- Type of `a` is `float *`
- `a[2]` ≡ `*(a + 2)`
  - `ptr = &(a[2])`
  - `ptr = &(*(a + 2))`
  - `ptr = a + 2`
- `a` is a memory address *constant*
- `ptr` is a pointer *variable*

The Relationship Between Pointers and Arrays

- Element **`b[ 3 ]`**
  - Can be accessed by **`*( bPtr + 3 )`**
    - Where **`n`** is the offset. Called pointer/offset notation
  - Can be accessed by **`bptr[ 3 ]`**
    - Called pointer/subscript notation
    - **`bPtr[ 3 ]`** same as **`b[ 3 ]`**
  - Can be accessed by performing pointer arithmetic on the array itself

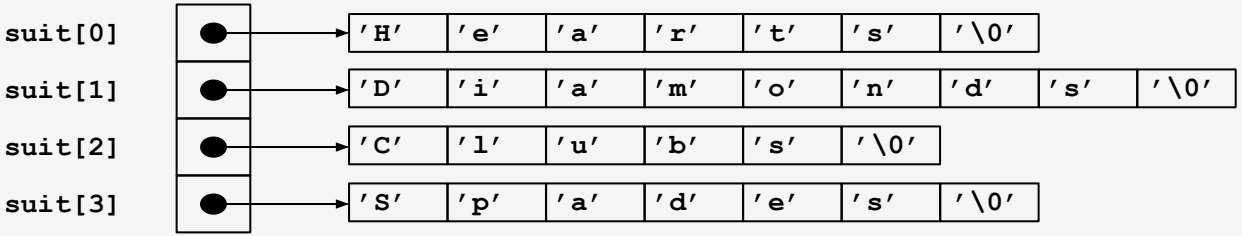
    `*( b + 3 )`

Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - **`char *`** – each element of **`suit`** is a pointer to a **`char`**
  - The strings are not actually stored in the array **`suit`**, only pointers to the strings are stored



  - **`suit`** array has a fixed size, but strings can be of any size

4. Functions and pointers

TYPES OF FUNCTION CALLS

- **Call by Value:**

  When a function is called by an argument/parameter which is not a pointer the copy of ***the argument*** is passed to the function. Therefore a possible change on the copy does not change the original value of the argument.

Example

Write a program to calculate and print the area and the perimeter of a circle. Note that theradius is to be entered by the user. (Use ***Call by value*** approach)

```
#include<stdio.h> /*The function calls are Call by Value*/
#define pi 3.14
float area(float);
float perimeter(float);
int main( )
{
float r, a, p;
printf("Enter the radius\n");
scanf("%f",&r);
a = area(r);
p = perimeter(r);
printf("The area = %.2f, \n The Perimeter = %.2f", a, p);
return 0;
}
```

Example

```
float area(float x)
{
return pi*x*x;
}
float perimeter(float y)
{
return 2.0*pi*y;
}
```

TYPES OF FUNCTION CALLS

- **Call by Reference:**

When a function is called by an argument/parameter which is a pointer (address of the argument) the copy of ***the address of the argument*** is passed to the function. Therefore a possible change on the data at the referenced address change the original value of the argument.

Example

Write a program to calculate and print the area and the perimeter of a circle. Note that the radius is to be entered by the user. (Use ***Call by reference*** approach)

```
#include<stdio.h> /*The function calls is Call by Reference*/
#define pi 3.14
void area_perimeter(float, float *, float *);
int main( )
{
float r, a, p;
printf("Enter the radius\n");
scanf("%f",&r);
area_perimeter(r,&a,&p);
printf("The area = %.2f, \n The Perimeter = %.2f", a, p);
return 0;
}
```

Example

```
void area_perimeter(float x, float *aptr, float *pptr);
{
*aptr = pi*x*x;
*pptr = 2.0*pi*x;
}
```

POINTERS AND FUNCTION

- Call by reference
  - If instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference" since we are *referencing* the variables.

- Pointers can be used to pass addresses of variables to called functions
  - Permanent change
  - Multiple outputs from a function

Swapping two numbers

```
#include <stdio.h>
void swap ( int a, int b ) ;
int main ( )
{
   int a = 5, b = 6;
   printf("a=%d b=%d\n",a,b) ;
   swap (a, b) ;
   printf("a=%d b=%d\n",a,b) ;
   return 0 ;
}
```

```
void swap( int a, int b )
{
   int temp;
   temp= a;  a= b;  b = temp ;
   printf ("a=%d  b=%d\n", a, b);
}
```

Results:

a=5 b=6

a=6 b=5

a=5 b=6

Pointers with Functions (example)

```
#include <stdio.h>
void swap ( int *a, int *b ) ;
int main ( )
{
   int a = 5, b = 6;
   printf("a=%d b=%d\n",a,b) ;
   swap (&a, &b) ;
   printf("a=%d b=%d\n",a,b) ;
   return 0 ;
}
```

```
void swap( int *a, int *b )
{
   int temp;
   temp= *a;  *a= *b;  *b = temp ;
   printf ("a=%d  b=%d\n", *a, *b);
}
```

Results:

a=5 b=6

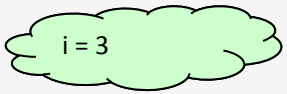
a=6 b=5

a=6 b=5 //permanent change

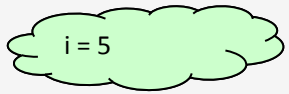
Example: Pass by Reference

- Modify behaviour in argument passing

```
void f(int j)
{
  j = 5;
}
void g()
{
  int i = 3;
  f(i);
}
```




```
void f(int *ptr)
{
  *ptr = 5;
}
void g()
{
  int i = 3;
  f(&i);
}
```



Pointers as Function Parameters

- Sometimes, you want a function to assign a value to a variable. E.g. you want a function that computes the minimum AND maximum numbers in 2 integers.
- Method 1, use two global variables.
  - In the function, assign the minimum and maximum numbers to the two global variables.
  - When the function returns, the calling function can read the minimum and maximum numbers from the two global variables.
- This is bad because the function is not reusable.

Pointers as Function Parameters

- Instead, we use the following function

```
void min_max(int a, int b,
        int *min, int *max){
   if(a>b){
        *max=a;
        *min=b;
   }
    else{
        *max=b;
        *min=a;
   }
```

```
int main()
{
   int x,y;
   int small,big;
   printf("Two integers: ");
   scanf("%d %d", &x, &y);
   min_max(x,y,&small,&big);
   printf("%d <= %d", small, big);
   return 0;
}
```

Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change at actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer

5. Function pointers

What are function Pointers?

- C does not require that pointers only point to data, it is possible to have pointers to functions
- Functions occupy memory locations therefore every function has an address just like each variable

Pointers to Functions

- Pointer to function/ function pointers
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to Functions

- Example: bubblesort
  - Function **`bubble`** takes a function pointer
    - **`bubble`** calls this helper function
    - this determines ascending or descending sorting
  - The argument in **`bubblesort`** for the function pointer:

    ```
    int ( *compare )( int, int )
    ```

    tells bubblesort to expect a pointer to a function that takes two **`int`**s and returns a **`int`**
  - If the parentheses were left out:

    ```
    int *compare( int, int )
    ```

    - Declares a function that receives two integers and returns a pointer to a **`int`**

Why do we need function Pointers?

- Useful when *alternative functions* maybe used to perform similar tasks on data (eg sorting)
- One common use is in *passing a function as a parameter in a function call.*
- Can pass the data and the function to be used to some *control function*
- Greater flexibility and better code reuse

Define a Function Pointer

- A function pointer is nothing else than a variable, it must be defined as usual.

Eg,

int (*funcPointer) (int, char, int);

funcPointer is a pointer to a function.

- The extra parentheses around (*funcPointer) is needed because there are precedence relationships in declaration just as there are in expressions

Assign an address to a Function Pointer

```
 //assign an address to the function pointer
int (*funcPointer) (int, char, int);

int firstExample ( int a, char b, int c){
 printf(" Welcome to the first example");
 return a+b+c;
}
funcPointer= firstExample; //assignment
funcPointer=&firstExample; //alternative using address
  operator
```

Assign an address to a Function Pointer

- It is optional to use the address operator ‘&’ infront of the function’s name
- When you mention the name of a function but are not calling it, there’s nothing else you could possibly be trying to do except for generating a pointer to it
- Similar to the fact that a pointer to the first element of an array is generated automatically when an array appears in an expression

Comparing Function Pointers

- Can use the (==) operator

```
//comparing function pointers
If (funcPointer == &firstExample)
   printf ("pointer points to firstExample");
```

Calling a function using a Function Pointer

- There are two alternatives

1) Use the name of the function pointer
2) Can explicitly dereference it

```
int (*funcPointer) (int, char, int);
// calling a function using function pointer
 int answer= funcPointer (7, 'A' , 2 );
 int answer=(* funcPointer) (7, 'A' , 2 );
```

Arrays of Function Pointers

- C treats pointers to functions just like pointers to data therefore we can have arrays of pointers to functions
- This offers the possibility to select a function using an index

Eg.

suppose that we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array:

```
void (*file_cmd[]) (void) =
{   new_cmd,
    open_cmd,
    close_cmd,
    save_cmd ,
    save_as_cmd,
    print_cmd,
    exit_cmd
};
```

If the user selects a command between 0 and 6, then we can subscript the file_cmd array to find out which function to call

```
file_cmd[n]();
```

6. Structures and pointers

Pointer to Structure

```
void main()
{
   struct book
   {
        char name[25];
        char author[25];
        int callno;
   };
   struct book b1={"Let us c","Yaswant Kanetkar",101};
   struct book *ptr;
   ptr=&b1;
   printf("%s %s %d",b1.name,b1.author,b1.callno);
   printf("\n %s %s %d",ptr->name,ptr->author,ptr->callno);
   }
```

Passing address of a structure variable

```
struct book
{
        char name[25];
        char author[25];
        int callno;
};
void display(struct book*);
void main()
{
        struct book b1={"Let us c","Yaswant Kanetkar",101};
        display(&b1);
}
```

Continue…

```
void display(struct book *b)
  {
      printf("\n %s %s %d",b->name,b-
      >author,b->callno);
  }
```

Comparison of Pointer and Nonpointer Variables

- The actual data value of a pointer variable is accessed indirectly.
- The actual data value of a nonpointer variable can be accessed directly.

Reference	Explanation	Value
num	Direct value of num	3
nump	Direct value of nump	Pointer to location containing 3
*nump	Indirect value of nump	3

Pointer Review

- A call to a function with pointer parameters may need to use the & operator.
  - e.g., if we have an int variable `value1` and `f1(int *value)`, `f1(&value1)` is a legal call.
- A pointer can be used to represent an array.
  - e.g., char n[] is equal to char *n.
- A pointer can also represent a structure.
  - e.g., File * is a pointer to a File structure.

7. Dynamic memory allocation

Memory Allocation (1/3)

- C provides a memory allocation function called `malloc`, which resides in the `stdlib` library.
  - This function requires *an argument* which indicates the amount of memory space needed.
  - The returned data type is `(void *)` and should be always cast to the specific type.
- E.g.,
  Declaration:

```
int *nump; char *letp; planet_t
*planetp;
```

Memory Allocation (2/3)

- Allocation:

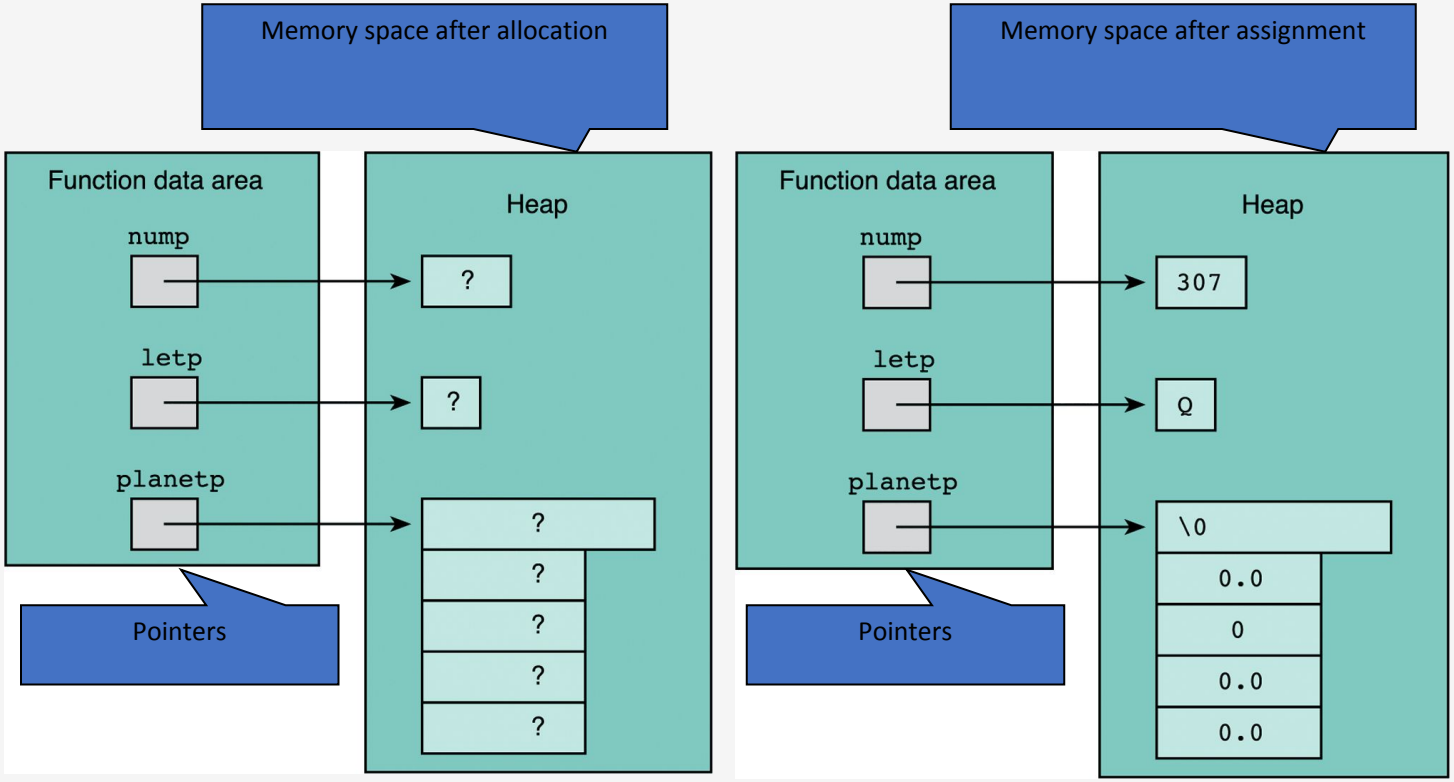
```
nump = (int *) malloc (sizeof (int));
letp = (char *) malloc (sizeof (char));

planetp = (planet_t *) malloc (sizeof(planet_t));
```

- Assignment:

```
*nump = 307;
*letp = 'Q';
*planetp = blank_planet;
```

Memory Allocation (3/3)



Dynamic Array Allocation

- C provides a function `calloc` which creates an array of elements of any type and *initializes the array elements to zero.*
  - Function `calloc` takes two arguments: the *number of array elements and the size of one element*.
- E.g.,

```
int *array_of_nums;
array_of_nums =(int *)calloc(10, sizeof(int));
```

Free Memory

- The allocated memory space can be released by the function `free`.
  - E.g., `free(letp)` returns the allocated memory space for the pointer variable `letp`.
- Once the memory space is released, we can not access the space again. Otherwise, it is considered as an illegal memory access.

We have covered

- Introduction to Pointers, pointer arithmetic, Pointer to pointer
- Dynamic memory allocation
- Pointer to single and multidimensional arrays, String manipulation using pointers
- Array of pointers
- Structure manipulation using pointers
- Call by reference
- Pointer to functions
- Pointer to file structure and basic operations on file
- File handling in C.

Wrap up and related outcomes

After learning this subunit, you must be able to

1. Perform dynamic memory allocation
2. Perform pointer arithmetic
3. Use pointers with arrays and structures
4. Use pointers with functions.

Instructions/Guidelines/References

Practice the problem statements given in the exercises of

1. Pointers in C, Yashwant Kanitkar
2. Let us C, Yashwant Kanitkar

To be discussed next time

Part –I

- Introduction to Pointers
- Dynamic memory allocation
- Pointer to pointer

Part-II

- Pointer to single and multidimensional arrays
- Array of pointers, String and structure manipulation using pointers
- Pointer to functions

Part-III

- Pointer to file structure
- basic operations on file
- File handling in C

Thank You