2.2 Pointers

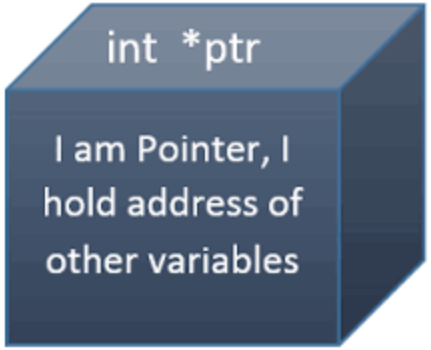
Objectives

- To understand the need and application of pointers
- To learn how to declare a pointer and how it is represented in memory
- To learn the relation between arrays and pointers
- To study the need for call-by-reference
- To distinguish between some special types of pointers

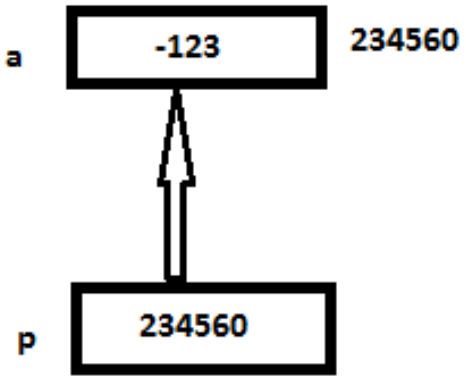
Agenda

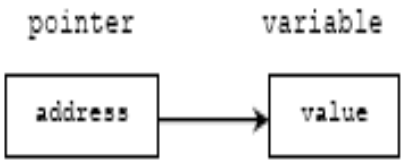
- Basics of Pointers
- Declaration and Memory Representation
- Operators associated with pointers
  - **address of** operator
  - **dereferencing** operator
- Arrays and Pointers.
- Compatibility of pointers
- Functions and Pointers
- Special types of pointers
  - void pointer
  - null pointer
  - constant pointers
  - dangling pointers
  - pointer to pointer

Introduction



- A **pointer** is defined as a variable whose value is the address of another variable.
- It is mandatory to declare a pointer before using it to store any variable address.





a is a variable with value -123 and address 234560

p is apointer to a;i.e.; it stores &a as its value

Pointer Declaration

- General form of a pointer variable declaration:-

  datatype *ptrname;

- **Eg:-**
  - **int *p;** (p is a pointer that can point only integer variables)
  - **float *fp;** (fp can point only floating-point variables)

- Actual data type of the value of all pointers is a long hexadecimal number that represents a memory address

Initialization of Pointer Variable

- Uninitialized pointers will have some unknown memory address in them.
- Initialize/ Assign a valid memory address to the pointer.

Initialization	Assignment
int a; int *p = &a;	int a; int *p; p = &a;

- The variable should be defined before pointer.
- Initializing pointer to NULL

  int *p = NULL;

Why Pointers?

- Manages memory more efficiently.
- Leads to more compact and efficient code than that can be obtained in other ways
- One way to have a function modify the actual value of a variable passed to it.
- Helps to dynamically allocate memory when the exact amount of memory required is not known in the beginning.

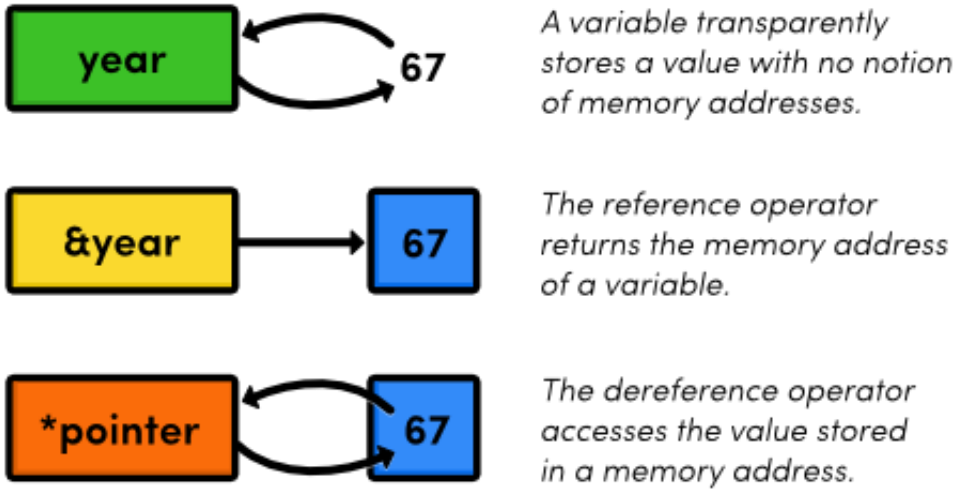
Referencing/ "Address of" operator

- To make a pointer point to another variable, it is necessary to obtain the memory address of that variable.
- To get the memory address of a variable (its location in memory), put the **&** sign in front of the variable name.
- **&** is called the **address-of** operator, because it returns the memory address. It's a unary operator.
- It is also known as Referencing operator as it refers/points to another variable of same data type.

Dereferencing/Indirection Operator

- It's a unary operator - *
- '*' is followed by the pointer name, say p ; i.e.; *p.
- It looks at the address stored in p, and goes to that address and returns the value.
- This is akin to looking inside a safety deposit box only to find the number of (and, presumably, the key to) another box, which you then open.

Referencing & Dereferencing Operators



Sample Code -1 : Simple Pointer

```
#include<stdio.h>
int main()
{
int x=10;
int *ip;
ip=&x;
printf("Number : %d\n",x);
printf("Address: %p\n",(&x));
printf("Number using pointer :  %d\n", *ip);
printf("Address using Pointer: %p\n",ip);
return 0;
}
```

Output:-

Number : 10

Address: 0x7fff4fab3044

Number using pointer : 10

Address using Pointer: 0x7fff4fab3044

Sample Code -2 : Pointers to different types

```
#include<stdio.h>
int main()
{
int x=10;
int *ip;
float y=2.5, *fp;
fp = &y;
ip=&x;
printf("Number using pointer :  %d\n", *ip);
printf("Address using Pointer: %p\n",ip);
printf("Decimal value : %f\n",*fp);
printf("Address of y : %p\n",fp);
return 0;
}
```

Output:-

Number using pointer : 10

Address using Pointer: 0x7fff4f5c31bc

Decimal value : 2.500000

Address of y : 0x7fff4f5c31b8

Sample Code -3 : Same pointer to multiple variables

```
#include<stdio.h>
int main()
{
 int a,b,c,*p; //a,b and c are variables and p is a
pointer
 printf("Enter three integers : ");
 scanf("%d %d %d",&a,&b,&c);
 p = &a;
 printf("pointer points to a. Value is %d\n",*p);
 p = &b;
 printf("pointer points to b. Value is %d\n",*p);
 p = &c;
 printf("pointer points to c. Value is %d\n",*p);
 return 0;
}
```

Output:-

Enter three integers : 10 20 30

pointer points to a. Value is 10

pointer points to b. Value is 20

pointer points to c. Value is 30

Sample Code -4 : Multiple Pointers to same variable

```
#include<stdio.h>
int main()
{
int a;
int *p = &a;
int *q = &a;
int *r = &a;

printf("Enter an integer : ");
scanf("%d",&a);
printf("%d\n",*p);
printf("%d\n",*q);
printf("%d\n",*r);

return 0;
}
```

Output:-

```
Enter an integer : 15
15
15
15
```

Pointer and 1D Array

Relationship between array and pointer

- The name of array is a pointer to the first element
- Address of first element and name of array represent the same memory address.
- Array name can be used as a pointer.
- When a separate pointer is used to point to an array, it is initialized using the following syntax:-

datatype *ptrname = array_name;

Eg:- int a[5] = {1,2,3,4,5};

int *ptr = a; //Equivalent to writing int *ptr = &a[0];

Sample Program 5 : Pointer and Array I

```
#include<stdio.h>
int main()
{
  int a[5] = {1,2,3,4,5};
  int *p = a;

  printf("%p %p\n",&a[0],a);
  printf("%d %d\n",*a,*p);

  return 0;
}
```

Output:-

0x7fff03b2d380 0x7fff03b2d380

1 1

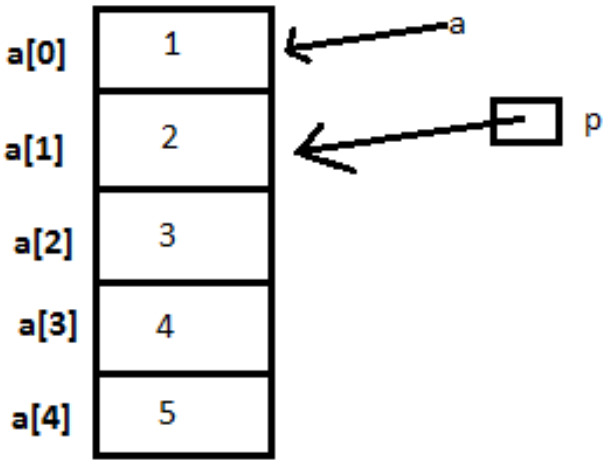
Sample Program 6 : Pointer and Array II

```
#include<stdio.h>
int main()
{
 int a[5] = {1,2,3,4,5};
 int *p = &a[1];

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

 return 0;
}
```

Output:-

First element : 1 1

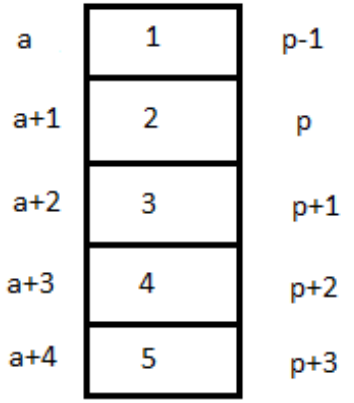
Second element: 2 2



Note:-When a pointer to an array is not pointing to the first element, index can be negative.

Pointer Arithmetic and 1D Arrays

- If 'a' is an array name, then 'a' points to first element
- a+1 points to the second element, a+2 points to third element and so on.
- Generally *(a+n)* points to *(n+1)*th element.

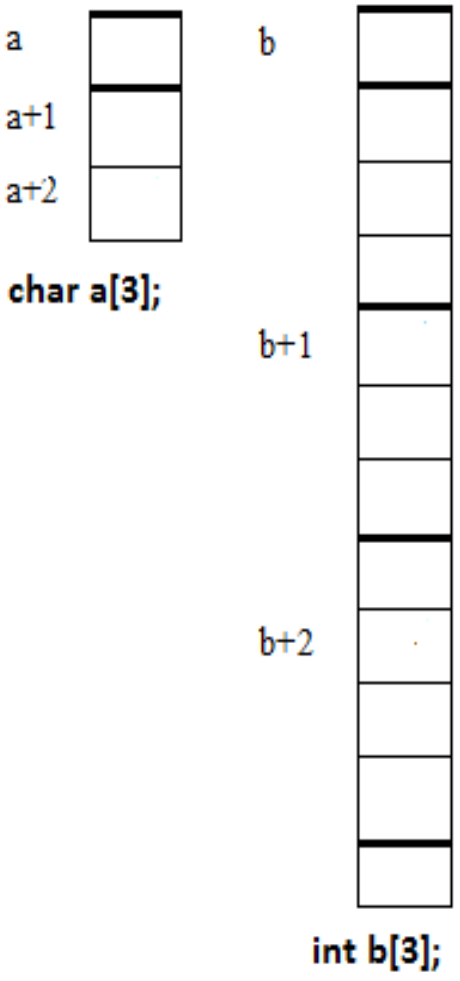


Assumption : p points to second element of the array

- Similarly, for a pointer p, ***p±n*** points to a location which is *n elements away* from current location.
- Actual address will be **p+n*(size of one element).**

Pointer arithmetic on different data types

- Size of single element varies with respect to data type of array.
- 'char' takes one byte per character stored, whereas 'int' and 'float' takes 4 bytes per value stored.
- Hence adding 1 to array name points to different addresses for different data types



The image part with relationship ID rId3 was not found in the file.

Modifying values using pointers

Modifying value using pointer

- If ip points to an integer x, ***ip** can be used in places where x could have been used.
  - ***ip = *ip + 10;** will modify the value of x by 10
  - **y = *ip + 1;** is equivalent to **y = x+1;**
  - ***ip += 1** can be written as ++(*ip) or (*ip)++

Sample Code –7: Updating value using Pointer

```
#include<stdio.h>
int main()
{
int x=10;
int *ip;
float y=2.5, *fp;
fp = &y;
ip=&x;
printf("Number using pointer :  %d\n", *ip);
printf("Address using Pointer: %p\n",ip);
printf("Decimal value : %f\n",*fp);
printf("Address of y : %p\n",fp);
x+=1;
printf("Updated Number :  %d\n", *ip);
*ip *=5;
printf("Updated Number :  %d\n", *ip);
++*ip;
printf("Updated Number :  %d\n", *ip);
return 0;
}
```

Output:-

```
Number using pointer :  10
Address using Pointer: 0x7fff76d4f8ec
Decimal value : 2.500000
Address of y : 0x7fff76d4f8e8
Updated Number :  11
Updated Number :  55
Updated Number :  56
```

Sample Code -8 : Adding two numbers using Pointers

```
#include<stdio.h>
int main()
{
 int a=10,b=5,c;
 int *p1 = &a;
 int *p2 = &b;
 int *res = &c;

 *res = *p1 + *p2;

 printf("%d + %d = %d\n",*p1,*p2,c);

 return 0;
}
```

Output:-

10 + 5 = 15

Pointer to array : Order of placing '*' and '++'

- Assume that z is an inger array with two values 1 and 2 (int z[2]={1,2};)
- Let ip be a pointer to z; (int *ip = z;)

- **printf("%d\n", ++*ip);**
  - increments content in the address pointed by ip.
  - z[0]=1 was taken and incremented by 1.
  - In output the value is 2

- **printf("%d\n", *++ip);**
  - increments the address pointed by ip.
  - ip currently points to z[1]=3
  - In output the value is 3

- **Order of placing '++' and '*' is crucial**
  - ++ followed by * → Value is incremented.
  - * followed by ++ → Address is incremented.

Sample Code Snippet - 9a – ++ before *

```
#include<stdio.h>
main()
{
 int z[2]={1,3};
 int * ip = z;
 printf("%p\n",ip);
 printf("%d\n", *ip);
 printf("%d\n", ++*ip);
 printf("%p",ip);
 return 0;
}
```

- Sample Output

0x7fff0fc66d30

1

2

0x7fff0fc66d30

Sample Code Snippet -9b : * before ++

```
#include<stdio.h>
main()
{
 int z[2]={1,3};
 int * ip = z;
 printf("%p\n",ip);
 printf("%d\n", *ip);
 printf("%d\n",*++ip);
 printf("%p",ip);
 return 0;
}
```

- Sample Output

0x7ffffc801740

1

3

0x7ffffc801744

Pointer Compatibility

Pointer Compatibility

- Pointers have a type associated with them → They can **point only to specific type.**

- **Two types:-**
  - Pointer Size Compatibility
  - Pointer Dereferencing compatibility

Pointer Size Compatibility

- Size of all pointers is the same; i.e.; every pointer variable holds the address of one memory location. But the size of variable that the pointer points to can be different.

- Size of the type that a pointer points to is same as its data size.

- Size is dependent on type; not on the value.

Sample Code -10 : Pointer Size Compatibility

```
#include<stdio.h>
int main()
{
 char c;
 char* pc;
 int sizeofc = sizeof(c);
 int sizeofpc = sizeof(pc);
 int sizeofstarpc = sizeof(*pc);

 int a;
 int* pa;
 int sizeofa = sizeof(a);
 int sizeofpa = sizeof(pa);
 int sizeofstarpa = sizeof(*pa);

 double d;
 double* pd;
 int sizeofd = sizeof(d);
 int sizeofpd = sizeof(pd);
 int sizeofstarpd = sizeof(*pd);
```

```
printf("Size of c : %3d | ",sizeofc);
 printf("Size of pc : %3d | ",sizeofpc);
 printf("size of *pc : %3d\n",sizeofstarpc);
 printf("Size of a : %3d | ",sizeofa);
 printf("Size of pa : %3d | ",sizeofpa);
 printf("size of *pa : %3d\n",sizeofstarpa);
 printf("Size of d : %3d | ",sizeofd);
 printf("Size of pd : %3d | ",sizeofpd);
 printf("size of *pd : %3d\n",sizeofstarpd);
 return 0;
}
```

Sample Output

Size of c : 1 | Size of pc : 8 | size of *pc : 1

Size of a : 4 | Size of pa : 8 | size of *pa : 4

Size of d : 8 | Size of pd : 8 | size of *pd : 8

Dereferencing Compatibility

- Dereference type is the type of variable that the pointer is referencing.
- It is usually invalid to assign a pointer of one type to address of a variable of another type.
- It is also invalid to assign a pointer of one type to pointer of another type.
- Exception : pointer to void (Will be discussed later.)

Sample Code 11: Pointer Dereferencing Incompatibility

```
#include<stdio.h>
int main()
{
int x=10;
int *ip;
float y=2.5, *fp;
fp = &y;
ip=&x;
printf("Number using pointer :  %d\n", *ip);
printf("Address using Pointer: %p\n",ip);
printf("Decimal value : %f\n",*fp);
printf("Address of y : %p\n",fp);
fp = &x;
printf("New value pointed by fp = %f\n",*fp);
return 0;
}
```

ptrcompat.c: In function ‘main’:

ptrcompat.c:13: warning: assignment from incompatible pointer type

Output:-

Number using pointer : 10

Address using Pointer: 0x7fffda19b6ec

Decimal value : 2.500000

Address of y : 0x7fffda19b6e8

New value pointed by fp = 0.000000

Pointers and Functions

How to swap two numbers using function?

```
#include<stdio.h>
int main()
{
 int a,b;
 void swap(int ,int );
 printf("Enter first number : " );
 scanf("%d",&a);
 printf("Enter second number: ");
 scanf("%d",&b);
 printf("Numbers before function call: %d\t%d\n",a,b);
 swap(a,b);
 printf("Numbers after function call : %d\t%d\n",a,b);
 return 0;
}
```

Output:-

```
Enter first number : 5
Enter second number: 10
Numbers before function call: 5 10
Numbers before swapping : 5     10
Numbers after swapping  : 10    5
Numbers after function call : 5 10
```

```
void swap(int a, int b)
{
 int t;
 printf("Numbers before swapping : %d\t%d\n",a,b);
 t = a;
 a = b;
 b = t;
 printf("Numbers after swapping  : %d\t%d\n",a,b);
}
```

How to swap two numbers using function?

- Values are getting interchanged inside the function. But that is not getting reflected in main.
- Call-by-value will not interchange numbers.
- If you want to modify the actual parameters, you require 'Call-by-Reference'.
- This type of function requires pointers.

How to swap two numbers using function?

```
#include<stdio.h>
int main()
{
 int a,b;
 void swap(int *,int *);
 printf("Enter first number : " );
 scanf("%d",&a);
 printf("Enter second number: ");
 scanf("%d",&b);
 printf("Numbers before function call: %d\t%d\n",a,b);
 swap(&a,&b);
 printf("Numbers after function call : %d\t%d\n",a,b);
 return 0;
}
```

Output:-

```
Enter first number : 5
Enter second number: 10
Numbers before function call: 5 10
Numbers before swapping : 5     10
Numbers after swapping  : 10    5
Numbers after function call : 10   5
```

```
void swap(int *a, int *b)
{
 int t;
 printf("Numbers before swapping : %d\t%d\n",*a,*b);
 t = *a;
 *a = *b;
 *b = t;
 printf("Numbers after swapping  : %d\t%d\n",*a,*b);
}
```

Points to be noted while using Call-by-Reference

	Call-by-Value	Call-by-Reference
Function Declaration	void swap(int ,int);	void swap(int *,int *);
Function Header	void swap(int a, int b)	void swap(int *a, int *b)
Function Call	swap(a,b);	swap(&a,&b);

- Requires '*' operator along with data type of arguments – in declaration as well as Function header.
- Requires '&' along with actual arguments in Function call.
- Requires '*' operator inside function body.

When do you need pointers in functions?

- First scenario – In Call-by-Reference.
  - There is a requirement to modify the values of actual arguments.

- Second scenario – While passing array as an argument to a function.

- Third Scenario - If you need to return multiple values from a function.

Passing array as an argument to a function

- When an array is passed as an argument to a function, it is actually passed as reference.

- If any modification of array elements is done inside the function, it actually changes the original value stored in the array.

- Since modifications affect actual values, array need not be returned from the function.

Sample Code 12 : Passing an array to a function

```
#include<stdio.h>
int main()
{
 int a[5]={1,2,3,4,5};
 int i;
//First argumnt is an array and second argument is its size
 void square(int *,int);
 printf("Array before modification:-\n");
 for(i=0;i<5;i++)
   printf("%d\t",a[i]);
 square(a,5);
 printf("\nArray after  modification:-\n");
 for(i=0;i<5;i++)
   printf("%d\t",a[i]);
 printf("\n");
 return 0;
}
```

```
void square(int *a,int n)
{
 int i;
 for(i=0;i<n;i++)
    a[i] *= a[i];
}
```

Output:-

Array before modification:-

1 2 3 4 5

Array after modification:-

1 4 9 16 25

Returning Multiple values from a function

- Normally, a function can return only a single value from it, using 'return'.

- What if, you have to return two or more values from a function?

- You can make use of pointers to return multiple values.
  - Use one or more additional pointer variables as arguments

Sample Code 13 : Returning Multiple Values

```
#include<stdio.h>
int main()
{
  int a[5];
  void MinMax(int *,int,int*,int*);
  int i,min,max;
  for(i=0;i<5;i++)
  {
     printf("Number %d : ",(i+1));
     scanf("%d",&a[i]);
  }
  MinMax(a,5,&min,&max);
  printf("Minimum element entered : %d\n",min);
  printf("Maximum element entered : %d\n",max);
  return 0;
}
```

```
void MinMax(int *a, int n, int *min,int *max)
{
  int i;
  *min = *max = a[0];
  for(i=1;i<n;i++)
  {
    if(a[i] < (*min))
      *min = a[i];
  }
  for(i=1;i<n;i++)
  {
    if(a[i] > (*max))
      *max = a[i];
  }
}
```

Sample Code 13 : Returning Multiple Values - Output

Output:-

Number 1 : 2

Number 2 : 1

Number 3 : 3

Number 4 : 5

Number 5 : 4

Minimum element entered : 1

Maximum element entered : 5

The image part with relationship ID rId3 was not found in the file.

Special Pointers

Void Pointer

- A generic type that is not associated with a reference type.
- It is not the address of a particular data type.
- A pointer with no reference type that can store only address of any variable.
- Compatible for assignment purposes only with all other types of pointers.
- A pointer of any reference can be assigned to void type and vice verse.
- Restriction : It can not be dereferenced unless it is cast.
- Declaration:- void* pvoid;
- General Casting:- dest_ptr = (dest_ptr_type *) source_ptr_name;

NULL Pointer

- A pointer of any type that is assigned the constant NULL
- The reference type of pointer will not change by the assignment of NULL
- Eg:-

  int* iptr = NULL; //NULL pointer of type 'int'

  Char* cptr = NULL; //NULL pointer of type 'char'

Dangling Pointer

- Arises during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.
- Example of creating Dangling pointer

```
int main()
 {
   char *ptr=NULL;
   {
        char c;
        ptr = &c;
   }
 }
```

- c falls out of scope after the inner block making ptr a dangling pointer

Constant pointer

- A **pointer** that cannot change the address its holding.
- Once a constant pointer points to a variable then it cannot point to any other variable.
- Declaration:- <type of pointer> * const <name of pointer>
- Example:- int* const ptr;
- Sample Code : Will give the error 7: error: assignment of read-only variable ‘ptr’

```
#include<stdio.h>
int main(void)
{
   int var1 = 0, var2 = 0;
   int *const ptr = &var1;
   ptr = &var2;
   printf("%d\n", *ptr);
   return 0;
}
```

Pointer to a Pointer

- A **pointer** points to an address of another **pointer.**
- Also called Double Pointer.
- Declaration:- type **ptr_name;
- Example:- int **p;
- Sample:-

```
int main()
{
   int p=5;
   int *p1 = &p;
   int **pp;
   pp = &p1;
   printf("Value of P : %d\n",**pp);
   return 0;
}
```

Output : Value of P : 5

Summary

- Discussed about Pointer and its importance.
- Discussed relationship between array and pointer
- Discussed about Call-by-Reference and other places where pointers are needed for functions.
- Discussed special pointers.

References

- Books/Materials

1. C Programming Course – Compiled HTML Help File
2. Brian W Kernighan, Dennis M Ritchie, "The C Programming Language", 2nd Edition, PHI

- Web

1. http://www.tutorialspoint.com/cprogramming/c_pointers.htm
2. http://www.cprogramming.com/tutorial/c/lesson6.html