

Pointers

Part 1

Introduction

- A pointer is a variable that represents the location (rather than the value) of a data item.
- They have a number of useful applications.
 - Enables us to access a variable that is defined outside the function.
 - Can be used to pass information back and forth between a function and its reference point.
 - More efficient in handling data tables.
 - Reduces the length and complexity of a program.
 - Sometimes also increases the execution speed.

Basic Concept

- Within the computer memory, every stored data item occupies one or more contiguous memory cells.
 - The number of memory cells required to store a data item depends on its type (char, int, double, etc.).
- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
 - Since every byte in memory has a unique address, this location will also have its own (unique) address.

Contd.

- Consider the statement

```
int xyz = 50;
```

- This statement instructs the compiler to allocate a location for the integer variable `xyz`, and put the value `50` in that location.
- Suppose that the address location chosen is `1380`.

xyz	→	variable
50	→	value
1380	→	address

Contd.

- During execution of the program, the system always associates the name **xyz** with the address **1380**.
 - The value **50** can be accessed by using either the name **xyz** or the address **1380**.
- Since memory **addresses** are simply numbers, they can be **assigned to some variables** which can be stored in memory.
 - Such variables that hold memory addresses are called **pointers**.
 - Since a pointer is a variable, its value is also stored in some memory location.

Contd.

- Suppose we assign the **address of xyz** to a variable **p**.
 - **p** is said to point to the variable **xyz**.

<u>Variable</u>	<u>Value</u>	<u>Address</u>
xyz	50	1380
p	1380	2545

p = &xyz;



Accessing the Address of a Variable

- The address of a variable can be determined using the **'&'** operator.
 - The operator **'&'** immediately preceding a variable returns the **address** of the variable.
- Example:
 - p = &xyz;**
 - The **address** of xyz (1380) is assigned to p.
- The **'&'** operator can be used only with a **simple variable** or an **array element**.

&distance

&x[0]

&x[i-2]

Contd.

- Following usages are illegal:

&235

- Pointing at constant.

```
int arr[20];
```

:

```
&arr;
```

- Pointing at array name.

```
&(a+b)
```

- Pointing at expression.

Example

```
#include <stdio.h>
int main( )
{
    int a;
    float b, c;
    double d;
    char ch;

    a = 10; b = 2.5; c = 12.36; d = 12345.66; ch = 'A';
    printf ("%d is stored in location %u \n", a, &a);
    printf ("%f is stored in location %u \n", b, &b);
    printf ("%f is stored in location %u \n", c, &c);
    printf ("%ld is stored in location %u \n", d, &d);
    printf ("%c is stored in location %u \n", ch, &ch);
}
```

Output:

10 is stored in location 3221224908

a

2.500000 is stored in location 3221224904

b

12.360000 is stored in location 3221224900

c

12345.660000 is stored in location 3221224892

d

A is stored in location 3221224891

ch

Incidentally variables a,b,c,d and ch are allocated to contiguous memory locations.

Pointer Declarations

- Pointer variables must be declared before we use them.
- General form:

```
data_type *pointer_name;
```

Three things are specified in the above declaration:

1. The asterisk (*) tells that the variable `pointer_name` is a pointer variable.
2. `pointer_name` needs a memory location.
3. `pointer_name` points to a variable of type `data_type`.

Contd.

- Example:

```
int *count;
```

```
float *speed;
```

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```
int *p, xyz;
```

```
:
```

```
p = &xyz;
```

- This is called **pointer initialization**.

Things to Remember

- Pointer variables must always point to a data item of the *same type*.

```
float x;
```

```
int *p;
```

```
:
```

→ will result in erroneous output

```
p = &x;
```

- Assigning an absolute address to a pointer variable is prohibited.

```
int *count;
```

```
:
```

```
count = 1268;
```

Accessing a Variable Through its Pointer

- Once a pointer has been assigned the **address** of a variable, the **value** of the variable can be accessed using the **indirection operator** (*).

```
int a, b;
```

```
int *p;
```

```
:
```

```
p = &a;
```

```
b = *p;
```



```
b = a
```

Example 1

```
#include <stdio.h>
main()
{
    int  a, b;
    int  c = 5;
    int  *p;

    a = 4 * (c + 5);

    p = &c;
    b = 4 * (*p + 5);
    printf ("a=%d b=%d \n", a, b);
}
```

Equivalent



Example 2

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    int x, y;
```

```
    int *ptr;
```

```
    x = 10 ;
```

```
    ptr = &x ;
```

```
    y = *ptr ;
```

```
    printf ("%d is stored in location %u \n", x, &x) ;
```

```
    printf ("%d is stored in location %u \n", *&x, &x) ;
```

```
    printf ("%d is stored in location %u \n", *ptr, ptr) ;
```

```
    printf ("%d is stored in location %u \n", y, &*ptr) ;
```

```
    printf ("%u is stored in location %u \n", ptr, &ptr) ;
```

```
    printf ("%d is stored in location %u \n", y, &y) ;
```

```
    *ptr = 25;
```

```
    printf ("\nNow x = %d \n", x);
```

```
}
```

***&x ⇔ x**

**ptr=&x;
&x ⇔ &*ptr**

Output:

10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
10 is stored in location 3221224908
3221224908 is stored in location 3221224900
10 is stored in location 3221224904

Now x = 25

Address of x: 3221224908

Address of y: 3221224904

Address of ptr: 3221224900

Pointer Expressions

- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are two pointers, the following statements are valid:

```
sum = *p1 + *p2 ;
```

```
prod = *p1 * *p2 ;
```

```
prod = (*p1) * (*p2) ;
```

```
*p1 = *p1 + 2;
```

```
x = *p1 / *p2 + 5 ;
```

Contd.

- What are allowed in C?
 - Add an integer to a pointer.
 - Subtract an integer from a pointer.
 - Subtract one pointer from another (related).
 - If `p1` and `p2` are both pointers to the same array, then `p2-p1` gives the number of elements between `p1` and `p2`.
- What are not allowed?
 - Add two pointers.
`p1 = p1 + p2 ;`
 - Multiply / divide a pointer in an expression.
`p1 = p2 / 5 ;`
`p1 = p1 - p2 * 10 ;`

Scale Factor

- We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int *p1, *p2 ;  
int i, j;  
:  
p1 = p1 + 1 ;  
p2 = p1 + j ;  
p2++ ;  
p2 = p2 - (i + j) ;
```

- In reality, it is not the integer value which is added/subtracted, but rather the **scale factor times the value**.

Contd.

<u>Data Type</u>	<u>Scale Factor</u>
char	1
int	4
float	4
double	8

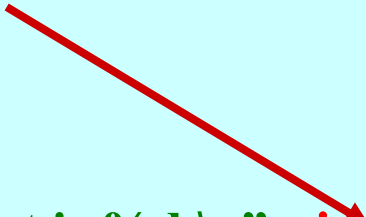
– If p1 is an integer pointer, then

`p1++`

will increment the value of `p1` by 4.

Returns no. of bytes required for data type representation

```
#include <stdio.h>
main()
{
    printf ("Number of bytes occupied by int is %d \n", sizeof(int));
    printf ("Number of bytes occupied by float is %d \n", sizeof(float));
    printf ("Number of bytes occupied by double is %d \n", sizeof(double));
    printf ("Number of bytes occupied by char is %d \n", sizeof(char));
}
```



Output:

Number of bytes occupied by int is 4
Number of bytes occupied by float is 4
Number of bytes occupied by double is 8
Number of bytes occupied by char is 1

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
 - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
 - Called **call-by-reference** (or by **address** or by **location**).
- Normally, arguments are passed to a function **by value**.
 - The data items are copied to the function.
 - Changes are not reflected in the calling program.

Example: passing arguments by value

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ; b = 20 ;
    swap (a, b) ;
    printf (“\n a = %d, b = %d”, a, b);
}

void swap (int x, int y)
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
}
```

**a and b
do not
swap**

x and y swap

Output

a = 5, b = 20

Example: passing arguments by reference

```
#include <stdio.h>
main()
{
    int a, b;
    a = 5 ; b = 20 ;
    swap (&a, &b) ;
    printf (“\n a = %d, b = %d”, a, b);
}

void swap (int *x, int *y)
{
    int t ;
    t = *x ;
    *x = *y ;
    *y = t ;
}
```

***(&a) and *(&b)
swap**

***x and *y
swap**

Output

a = 20, b = 5

scanf Revisited

```
int x, y ;  
printf ("%d %d %d", x, y, x+y) ;
```

- What about scanf ?

NO

```
scanf ("%d %d %d", x, y, x+y) ;
```

YES

```
scanf ("%d %d", &x, &y) ;
```

Example: Sort 3 integers

- Three-step algorithm:
 1. Read in three integers x , y and z
 2. Put smallest in x
 - Swap x , y if necessary; then swap x , z if necessary.
 3. Put second smallest in y
 - Swap y , z if necessary.

Contd.

```
#include <stdio.h>
main()
{
    int x, y, z ;
    .....
    scanf ("%d %d %d", &x, &y, &z) ;
    if (x > y) swap (&x, &y);
    if (x > z) swap (&x, &z);
    if (y > z) swap (&y, &z) ;
    .....
}
```

sort3 as a function

```
#include <stdio.h>
main()
{
    int x, y, z ;
    .....
    scanf ("%d %d %d", &x, &y, &z) ;
    sort3 (&x, &y, &z) ;
    .....
}

void sort3 (int *xp, int *yp, int *zp)
{
    if (*xp > *yp) swap (xp, yp);
    if (*xp > *zp) swap (xp, zp);
    if (*yp > *zp) swap (yp, zp);
}
```

**xp/yp/zp
are
pointers**

Contd.

- Why no '&' in swap call?
 - Because xp, yp and zp are already pointers that point to the variables that we want to swap.

Pointers and Arrays

- When an array is declared,
 - The compiler allocates a **base address** and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
 - The **base address** is the location of the first element (index 0) of the array.
 - The compiler also defines the array name as a **constant pointer** to the first element.

Example

- Consider the declaration:

```
int x[5] = {1, 2, 3, 4, 5};
```

- Suppose that the base address of x is 2500, and each integer requires 4 bytes.

<u>Element</u>	<u>Value</u>	<u>Address</u>
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Contd.

$x \Leftrightarrow \&x[0] \Leftrightarrow 2500 ;$

- $p = x;$ and $p = \&x[0];$ are equivalent.
- We can access successive values of x by using $p++$ or $p--$ to move from one element to another.
- Relationship between p and x :

$p = \&x[0] = 2500$

$p+1 = \&x[1] = 2504$

$p+2 = \&x[2] = 2508$

$p+3 = \&x[3] = 2512$

$p+4 = \&x[4] = 2516$

***(p+i) gives the
value of x[i]**

Example: function to find average

```
#include <stdio.h>
main()
{
    int x[100], k, n ;

    scanf ("%d", &n) ;

    for (k=0; k<n; k++)
        scanf ("%d", &x[k]) ;

    printf ("\nAverage is %f",
            avg (x, n));
}
```

int *array

```
float avg (int array[ ],int size)
{
    int *p, i , sum = 0;

    p = array ;

    for (i=0; i<size; i++)
        sum = sum + *(p+i);

    return ((float) sum / size);
}
```

p[i]

