Pointers

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**7**	**Week No:**	**5 (1X1.5 hrs)**	**Semester:**	
Lecturer:	*Name & email*				

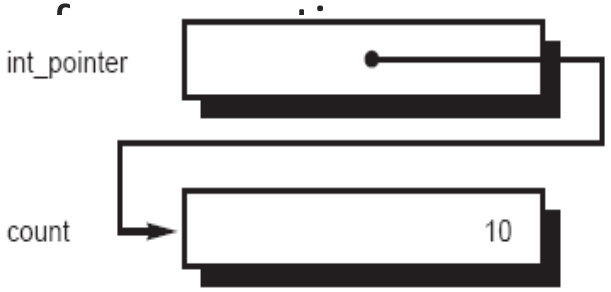
Outline

- Pointers
  - Pointers and Addresses
  - Pointers and Function Arguments
  - Pointers and Arrays
  - Pointer Arithmetic
  - Pointers and strings
  - Dynamic memory allocation
  - Pointer arrays. Pointers to pointers
  - Multidimensional arrays and pointers
  - Structures and pointers

Pointers and addresses

a *pointer* is a *variable* whose *value* is a *memory address*

- `int count = 10;`
- `int *int_pointer;`
- `int_pointer = &count;`
- The **address operator** has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a *pointer* to the variable `count`.
- We say that `int_ptr` "points to" `count`
- The values and the format of the numbers representing memory addresses depend on the computer architecture and operating system. In order to have a portable wa[illegible] addresses, we need a different type th[illegible]
- To print addresses: `%p`



Lvalues and Rvalues

- There are two “values” associated with any variable:
  - An "lvalue" (left value) of a variable is the value of its address, where it is stored in memory.
  - The "rvalue" (right value) of a variable is the value stored in that variable (at that address).
- The lvalue is the value permitted on the left side of the assignment operator '=' (the address where the result of evaluation of the right side will be stored).
- The rvalue is that which is on the right side of the assignment statement

a = a+1

Declaring pointer variables

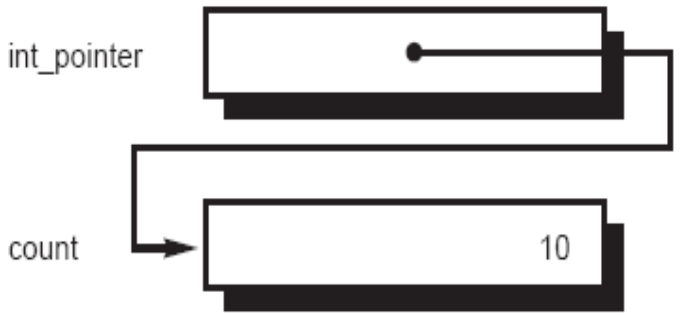
```
type *  variable_name;
```

- it is not enough to say that a variable is a pointer. You also have to specify the *type of variable to which the pointer points !*
  - `int * p1; // p1 points to an integer`
  - `float *  p2; // p2 points to a float`
- Exception: generic pointers `(void *)` indicate that the pointed data type is unknown
  - may be used with explicit type cast to any type (type *)
  - `void * p;`

Indirection (dereferencing) operator *

- To reference the contents of count through the pointer variable int_pointer, you use the ***indirection* operator**, which is the asterisk * as an unary prefix operator.
  `*int_pointer`
- If a pointer variable `p` has the type `t*`, then the expression `*p` has the type `t`

```
// Program to illustrate pointers
#include <iostream>
int main (void)
{
    int count = 10, x;
    int *int_pointer;
    int_pointer = &count;
    x = *int_pointer;
    cout<<"count ="<< count<<endl;
    cout<<"x= x<<endl;
    return 0;
}
```



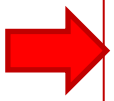
Example: pointers

```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
   int count = 10;
   int *ip;
   ip = &count;
   printf ("count = %i, *ip = %i\n", count, *ip);
   *ip=4;
   printf ("count = %i, *ip = %i\n", count, *ip);
   return 0;
}
```

Using pointer variables

- The value of a pointer in C is meaningless until it is set pointing to something !

```
int *p;
*p = 4;
```



Severe runtime error !!! **the value 4 is stored in the location to which p points. But p, being uninitialized, has a random value, so we cannot know where the 4 will be stored !**

- How to set pointer values: Using the address operator

```
int *p;
int  x;
p = &x;
*p = 4;
```

- Using directly assignements between pointer variables

```
int *p;
int *p1;
int  x;
p1 = &x;
p = p1;
*p = 4;
```

NULL pointers

- Values of a pointer variable:
  - Usually the value of a pointer variable is a pointer to some other variable
  - Another value a pointer may have: it may be set to a *null pointer*
- A *null pointer* is a special pointer value that is known not to point anywhere.
- No other valid pointer, to any other variable, will ever compare equal to a null pointer !
- Predefined constant NULL, defined in <stdio.h>
- Good practice: test for a null pointer before inspecting the value pointed !

```
#include <stdio.h>

int *ip = NULL;

if(ip != NULL)  printf("%d\n", *ip);

if(ip )  printf("%d\n", *ip);
```

const and pointers

- **With pointers, there are two things to consider:**
  - **whether the pointer will be changed**
  - **whether the value that the pointer points to will be changed.**
- Assume the following declarations:

```
char c = 'X';
char *charPtr = &c;
```

- If the pointer variable is always set pointing to c, it can be declared as a const pointer as follows:

```
char * const charPtr = &c;
*charPtr = 'Y'; // this is valid
charPtr = &d;   // not valid !!!
```

- If the location pointed to by charPtr will not change *through the pointer variable charPtr*, that can be noted with a declaration as follows:

```
const char *charPtr = &c;
charPtr = &d;   // this is valid
*charPtr = 'Y'; // not valid !!!
```

Pointers and Function Arguments

- Recall that the C language passes arguments to functions by value (except arrays)
- there is no direct way for the called function to alter a variable in the calling function.

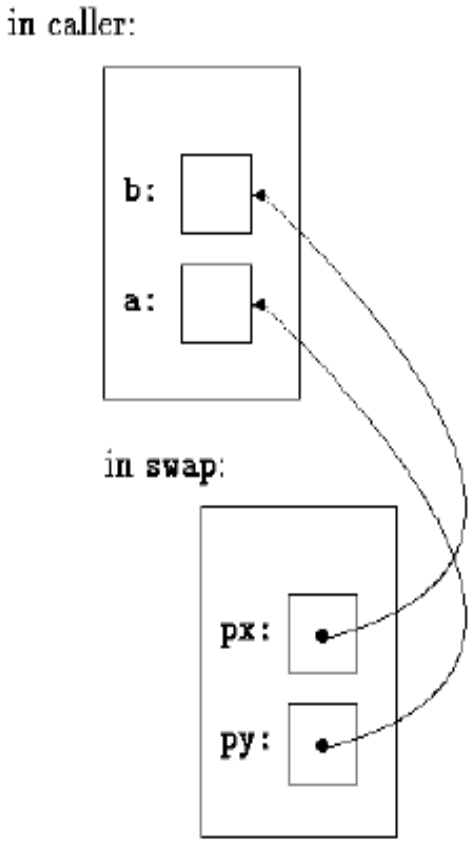
```
void swap(int x, int y)   /* WRONG */
    {
        int temp;
        temp = x;
        x = y;
        y = temp;
    }

swap(a,b);
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps *copies* of a and b.

Pointers and Function Arguments

- *If it is necessary that a function alters its arguments, the caller can pass pointers to the values to be changed*
- Pointer arguments enable a function to access and ***change*** variables in the function that called it

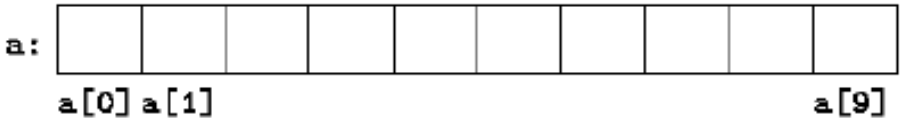
```
void swap(int *px, int *py)
/* interchange *px and *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int a=3, b=5;
swap(&a, &b);
```



Pointers and arrays

- In C, there is a strong relationship between pointers and arrays
- Any operation that can be achieved by array subscripting can also be done with pointers

```
int a[10];
```




```
int *pa;
pa=&a[0];
```

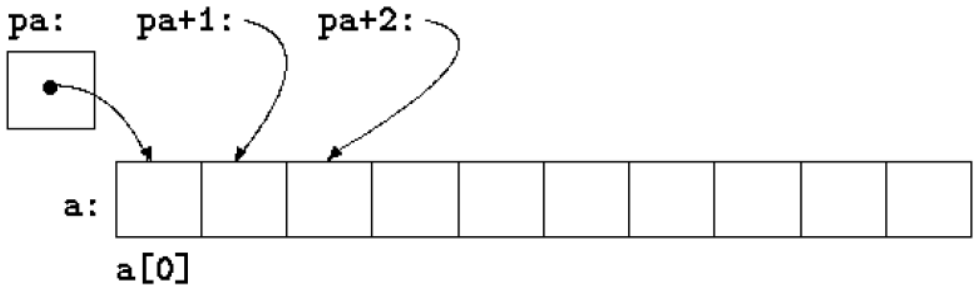
Or

```
pa=a;
```

The value of a variable of type array is the address of element zero of the array.
The name of an array is a synonym for the location of the initial element.

Pointers and Arrays

- If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before.
- If pa points to a[0], *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].
- The value in a[i] can also be written as *(a+i). The address &a[i] and a+i are also identical
- These remarks are true regardless of the type or size of the variables in the array a !



Arrays are **constant** pointers

```
int a[10];
int *pa;

pa=a;

pa++;
```

OK. Pointers are variables that can be assigned or incremented

```
int a[10];
int *pa;

a=pa;

a++;
```

Errors !!!

The name of an array is a CONSTANT having as a value the location of the first element.
You cannot change the address where the array is stored !
An array's name is equivalent with a ***constant*** pointer

Arrays as parameters

- When an array name is passed to a function, what is passed is the location of the first element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.
- As formal parameters in a function definition, T `s[]` and T `*s` are equivalent, for any type T; The latter is preferred because it says more explicitly that the variable is a pointer.
- Examples:
- f(int arr[]) { ... } is equivalent with f(int *arr) { ... }
- f(char s[]) { …} is equivalent with f(char *s) { …}

Example: Arrays as parameters

```
void print1(int tab[], int N) {
    int i;
    for (i=0; i<N; i++)
        printf("%d ",tab[i]);
}
void print2(int tab[],int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}
void print3(int *tab,int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}
void print4(int *tab,int N) {
    int i;
    for (i=0; i<N; i++, tab++)
        printf("%d ", *tab);
}
```

The formal parameter can be declared as array or pointer !
In the body of the function, the array elements can be accessed through indexes or pointers !

```
void main(void) {
    int a[5]={1,2,3,4,5};
    print1(a,5);
    print2(a,5);
    print3(a,5);
    print4(a,5);
}
```

Example: Arrays as parameters

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

The actual parameter can be declared as array or pointer !

```
char array[100]="Hello, world";
char *ptr="Hello, world";

strlen("Hello, world"); /* string constant */
strlen(array); /* char array[100]; */
strlen(ptr);   /* char *ptr; */
```

Example: Arrays as parameters

```
int strlen(char *s)
{
   if (*s=='\0')
      return 0;
   else
        return 1 + strlen(++s);
}
```

The recursive call gets as parameter the subarray starting with the second element

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray.
For example, if a is an array, f(&a[2]) and f(a+2) both pass to the function f the address of the subarray that starts at a[2].