# SDN150S – SOFTWARE DESIGN 1

## Pointers

DEPARTMENT OF ELECTRICAL ELECTRONIC AND COMPUTER ENGINEERING
CAPE PENINSULA UNIVERSITY OF TECHNOLOGY

Cape Peninsula University of Technology

# OUTLINE

- What is are Pointers?
- How to use Pointers
- Null Pointers
- Pointer in Detail
- Pointer Arithmetic
- Comparing Pointers
- Pointer to Pointer
- Passing Pointers to Functions
- Return Pointer from Functions

# WHAT ARE POINTERS? (1 OF 2)

- A pointer is a variable whose value is the address of another variable (i.e., the direct address of a memory location).
- Like any variable or constant, you must declare a pointer before using it.
- The general form of a pointer variable declaration is given in the syntax below:

```
type *varname;
```

- Where, `type` is the pointer's base type (a valid C data type) and `varname` is the name of the pointer variable.
- Here asterisk (*) is used to designate a variable as a pointer (although asterisk is usually used for multiplication).

# WHAT ARE POINTERS? (2 OF 2)

- Below are some examples of the valid pointer declarations:
  - `int *ip; /* pointer to an integer */`
  - `double *dp; /* pointer to a double */`
  - `float *fp; /* pointer to a float */`
  - `char *ch /* pointer to a character */`
- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address.
- The only difference between pointers of different data types is the data type of the variable or constant which the pointer points to.

# HOW TO USE POINTERS (1 OF 2)

- Like must functions they're three important operations frequently used in pointers:

  (a) Define the pointer variable

  (b) Assign the address of a variable to a pointer

  (c) Access the value via the address of the pointer variable.

- This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.
- The following example makes use of these operations:

# HOW TO USE POINTERS (2 OF 2)

```c
#include <stdio.h>
int main ()
{
int var = 20; /* actual variable declaration */
int *ip; /* pointer variable declaration */
ip = &var; /* store address of var in pointer variable*/
printf("Address of var variable: %p\n", &var );
/* address stored in pointer variable */
printf("Address stored in ip variable: %p\n", ip );
/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
return 0;
}
```

**OUTPUT**

```
Address of var variable:  0x7ffd6692cb7c
Address stored in ip variable:  0x7ffd6692cb7c
Value of *ip variable: 20
```

# NULL POINTERS

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned.
- This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main ()
{
int *ptr = NULL;
printf("The value of ptr is : %x\n", ptr );
return 0;
}
```

**OUTPUT**
The value of ptr is 0

The %p is used to print the pointer value, and %x is used to print hexadecimal values.

# POINTERS IN DETAIL

- Pointers have many easy concepts that are very important to C programming. Such as:

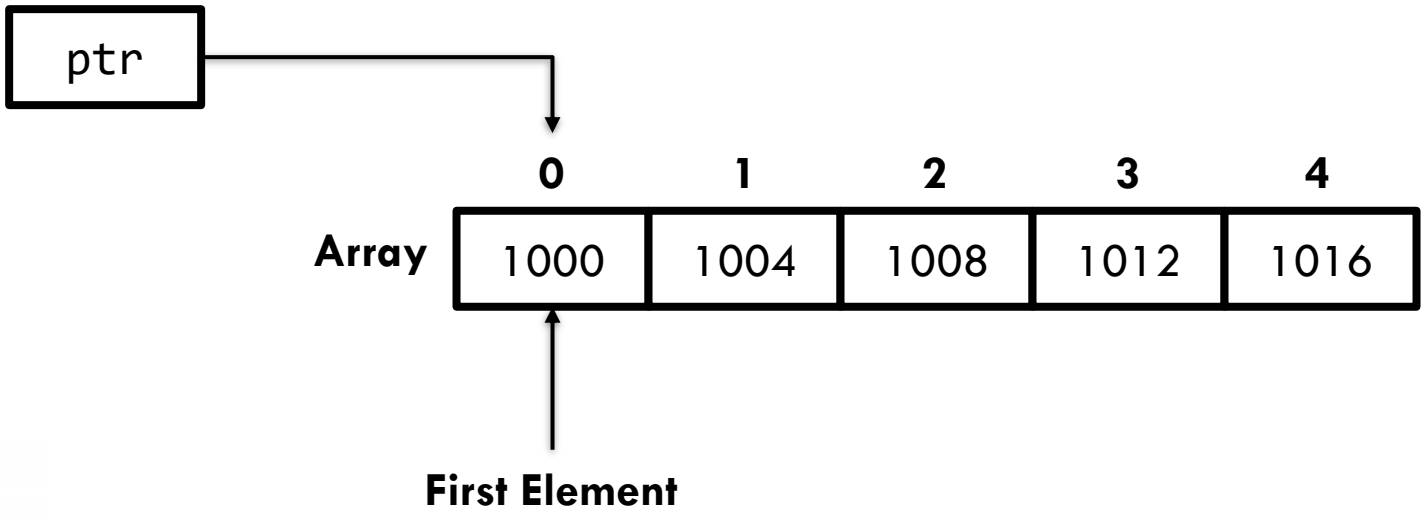
| Concept | Description |
|---|---|
| Pointer arithmetic | There are four arithmetic operators that can be used in pointers: ++, --, +, - |
| Array of pointers | You can define arrays to hold a number of pointers. |
| Pointer to pointer | C allows you to have pointer on a pointer and so on. |
| Passing pointers to functions in C | Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function. |
| Return pointer from functions in C | C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well. |

# POINTER ARITHMETIC (1 OF 7)

- A pointer in C is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and -
- To understand pointer arithmetic, let us consider that `ptr` is an integer pointer which points to the address 1000.
- Assuming each integer is allocated 4 bits, let us perform the following arithmetic operation on the pointer: `ptr++`
- After the above operation, the `ptr` will point to the location 1004 because each time `ptr` is incremented, it will point to the next integer location which is 4 bytes next to the current location

# POINTER ARITHMETIC (2 OF 7)

- This operation will move the pointer to the next memory location without impacting the actual value at the memory location.
- Similarly, if `ptr` points to a character whose address is 1000, then an incremental operation will point to the location 1001, if the next character will be available at 1001.



# POINTER ARITHMETIC (3 OF 7)

**Incrementing Pointer:**

- The following example increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;
/* let us have array address in pointer */
ptr = var;
for ( i = 0; i < MAX; i++)
{
printf("Address of var[%d] = %p\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the next location */
ptr++;
}
return 0;
}
```

**OUTPUT**

```
Address of var[0] = 0x7fffe41ecb9c
Value of var[0] = 10
Address of var[1] = 0x7fffe41ecba0
Value of var[1] = 100
Address of var[2] = 0x7fffe41ecba4
Value of var[2] = 200
```

# POINTER ARITHMETIC (4 OF 7)

- **Decrementing pointer:**
  - The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;
/* let us have array address in pointer */
ptr = &var[MAX-1];
for ( i = MAX; i > 0; i--)
{
printf("Address of var[%d] = %p\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the previous location */
ptr--;
}
return 0;
}
```

**OUTPUT**

```
Address of var[3] = 0x7ffccbcd0be4
Value of var[3] = 200
Address of var[2] = 0x7ffccbcd0be0
Value of var[2] = 100
Address of var[1] = 0x7ffccbcd0bdc
Value of var[1] = 10
```

# POINTER ARITHMETIC (5 OF 7)

- **Adding pointer:**
  - When a pointer is added with a value, the value is first multiplied by the size of data type and then added to the pointer.

```
#include <stdio.h>
int main()
{
    int N[] = {1,2,3,4}; // Integer variable
    int *ptr; // Pointer to an integer
    ptr = &N[0]; // Pointer stores the address of N
    printf("Pointer ptr before Addition: ");
    printf("%d = %p \n", *ptr, ptr);
    // Addition of 3 to ptr
    ptr = ptr + 3;
    printf("Pointer ptr after Addition: ");
    printf("%d = %p \n", *ptr, ptr);
    return 0;
}
```

**OUTPUT**

Pointer ptr before Addition: 1 = 0x7ffd4325aa50
Pointer ptr after Addition: 4 = 0x7ffd4325aa5c

# POINTER ARITHMETIC (6 OF 7)

- **Subtracting pointer:**
  - When a pointer is subtracted with a value, the value is first multiplied by the size of the data type and then subtracted from the pointer.

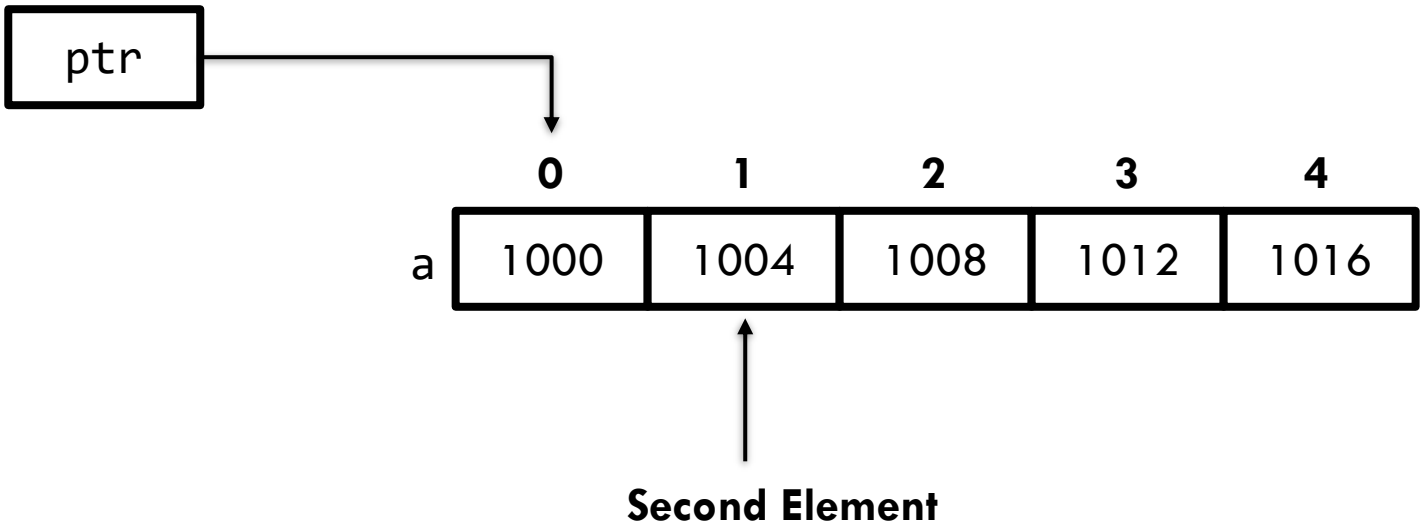
```
#include <stdio.h>
int main()
{
    int N[] = {1,2,3,4}; // Integer variable
    int *ptr; // Pointer to an integer
    ptr = &N[4]; // Pointer stores the address of N
    printf("Pointer ptr before Subtraction: ");
    printf("%d = %p \n", *ptr, ptr);
    // Addition of 3 to ptr
    ptr = ptr - 3;
    printf("Pointer ptr after Subtraction : ");
    printf("%d = %p \n", *ptr, ptr);
    return 0;
}
```

**OUTPUT**

Pointer ptr before Subtraction: 4 = 0x7ffd4325aa5c
Pointer ptr after Subtraction: 1 = 0x7ffd4325aa50

# POINTER ARITHMETIC (7 OF 7)

Ptr = &a[0] + 1 ≡ 1000 + 1 × 4 = 1004



# COMPARING POINTER (1 OF 2)

- Pointers may be compared by using relational operators, such as ==, <, and >.
- If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example - one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

# COMPARING POINTER (2 OF 2)

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;
/* let us have address of the first element in pointer */
ptr = var;
i = 0;
while ( ptr <= &var[MAX - 1] )
{
printf("Address of var[%d] = %p\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* point to the previous location */
ptr++;
i++;
}
return 0;
}
```

**OUTPUT**

Address of var[0] = 0x7ffc57e006fc
Value of var[0] = 10
Address of var[1] = 0x7ffc57e00700
Value of var[1] = 100
Address of var[2] = 0x7ffc57e00704
Value of var[2] = 200

# POINTER TO POINTER (1 OF 2)

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.
- Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.
- A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int: `int **var;`

# POINTER TO POINTER (2 OF 2)

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.

```
#include <stdio.h>
int main ()
{
int var;
int *ptr;
int **pptr;
var = 3000;
/* take the address of var */
ptr = &var;
/* take the address of ptr using address of operator & */
pptr = &ptr;
/* take the value using pptr */
printf("Value of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

**OUTPUT**

Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000

# PASSING POINTERS TO FUNCTIONS (1 OF 2)

- C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- The example below passes an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```c
#include <stdio.h>
#include <time.h>
void getSeconds(unsigned long *par);
int main ()
{
unsigned long sec;
getSeconds( &sec );
/* print the actual value */
printf("Number of seconds: %ld\n", sec )
;
return 0;
}
```

```c
void getSeconds(unsigned long *par)
{
/* get the current number of seconds */
*par = time( NULL );
return;
}
```

**OUTPUT**

Number of seconds :1653982789

# PASSING POINTERS TO FUNCTIONS (2 OF 2)

- The function, which can accept a pointer, can also accept an array as shown in the following example:

```c
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);
int main ()
{
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
double avg;
/* pass pointer to the array as an argument */
avg = getAverage( balance, 5 ) ;
/* output the returned value */
printf("Average value is: %f\n", avg );
return 0;
}
```

```c
double getAverage(int *arr, int size)
{
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = (double)sum / size;
return avg;
}
```

**OUTPUT**

Average value is: 214.40000

# RETURN POINTER FROM FUNCTIONS (1 OF 2)

- Similar to return an array from a function, C also allows to return a pointer from a function.
- To do so, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
.
 // body of function
.
}
```

- Also, to return the address of a local variable outside the function, you would have to define the local variable as static variable.

# RETURN POINTER FROM FUNCTIONS (2 OF 2)

- Let consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

```c
#include <stdio.h>
#include <time.h>
/* function to generate and retrun random numbers. */
int * getRandom( )
{
static int r[10];
int i;
/* set the seed */
srand( (unsigned)time( NULL ) );
for ( i = 0; i < 10; ++i)
{
r[i] = rand();
printf("%d\n", r[i] );
}
return r;
}
```

```c
/* main function to call above defined function */
int main ()
{
/* a pointer to an int */
int *p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf("*(p + [%d]) : %d\n", i, *(p + i) );
}
return 0;
}
```