# CS1010E Lecture 10
# Addresses and Pointers

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
`www.comp.nus.edu.sg/∼joxan`
`cs1010e@comp.nus.edu.sg`

Semester II, 2016/2017

# Lecture Outline

- Addresses and Pointers.
- Pointers to Array Elements.
- Pointers in Function References.

# Addresses and Pointers

- When a C program is executed, memory locations are assigned to the variables used in the program.
- Each of these memory locations has a positive integer **address** that uniquely defines the location.
- When a variable is assigned a value, this value is stored in the corresponding memory location.
- The value of a variable can be used by statements in the program, and it can be changed by statements in the program.
- The specific addresses used for the variables are determined each time that the program is executed and may vary from one execution to another.

# Address Operator

- In C, the address of a variable can be referenced using the **address operator** `&`.
- This operator was introduced in Lecture 3 in conjunction with the `scanf` statement.
- For example, a statement to read a `double` value from the keyboard and to store it in the variable `x` is the following:
  - `scanf("%lf",&x);`
- This statement specifies that the value read from the keyboard is to be stored at the address specified by `&x`, the address of `x`.

# Address Operator

```
/*-----------------------------------------------------------*/
/*   Program chapter6_1                                      */
/*                                                           */
/*   This program demonstrates the relationship between     */
/*   variables and addresses.                                */

#include <stdio.h>

int main(void)
{
    /*  Declare and initialize variables.  */
    int a=1, b=2;

    /*  Print the contents and addresses of a and b.  */
    printf("a = %d;  address of a = %p \n",a,&a);
    printf("b = %d;  address of b = %p \n",b,&b);

    /*  Exit program.  */
    return 0;
}
```

# Address Operator

- Note that the address is printed with a `%p` specification that is used for printing addresses in hexadecimal.

- A sample output from this program is:

  -
    ```
    a = 1;  address of a = ffbff894
    b = 2;  address of b = ffbff890
    ```

- Note that the addresses are system dependent and may vary from one execution to another.

- In the next program, no initial values are given to variables `a` and `b`.

# A Note on Hexadecimal Numbers

- An address is a number *n* denoting the $n^{th}$ byte.
- A variable, including a pointer, comprises *several* bytes.
- In examples, it is more intuitive to display address values in hexadecimal (or base 16) format.
- There are 16 hex digits: the symbols 0-9 to represent values zero to nine, and the letters a-f to represent values ten to fifteen.
  For example, the hexadecimal number 2af3 is equal, in decimal, to
  $(2 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (3 \times 16^0), \quad \text{or} \quad 10,995$
- One hex digit is sometimes called a "nibble".
  Thus two nibbles corresponds to one byte.

# Address Operator

```
/*-------------------------------------------------------*/
/*  Program chapter6_2                                   */
/*                                                       */
/*  This program demonstrates the relationship between   */
/*  variables and addresses.                             */
#include <stdio.h>

int main(void)
{
    /*  Declare and initialize variables.  */
    int a, b;

    /*  Print the contents and addresses of a and b.  */
    printf("a = %d;  address of a = %p \n",a,&a);
    printf("b = %d;  address of b = %p \n",b,&b);

    /*  Exit program.  */
    return 0;
}
```

# Address Operator

- A sample output from this program is:

  -
    ```
    a = -4196076;  address of a = ffbff894
    b = 4;  address of b = ffbff890
    ```
- While we see that there are values in the variables (even though we have not assigned any in the program), we should not assume anything about these values.
- This example illustrates the importance of being sure that a program initializes a variable before using its value in other statements.

# Pointer Assignment

- The C language allows us to store the address of a memory location in a special type of variable called a **pointer**.

- When a pointer is defined, the type of variable to which it will point must also be defined.

- Thus, a pointer defined to point to an integer variable cannot also be used to point to a floating-point variable.

# Pointer Assignment

- Consider a declaration that defines two integer variables and a pointer to an integer value.
  - `int a, b, *ptr;`

- C uses an asterisk to indicate that the variable is a pointer.

- When used in a statement (not a declaration), this asterisk has a different meaning and is then called a **dereferencing** or **indirection** operator.

# Pointer Assignment

- The declaration on the previous slide specifies that memory addresses should be assigned to three variables—two integer variables and a pointer to an integer variable.

- The statement does not specify the initial values for `a` and `b`, and it also does not specify an address to be stored in `ptr`.

- The memory snapshot after this declaration is:
  - `a` $\boxed{\texttt{?}}$ `b` $\boxed{\texttt{?}}$ `ptr` $\longrightarrow$ `?`

# Pointer Assignment

- To specify that `ptr` should point to the variable `a`, we could use an assignment statement that stores the address of `a` in `ptr`:

  -
    ```
    int a, b, *ptr;
    ptr = &a;
    ```
- This assignment could also have been made on the declaration statement:
  - `int a, b, *ptr=&a;`
- In either case, the memory snapshot after the declaration is the following:
  - $\mathtt{a}\ \boxed{\mathtt{?}}\longleftarrow \mathtt{ptr} \qquad \mathtt{b}\ \boxed{\mathtt{?}}$

# Pointer Assignment

- Consider this set of statements:

  -
    ```
    int a=5, b=9, *ptr=&a;
    b = *ptr;
    ```

- The asterisk in the first line states that `ptr` is a pointer. The asterisk in the second line is a **dereferencing** or **indirection** operator.

- The second line is read as “`b` is assigned the value at the address contained in `ptr`” or “`b` is assigned the value pointed to by `ptr`”.

# Pointer Assignment

- The memory snapshot after the first line is the following:
    - $\texttt{a}\ \boxed{\texttt{5}}\longleftarrow\texttt{ptr}\qquad\texttt{b}\ \boxed{\texttt{9}}$
- The memory snapshot after the second line is the following:
    - $\texttt{a}\ \boxed{\texttt{5}}\longleftarrow\texttt{ptr}\qquad\texttt{b}\ \boxed{\texttt{5}}$
- Thus, `b` is assigned the value pointed to by `ptr`.

# Pointer Assignment

- Now consider this set of statements:

  -
    ```
    int a=5, b=9, *ptr=&a;
    *ptr = b;
    ```
- The second line is read as “assign the value of `b` to the variable to which `ptr` points”.

# Pointer Assignment

- The memory snapshot after the first line is the following:
    - $\texttt{a}\ \boxed{\texttt{5}} \longleftarrow \texttt{ptr} \qquad \texttt{b}\ \boxed{\texttt{9}}$
- The memory snapshot after the second line is the following:
    - $\texttt{a}\ \boxed{\texttt{9}} \longleftarrow \texttt{ptr} \qquad \texttt{b}\ \boxed{\texttt{9}}$
- Thus, the value pointed to by `ptr` is assigned the value in `b`.

# Address Operator

```
/*------------------------------------------------------*/
/*  Program chapter6_3                                  */
/*                                                      */
/*  This program demonstrates the relationship between  */
/*  variables, addresses, and pointers.                 */
#include <stdio.h>
int main(void)
{
    /*  Declare and initialize variables.  */
    int a=1, b=2, *ptr=&a;

    /*  Print the variable and pointer contents.  */
    printf("a = %d;  address of a = %p \n",a,&a);
    printf("b = %d;  address of b = %p \n",b,&b);
    printf("ptr = %p;  address of ptr = %p \n",ptr,&ptr);
    printf("ptr points to the value %d \n",*ptr);

    return 0;
}
```

# Pointer Assignment

- A sample output from this program is:

  -
    ```
    a = 1;  address of a = ffbff894
    b = 2;  address of b = ffbff890
    ptr = ffbff894;  address of ptr = ffbff88c
    ptr points to the value 1
    ```
- Note that the values of the pointer to `a` and the address of `a` are the same.

# Address Arithmetic

- The operations that can be performed with pointers (or addresses) are limited to the following:
  - A pointer can be assigned to another pointer of the same type.
  - A pointer can be assigned to or compared with the integer zero or to the symbolic constant `NULL`, which is defined in `<stdio.h>`.
- If a pointer is not initially assigned to a memory location, give it a `NULL` value to indicate that the pointer has not been assigned to a memory location.

# Address Arithmetic

- A pointer can point to only one location, but several pointers can point to the same location. Both `ptr_1` and `ptr_2` point to the same variable after we execute the following statements:

  -
    ```
    int x=5, y=8, *ptr_1, *ptr_2;
    ptr_1 = &x;
    ptr_2 = ptr_1;
    ```
- The memory snapshot after these statements are executed is the following:
  - $$\texttt{ptr\_1} \longrightarrow \overset{\texttt{x}}{\boxed{5}} \longleftarrow \texttt{ptr\_2} \qquad \texttt{y}\ \boxed{8}$$

# Address Arithmetic

- To illustrate some common errors that can be made when working with pointers, we now present several invalid statements using these variables:
    - `&y = ptr_1; /* Attempts to change the address of y. */`
    - `ptr_1 = y; /* Attempts to change ptr_1 to a nonaddress value. */`
    - `*ptr_1 = ptr_2; /* Attempts to move an address to an integer variable. */`
    - `ptr_1 = *ptr_2; /* Attempts to change ptr_1 to a nonaddress value. */`

# Address Arithmetic

- When simple variables are defined, we should not make any assumptions about the relationships of the memory locations assigned to the variables.

- For example, if a declaration statement defines two integers, `a` and `b`, we should not assume that the values are adjacent in memory; we also should not make assumptions about which value occurs first in memory.

- The memory assignment of a simple variable is system dependent.

# Address Arithmetic

- The memory assignment for an array is guaranteed to be a sequential group of memory locations.

- Thus, if array `x` contains five integers, then the memory location for `x[1]` will immediately follow the memory location for `x[0]`, and the memory location for `x[2]` will immediately follow the memory location for `x[1]`, and so on.

# Address Arithmetic

- Therefore, if `ptr_x` is a pointer to an integer, we can initialize it to point to the integer `x[0]` with this statement:
  - `ptr_x = &x[0];`
- An array name can also be used to represent the address of the first element of the array. Thus, the following statement is equivalent to the one above:
  - `ptr_x = x;`

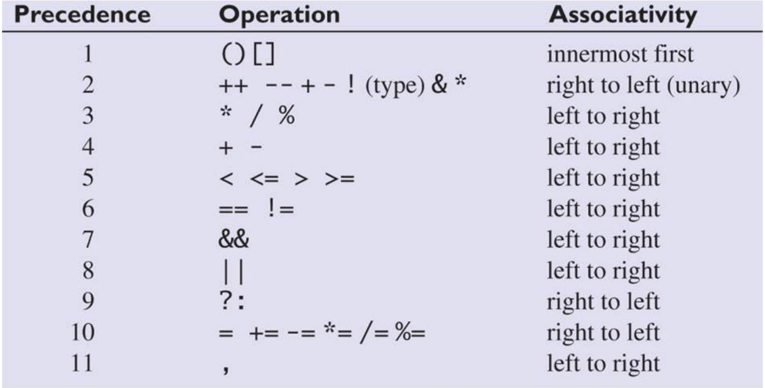
# Address Arithmetic

- To move the pointer to `x[1]`, we can increment `ptr_x` by 1, which causes it to point to the value that follows `x[0]`, or we can assign `ptr_x` the address of `x[1]`.
- Thus, we could use any of the following statements:
  - `++ptr_x; /* Increment ptr_x to point to the next value in memory. */`
  - `ptr_x++; /* Increment ptr_x to point to the next value in memory. */`
  - `ptr_x += 1; /* Increment ptr_x to point to the next value in memory. */`
  - `ptr_x = &x[1]; /* ptr_x is assigned the address of x[1]. */`

# Address Arithmetic

- Similarly, the following statement increments `ptr_x` to point to the value that is `k` values past the one to which `ptr_x` pointed before this statement was executed.
  - `ptr_x += k;`
- These are all examples of adding integers to pointers.
- Similarly, integers can also be subtracted from pointers.

# Operator Precedence Table



| Precedence | Operation | Associativity |
|---|---|---|
| 1 | () [] | innermost first |
| 2 | ++ -- + - ! (type) & * | right to left (unary) |
| 3 | * / % | left to right |
| 4 | + - | left to right |
| 5 | < <= > >= | left to right |
| 6 | == != | left to right |
| 7 | && | left to right |
| 8 | \|\| | left to right |
| 9 | ?: | right to left |
| 10 | = += -= *= /= %= | right to left |
| 11 | , | left to right |

# Pointers to One-Dimensional Arrays

Suppose we have

```
double x[6] = {1.5, 2.2, 4.3, 7.5, 9.1, 10.5}
ptr = &x[0];
```

Then the following are equivalent:

- ptr = &x[0];
- ptr = x;

More in detail: the following are equivalent:

- for (k = 0; k <= 5; k++) sum += x[k];
- for (k = 0; k <= 5; k++) sum += *(ptr + k);

**Note:** the expression *(ptr + k) is different from *ptr + k.

# Pointers to Two-Dimensional Arrays

The following are equivalent:

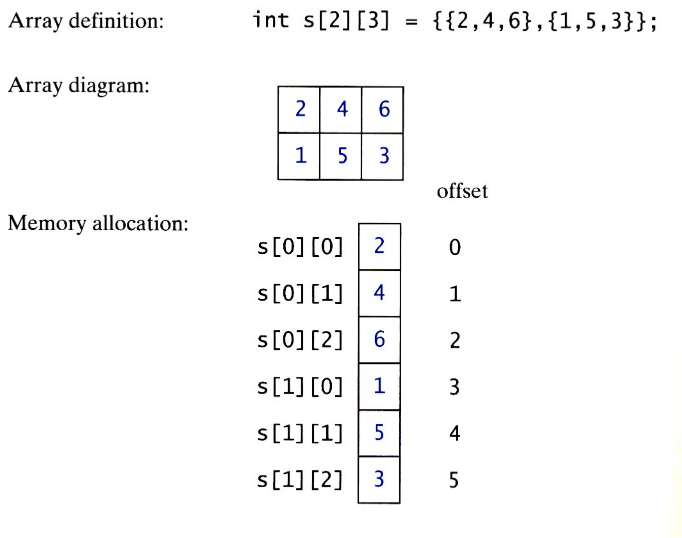
```
int s[2][3], srows = 2, scols = 3, i, j, sum = 0;
...
/* sum the values in array s */
for (i = 0; i <= srows-1; i++)
   for (j = 0; j <= scols-1; j++)
      sum += s[i][j];
```

---

```
int s[2][3], s_count = 6, k, sum = 0, *ptr = &s[0][0];
...
/* sum the values in array s */
for (k = 0; k <= s_count-1; k++)
   sum += *(ptr + k);
```

**Note:** the second code fragment only needed one loop.

# Pointers to Two-Dimensional Arrays



Array definition: `int s[2][3] = {{2,4,6},{1,5,3}};`

Array diagram:

| | | |
|---|---|---|
| 2 | 4 | 6 |
| 1 | 5 | 3 |

Memory allocation:

| | | offset |
|---|---|---|
| s[0][0] | 2 | 0 |
| s[0][1] | 4 | 1 |
| s[0][2] | 6 | 2 |
| s[1][0] | 1 | 3 |
| s[1][1] | 5 | 4 |
| s[1][2] | 3 | 5 |

# Pointers in Function References

- In C, most function references are call-by-value references.
- Thus, the values of the actual parameters are copied to the formal parameters.
- All computations in the function use the formal parameters, and thus an actual parameter cannot be changed in a function.

# Pointers in Function References

- One exception to this rule was presented in Lecture 8; when an array name is used as an argument in a function reference, the address of the array is transferred to the function, and all references to array values use the actual array locations.
- Thus, values in an array can be modified by statements within a function.
- Other exceptions can be implemented using pointers as function parameters.
- To illustrate the use of pointers as function parameters, we develop a function that exchanges the contents of two memory locations.

# Pointers in Function References

- Recall that it takes three statements to switch the values in two locations.
    - `a` $\boxed{5}$ `b` $\boxed{7}$ `hold` $\boxed{?}$
    - `hold = a;`<br>
      `a` $\boxed{5}$ `b` $\boxed{7}$ `hold` $\boxed{5}$
    - `a = b;`<br>
      `a` $\boxed{7}$ `b` $\boxed{7}$ `hold` $\boxed{5}$
    - `b = hold;`<br>
      `a` $\boxed{7}$ `b` $\boxed{5}$ `hold` $\boxed{5}$
- In problem solutions that switch several values, it would be convenient to access a function to perform the switch.

# Pointers in Function References

```
/*--------------------------------------------------------*/
/*  Incorrect function to switch values in two variables. */
/*  Uses call-by-value.                                   */

void switch1(int a, int b)
{
    /*  Declare variables.  */
    int hold;

    /*  Switch values in a and b.  */
    hold = a;
    a = b;
    b = hold;

    /*  Void return.  */
    return;
}
```

# Pointers in Function References

- Assume that the following statement references this function:
  - `switch1(x,y);`

- If `x` and `y` contain the values 5 and 7, then the transfer of the values from the actual parameters to the formal parameters at the beginning of the function execution is the following:
  - actual parameters    formal parameters

    $\texttt{x}\ \boxed{5} \quad \longrightarrow \quad \texttt{a}\ \boxed{5}$

    $\texttt{y}\ \boxed{7} \quad \longrightarrow \quad \texttt{b}\ \boxed{7}$

# Pointers in Function References

- After the function is executed, the values of the actual parameters and formal parameters are as follows:
  - actual parameters formal parameters

    $$\mathtt{x}\ \boxed{5} \quad \longrightarrow \quad \mathtt{a}\ \boxed{7}$$

    $$\mathtt{y}\ \boxed{7} \quad \longrightarrow \quad \mathtt{b}\ \boxed{5}$$

- Since this function uses call-by-value, the value of the actual parameters (not the address) is passed to the formal parameters.

- The values have been switched in the formal parameters, but these values are not transferred back to the actual parameters.

# Pointers in Function References

- After considering the incorrect solution, we are now ready to develop a function that switches the contents of two simple variables using pointers.

- The function has two parameters that are pointers to the two variables that we want to switch.

- A prototype statement for this function is the following:
  - `void switch2(int *a, int *b);`

- Thus, the function does not return a value, and its two parameters are pointers to integers.

- The correct function appears on the next slide.

# Pointers in Function References

```
/*------------------------------------------------------*/
/*  Correct function to switch values in two variables. */
/*  Uses call-by-reference.                             */

void switch2(int *a, int *b)
{
    /*  Declare variables.  */
    int hold;

    /*  Switch values in a and b.  */
    hold = *a;
    *a = *b;
    *b = hold;

    /*  Void return.  */
    return;
}
```

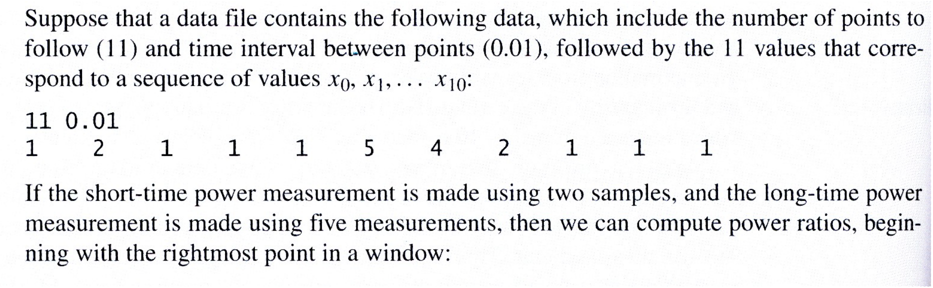
# Pointers in Function References

- If `x` and `y` are simple integer variables, then a valid call to this function is:
    - `switch2(&x,&y);`
- If `ptr_1` points to variable `x` and `ptr_2` points to variable `y`, then the values in `x` and `y` can be switched with this reference:
    - `switch2(ptr_1,ptr_2);`

# Example: Seismic Event Detection

Seismometers records a sequence of values from 1 to 10 according to the *Richter scale*. The sequence represents values taken over a uniform time interval between successive values.

For a given time, a *short-time window* is a small number of values around that time, and a *long-time window* is a bigger number of values around that time. Consider the *average squared value* of the these two sequences of values. We are interested to detect when the ratio of these two averages exceeds a certain threshold.

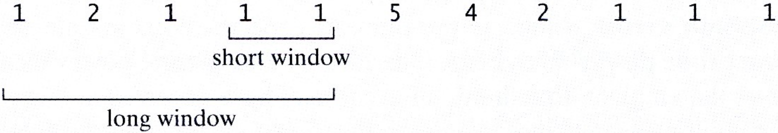


Suppose that a data file contains the following data, which include the number of points to follow (11) and time interval between points (0.01), followed by the 11 values that correspond to a sequence of values $x_0, x_1, \ldots x_{10}$:

```
11 0.01
1    2    1    1    1    5    4    2    1    1    1
```

If the short-time power measurement is made using two samples, and the long-time power measurement is made using five measurements, then we can compute power ratios, beginning with the rightmost point in a window:

# Example: Seismic Event Detection

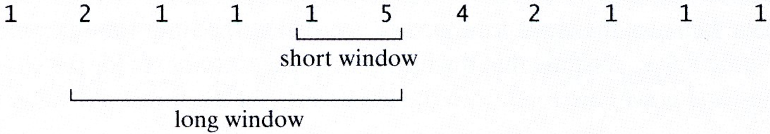


1 2 1 1 1 5 4 2 1 1 1

short window

long window

Point $x_4$: Short-time power = (1 + 1)/2 = 1,
Long-time power = (1 + 1 + 1 + 4 + 1)/5 = 1.6,
Ratio = 1/1.6 = 0.63.



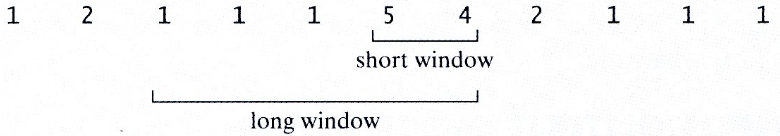
1 2 1 1 1 5 4 2 1 1 1

short window

long window

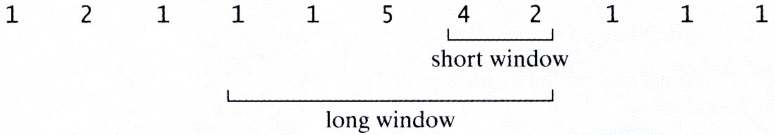
Point $x_5$: Short-time power = (25 + 1)/2 = 13,
Long-time power = (25 + 1 + 1 + 1 + 4)/5 = 6.4,
Ratio = 13/6.4 = 2.03.

# Example: Seismic Event Detection



Point $x_6$:

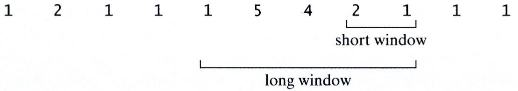
Short-time power = (16 + 25)/2 = 20.5,
Long-time power = (16 + 25 + 1 + 1 + 1)/5 = 8.8,
Ratio = 20.5/8.8 = 2.33.



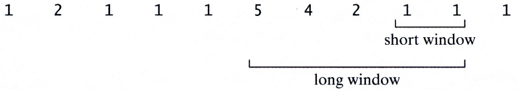
Point $x_7$:

Short-time power = (4 + 16)/2 = 10,
Long-time power = (4 + 16 + 25 + 1 + 1)/5 = 9.4,
Ratio = 10/9.4 = 1.06.

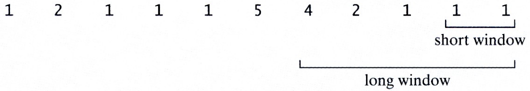
# Example: Seismic Event Detection



Point $x_8$: Short-time power = (1 + 4)/2 = 2.5,
Long-time power = (1 + 4 + 16 + 25 + 1)/5 = 9.4,
Ratio = 2.5/9.4 = 0.27.

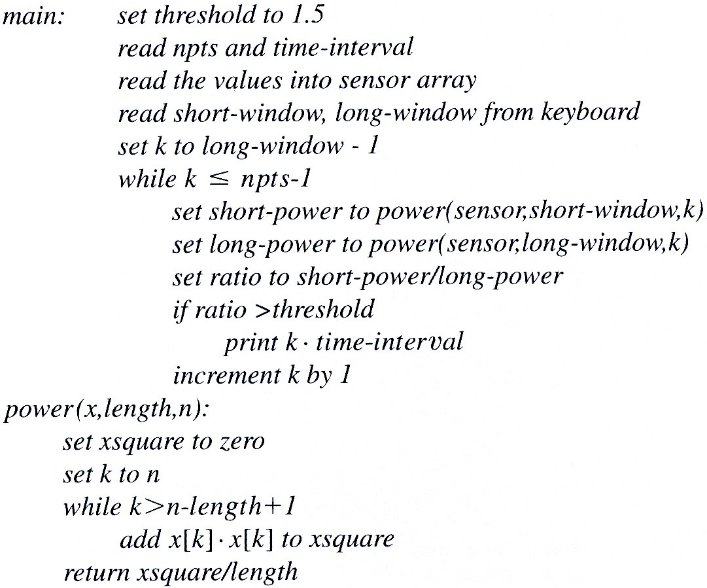


Point $x_9$: Short-time power = (1 + 1)/2 = 1,
Long-time power = (1 + 1 + 4 + 16 + 25)/5 = 9.4,
Ratio = 1/9.4 = 0.11.



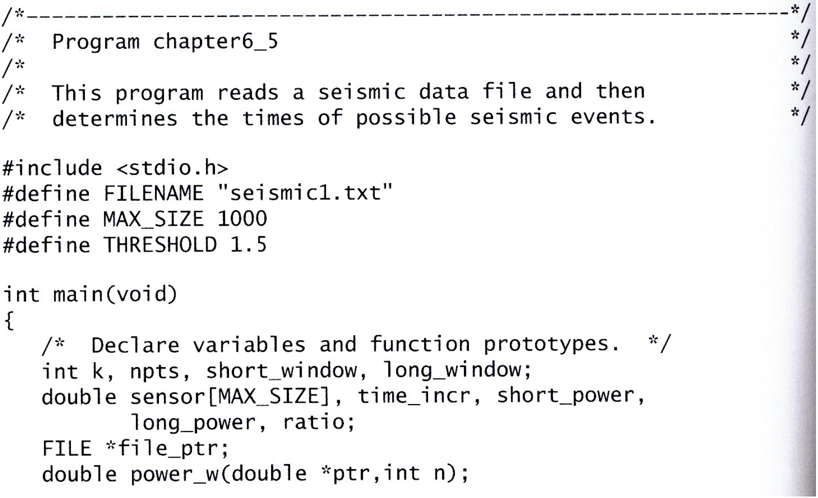
Point $x_{10}$: Short-time power = (1 + 1)/2 = 1,
Long-time power = (1 + 1 + 1 + 4 + 16)/5 = 4.6,
Ratio = 1/4.6 = 0.22.

# Example: Seismic Event Detection

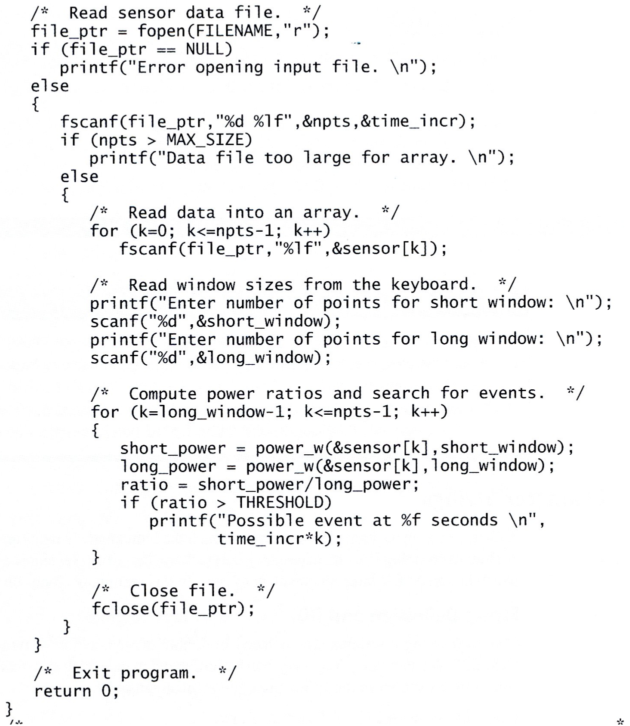


```
main:   set threshold to 1.5
        read npts and time-interval
        read the values into sensor array
        read short-window, long-window from keyboard
        set k to long-window - 1
        while k $\leq$ npts-1
            set short-power to power(sensor,short-window,k)
            set long-power to power(sensor,long-window,k)
            set ratio to short-power/long-power
            if ratio >threshold
                print $k \cdot$ time-interval
            increment k by 1
power(x,length,n):
    set xsquare to zero
    set k to n
    while k>n-length+1
        add $x[k] \cdot x[k]$ to xsquare
    return xsquare/length
```

# Example: Seismic Event Detection

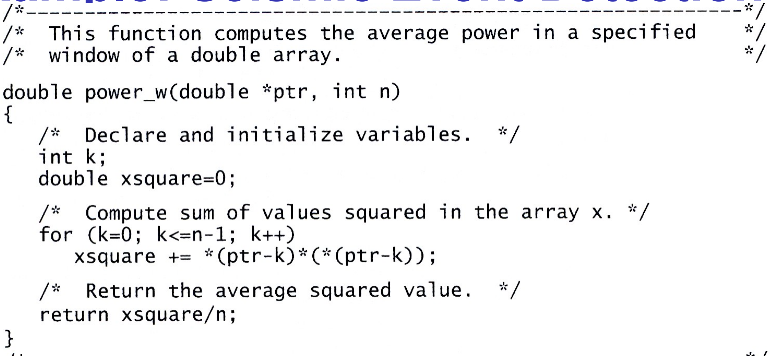


```
/*-------------------------------------------------------------*/
/*  Program chapter6_5                                         */
/*                                                             */
/*  This program reads a seismic data file and then            */
/*  determines the times of possible seismic events.           */

#include <stdio.h>
#define FILENAME "seismic1.txt"
#define MAX_SIZE 1000
#define THRESHOLD 1.5

int main(void)
{
   /*  Declare variables and function prototypes.  */
   int k, npts, short_window, long_window;
   double sensor[MAX_SIZE], time_incr, short_power,
          long_power, ratio;
   FILE *file_ptr;
   double power_w(double *ptr,int n);
```



```
   /*  Read sensor data file.  */
   file_ptr = fopen(FILENAME,"r");
   if (file_ptr == NULL)
      printf("Error opening input file. \n");
   else
   {
      fscanf(file_ptr,"%d %lf",&npts,&time_incr);
      if (npts > MAX_SIZE)
         printf("Data file too large for array. \n");
      else
      {
         /*  Read data into an array.  */
         for (k=0; k<=npts-1; k++)
            fscanf(file_ptr,"%lf",&sensor[k]);

         /*  Read window sizes from the keyboard.  */
         printf("Enter number of points for short window: \n");
         scanf("%d",&short_window);
         printf("Enter number of points for long window: \n");
         scanf("%d",&long_window);

         /*  Compute power ratios and search for events.  */
         for (k=long_window-1; k<=npts-1; k++)
         {
            short_power = power_w(&sensor[k],short_window);
            long_power = power_w(&sensor[k],long_window);
            ratio = short_power/long_power;
            if (ratio > THRESHOLD)
               printf("Possible event at %f seconds \n",
                      time_incr*k);
         }

         /*  Close file.  */
         fclose(file_ptr);
      }
   }

   /*  Exit program.  */
   return 0;
}
```

# Example: Seismic Event Detection

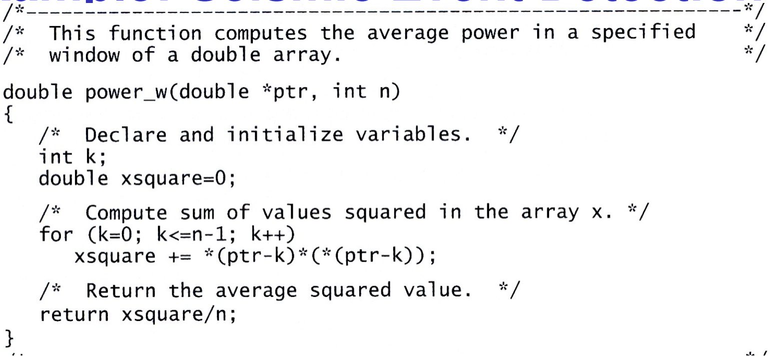


```
/*----------------------------------------------------------------*/
/*  This function computes the average power in a specified      */
/*  window of a double array.                                    */

double power_w(double *ptr, int n)
{
   /*  Declare and initialize variables.  */
   int k;
   double xsquare=0;

   /*  Compute sum of values squared in the array x. */
   for (k=0; k<=n-1; k++)
      xsquare += *(ptr-k)*(*(ptr-k));

   /*  Return the average squared value.  */
   return xsquare/n;
}
```

**Output**

```
Enter the number of points for short-window:
2
Enter the number of points for long-window:
5
Possible event at 0.050000 seconds
Possible event at 0.060000 seconds
```

# Example: Seismic Event Detection



```
/*-----------------------------------------------------------------*/
/*  This function computes the average power in a specified       */
/*  window of a double array.                                     */

double power_w(double *ptr, int n)
{
   /*  Declare and initialize variables.  */
   int k;
   double xsquare=0;

   /*  Compute sum of values squared in the array x. */
   for (k=0; k<=n-1; k++)
      xsquare += *(ptr-k)*(*(ptr-k));

   /*  Return the average squared value.  */
   return xsquare/n;
}
```

**Output**

```
Enter the number of points for short-window:
2
Enter the number of points for long-window:
5
Possible event at 0.050000 seconds
Possible event at 0.060000 seconds
```

# Summary on Pointers

A pointer is a variable whose value is an *address*.

A pointer is declared by specifying the *type* of object that it points to.

There is a special address called `NULL`.

NOT DISCUSSED: *void* pointers

Two main uses of pointers are to allow:

- two different variables to refer to *common* memory.
  A classic example is a variable in a main program which is to be modified by a formal parameter in a function.
- *dynamic data structures*.
  In conjunction with dynamic memory allocation (recall `malloc`), this allows the program to flexibly use amount the memory depending on its need, and to build complex relationsips between variables.
  (Not Examinable)

# Pointers and Memory

```
int *ptr; /* declaration of ptr as pointing to integer */
double *ptr2;
...
*ptr = 2; /* use of pointer, allocates 4 byes for integer 2 */
*ptr2 = 2; /* allocates 8 byes for double 2 */
```

NOTE: ensure that `ptr` and `ptr2` have *valid* addresses before assigning them.
How does one do this?

Some common errors (why?):

```
int i, *ptr1;
char *ptr2;

i = &ptr;
ptr1 = ptr2;
ptr2 = ptr1;
ptr1 = i;
*ptr2 = 'c';
```

# Pointers and Arrays

Pointers do not have to point to single variables. They can also point at the cells of an array.

```
int *ptr1, *ptr2;
int a[10];
ptr1 = &a[0]; /* ptr1 points to 1st int in array a */
ptr2 = &a[3]; /* ptr2 points to 4th int in array a */
```

NOTE: The expression `a` in the array declaration `int a[10];` is also a pointer (to an integer). However, the two declarations

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

are not the same. In the former case, `a` *cannot be changed*.

Similarly, `a` and `ptr` are interchangeable in usage, the declaration `int a[10];` provides *valid memory of 10 ints*, whereas the declaration `int *ptr1;` does not.

# Pointer Arithmetic

Valid expressions: where `ptr` and `ptr2` are pointers (to anything):

- Adding/Subtracting a constant offset: `ptr + ` *k* where *k* is an integer.
- Increment/Decrement: `ptr++`, `++ptr`, `ptr--`, `--ptr`
- Pointer subtraction: `ptr - ptr2`
- NOTE: there is no pointer addition

```
int a[5], b[5][10], *ap, *bp, i, j;
double d[5], *dp;
printf("a: ");    for (i = 0; i < 5; i++) printf("%p ", a + i);
printf("\nd: "); for (i = 0; i < 5; i++) printf("%p ", d + i);
printf("\nb: "); for (i = 0; i < 5; i++) printf("%p ", b + i);
i = (int) &a[0];
j = (int) &a[1];
printf("\ni = &a[0] = %p, j = &a[1] = %p\nj - i = %d, &a[1]-&a[0] = %d\n",
    i, j, j-i, &a[1]-&a[0]);
```

Output:

```
a:  0xbffa30 0xbffa34 0xbffa38 0xbffa3c 0xbffa40  (difference 4)
d:  0xbffa00 0xbffa08 0xbffa10 0xbffa18 0xbffa20  (difference 8)
b:  0xbff930 0xbff958 0xbff980 0xbff9a8 0xbff9d0  (difference 40)
i = &a[0] = 0x5fbffa30, j = &a[1] = 0x5fbffa34
j - i = 4, &a[1] - &a[0] = 1
```

NOTE: why are the last 2 numbers different?

# Arrays/Pointers as Function Arguments

Recall that a function call with an array parameter results in a passing of the *address* of the array, and not the array itself. Eg: suppose `getline(line, max)` reads at most a line of `max` chars into the array `line`.

```
int getline(int line[], int max); /* line[] is defined without a size */

int main(void) {
   char line[100], line2[200];
   getline(line, 100);
   getline(line2, 200);
}
```

Thus we can use the same function `getline` in order to get lines of input into two arrays of different sizes. This is because the call to `getline` is supplied not with an array, but with a *pointer* to an array. Further, it is known that the array contains integers.

Thus the two calls above are as if we had written

```
int getline(int *line, int max);

int main(void) {
   char line[100], line2[200];
   getline(&line[0], 100);
   getline(&line2[0], 200);
}
```

# getline

```
int getline(char *line, int max) {
    int nch = 0, c;
    max = max - 1; /* leave room for ’\0’ */
    while((c = getchar()) != EOF) {
        if(c == ’\n’) break;
        if(nch < max) *(line + nch++) = c;
    }
    if(c == EOF && nch == 0) return EOF;
    *(line + nch) = ’\0’;
    return nch;
}
```

# Arrays/Pointers as Function Arguments

Consider now two-dimensional arrays.

Could a function with a parameter array do without *both* the row and column sizes?

```
void setzero(int box[][], int max1, int max2);

int main(void) {
   char box[100][100], box2[200][200];
   setzero(box, 100, 100);
   setzero(box2, 200, 200);
}

void setzero(int box[][], int max1, int max2) {
   int i, j;
   for (i = 0; i < max1; i++)
      for (j = 0; j < max2; j++) box[i][j] = 0;
}
```

The answer is NO. The problem is how to compute the address of `box[i][j]`. Why?

In order to compute `box[i][j]` (or equivalently `*(box + i) + j`), we require

- The address of the first element of the array (`box`)
- The size of the type of the elements of the array (here, `sizeof(int)`)
- The 2nd dimension of the array (here, 100 or 200)
- The specific index value for the first dimension (here, `i`)
- The specific index value for the second dimension, (here, `j`).

# Arrays/Pointers as Function Arguments

A correct way:

```
void setzero1(int box[][100], int max);
void setzero2(int box[][200], int max);

int main(void) {
   char box[100][100], box2[200][200];
   setzero1(box, 100);
   setzero2(box2, 200);
}

void setzero1(int box[][100], int max) {
   int i, j;
   for (i = 0; i < max; i++)
      for (j = 0; j < 100; j++) box[i][j] = 0;
}

void setzero2(int box[][200], int max) {
   int i, j;
   for (i = 0; i < max; i++)
      for (j = 0; j < 200; j++) box[i][j] = 0;
}
```

# Additional Notes on Arrays

- Recall that

  `int a[100] = {6, 7, 8, 9}, *ptr = &a[1];`

  results in `a` being a pointer to a one-dimensional array.
  Thus `a[3]`, `*(a + 3)` and `*(ptr + 2)` are the same, equalling 9.

- A two-dimensional array eg: `int a[5][5];` is treated as a *one-dimensional array of a one-dimesional array*.
  - `a` refers to a two-dimensional array
  - `*a, *(a + 1), *(a + 2), ...` are pointers to a one-dimensional array.
    Recall that these are the same as `a[0], a[1], a[2], ...`
  - `**a, **(a + 1), **(a + 2)` are the first three elements of the first column of `a`.
  - `*(*(a + 4)), *(*(a + 4) + 1), *(*(a + 4) + 2)` are the first three elements of the fifth row of `a`.

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("Addresses of first three rows of a:\n");
printf("  *a=a[0]=%p, *(a+1)=a[1]=%p, *(a+2)=a[2]=%p\n",
    *a,*(a+1),*(a+2));
printf("Checking their differences:\n");
printf("  a[1]-a[0]=%d,a[2]-a[0]=%d\n", a[1]-a[0], a[2]-a[0]);
}
```

OUTPUT:

```
Addresses of first three rows of a:
  *a=a[0]=0x22ee50, *(a+1)=a[1]=0x22ee64, *(a+2)=a[2]=0x22ee78
Checking their differences:
  a[1]-a[0]=5, a[2]-a[0]=10
```

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("First three elements of first column of a:\n");
printf("  **a = %d, **(a+1) = %d, **(a+2) = %d\n",
**a, **(a+1), **(a+2));
printf("First three elements of fifth row of a:\n");
printf("  *(*(a+4)) = %d, *(*(a+4)+1) = %d, *(*(a+4)+2) = %d\n",
*(*(a+4)), *(*(a+4)+1), *(*(a+4)+2));
}
```

```
OUTPUT:

First three elements of first column of a:
  **a = 0, **(a+1) = 10, **(a+2) = 20
First three elements of fifth row of a:
  *(*(a+4)) = 40, *(*(a+4)+1) = 41, *(*(a+4)+2) = 42
```

```
main() {
int a[][5] = {{ 0,  1,  2,  3,  4},
              {10, 11, 12, 13, 14},
              {20, 21, 22, 23, 24},
              {30, 31, 32, 33, 34},
              {40, 41, 42, 43, 44}};
int *b[] = {a[0], a[1], a[2], a[3], a[4]};
int *p = a[0];
int **q = b;

printf("First three elements of b:\n");
printf("  *b = %p, *(b+1) = %p, *(b+2) = %p\n", *b,*(b+1),*(b+2));
q += 2;
printf("Last three elements of first column of a:\n");
printf("  **q = %d, **(q+1) = %d, **(q+2) = %d\n",
    **q, **(q+1), **(q+2));
}
```

```
OUTPUT:

First three elements of b:
  *b = 0x22ee50, *(b+1) = 0x22ee64, *(b+2) = 0x22ee78
Last three elements of first column of a:
  **q = 20, **(q+1) = 30, **(q+2) = 40
```

```
main() {
int a[][3] = {{ 0,  1,  2},
              {10, 11, 12},
              {20, 21, 22},
              {30, 31, 32}};
int *p = a[0];

printf("Addresses of a:\n   %p\n", a);
printf("Addresses of a[0] ... a[3]:\n    %p %p %p %p\n",
    &a[0], &a[1], &a[2], &a[3]);
printf("Contents  of a[0] ... a[3]:\n    %p %p %p %p\n",
    a[0], a[1], a[2], a[3]);

printf("Addresses of row 0: %p %p %p\n",
    &a[0][0], &a[0][1], &a[0][2]);
printf("Addresses of row 1: %p %p %p\n",
    &a[1][0], &a[1][1], &a[1][2]);
printf("Addresses of row 2: %p %p %p\n",
    &a[2][0], &a[2][1], &a[2][2]);
printf("Addresses of row 3: %p %p %p\n",
    &a[3][0], &a[3][1], &a[3][2]);

printf("Scan entire array:\n");
for (p = *a; *p != 32; printf("%d ", *p), p++) ;
printf("%d\n", *p);
}
```

```
OUTPUT:

Addresses of a:
   0x22ee90
Addresses of a[0] ...  a[3]:
   0x22ee90 0x22ee9c 0x22eea8 0x22eeb4 ...   WHY?!?
Contents of a[0] ...  a[3]:
   0x22ee90 0x22ee9c 0x22eea8 0x22eeb4

Addresses of row 0:  0x22ee90 0x22ee94 0x22ee98
Addresses of row 1:  0x22ee9c 0x22eea0 0x22eea4
Addresses of row 2:  0x22eea8 0x22eeac 0x22eeb0
Addresses of row 3:  0x22eeb4 0x22eeb8 0x22eebc

Scan entire array:
0 1 2 10 11 12 20 21 22 30 31 32
```

# References

Etter Sections 6.1 to 6.6

# Next Lecture

Strings and Structures