CS 201
Passing Function as Parameter & Array

Debzani Deb

Announcement

- Your TA will be out of town for next few weeks.
  - Labs will be taken by me and another grad student Shahriar Hossain.
  - Submit assignments to "Fuad" as before, we will be helping and grading in the class only.
  - If you need immediate attention, email me.
- I will be out of town on 2nd and 4th April. Classes will be taken by Shahriar.
- Lab 8 is removed. Instead Lab 9 will be counted as twice.
  - Don't expect linear distribution of lab grades.
  - Do not expect to have same kind of deadline for each lab.

Passing a Function Name as a Parameter

- In C it is possible to pass a function name as a parameter of another function.
- Gives the called function the ability to do something using different functions each time it is called.
- Declaring a function parameter is accomplished by simply including a prototype of the function in the parameter list.
- Let's look at a simple example similar to the evaluate example (Fig 7.7 in page 348) in the text.

Passing a function

```
#include <stdio.h>
#include <math.h>
double evaluate(double f( ), double);
int main (void)
{
        double sqrtvalue, sinvalue;
        sqrtvalue = evaluate(sqrt, 12.5);
        printf("%f \n", sqrtvalue);
        sinvalue = evaluate(sin, 0.5);
        printf("%f \n", sinvalue);
}
double evaluate ( double f(double f_arg), double pt1)
{
        return (f(pt1));
}
```


Passing a function

```
#include <stdio.h>
#include <math.h>
double evaluate(double f( ), double);
int main (void)
{
        double sqrtvalue, sinvalue;
        sqrtvalue = evaluate(sqrt, 12.5);
        printf("%f \n", sqrtvalue);          3.535534
        sinvalue = evaluate(sin, 0.5);
        printf("%f \n", sinvalue);           0.479426
}
double evaluate ( double f(double f_arg), double pt1)
{
        return (f(pt1));
}
```


Passing a function

```
#include <stdio.h>
#include <math.h>
double evaluate(double f( ), double);
int main (void)
{
        double sqrtvalue, sinvalue;
        sqrtvalue = evaluate(sqrt, 12.5);
        printf("%f \n", sqrtvalue);          3.535534
        sinvalue = evaluate(sin, 0.5);
        printf("%f \n", sinvalue);           0.479426
}
double evaluate ( double f(double f_arg), double pt1)
{
        return (f(pt1));
}
```

Passing a function

```
#include <stdio.h>
#include <math.h>
double evaluate(double f( ), double);
int main (void)
{
        double sqrtvalue, sinvalue;
        sqrtvalue = evaluate(sqrt, 12.5);
        printf("%f \n", sqrtvalue);
        sinvalue = evaluate(sin, 0.5);
        printf("%f \n", sinvalue);
}
double evaluate ( double f(double f_arg), double pt1)
{
        return (f(pt1));
}
```

3.535534

0.479426

Array

What is an Array?

- **Scalar** data types use a single memory cell to store a single value.
- For many problems you need to group data items together.
- A program that processes exam scores for a class, for example, would be easier to write if all the scores were stored in one area of memory and were able to be accessed as a group.
- C allows a programmer to group such related data items together into a single composite **data structure**.
- In this lecture, we look at one such data structure: the **Array**.

Array Terminology (1)

- An **array** is a collection of two or more adjacent memory cells that are:
  - The same type (i.e. int)
  - Referenced by the same name
- These individual cells are called **array elements**
- To set up an array in memory, we must declare both the name and type of the array and the number of cells associated with it

```
double x[8];
```

- This instructs C to associate eight memory cells with the name x; these memory cells will be adjacent to each other in memory.
- You can declare arrays along with regular variables

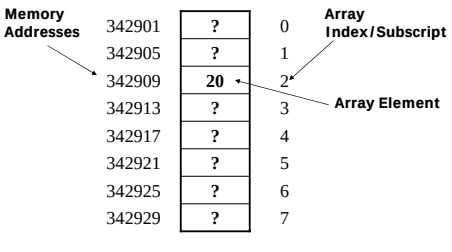
```
double cactus[5], needle, pins[7];
```


Array Terminology (2)

- Each **element** of the array x may contain a single value of type double, so a total of eight such numbers may be stored and referenced using the array name x.
- To process the data stored in an array, we reference each individual element by specifying the array name and identifying the element desired.
- **The elements are numbered starting with 0**
  - An array with 8 elements has elements at 0,1,2,3,4,5,6, and 7
- The subscripted variable **x[0]** (read as x sub zero) refers to the initial or **0th element** of the array x, x[1] is the next element in the array, and so on.
- The integer enclosed in brackets is the **array subscript** or **index** and its value must be in the range from zero to one less than the array size.

Visual representation of an Array

```
int x[8];  x[2] = 20;
```



Note: Index starts with 0, not with 1

Array Declaration - Syntax

<element-type> <array-name> [<array-size>]

- The number of elements, or array size must be specified in the declaration.
- Contiguous space in memory is allocated for the array.
- Related data items of the same type (i.e. contiguous memory cells are of the same size).
- Remain same size once created (i.e. they are "Fixed-length entries")

Array Initialization(1)

- When you declare a variable, its value isn't initialized unless you specify.

```
int sum;        // Does not initialize sum
int sum = 1;    // Initializes sum to 1
```

- Arrays, like variables, aren't initialized by default

```
int X[10];  //creates the array, but doesn't set any of its values.
```

- To initialize an array, list all of the initial values separated by commas and surrounded by curly braces:

```
int X[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

- The array elements are initialized in the order listed

```
X[0] == 2
X[4] == 11
```


Array Initialization(2)

- If there are values in the initialization block, but not enough to fill the array, all the elements in the array without values are initialized to 0 in the case of double or int, and NULL in the case of char.

```
int scores[20]  = {0};        // all 20 elements are initialized to 0
int scores[20]  = {1, 2, 3};  // First 3 elements are initialized to 1, 2,
                              // 3 and the rest are initialized to 0
```

- If there are values in the initialization block, an explicit size for the array does not need to be specified. Only an empty array element is sufficient, C will count the size of the array for you.

```
int scores[]  = {20, 10, 25, 30, 40}; // size of the array score is
                                      // automatically calculated as 5
```


Good Practice

```
const int maxAarraySize  = 12;
int myArray[maxArraySize];
```

OR

```
#define MAX_ARRAY_SIZE  12
int myArray[MAX_ARRAY_SIZE];
```


Array Subscripts

- We use subscripts/indices to differentiate between the individual array elements
- We can use any expression of type int as an array subscript.
- However, to create a valid reference, the value of this subscript must lie between 0 and one less than the array size.
- It is essential that we understand the distinction between an array subscript value and an array element value.

```
int x[2]; int y = 1; x[y] = 5;
```

The **subscript** is y (which is 1 in this case), and the **array element value** is 5

- C compiler does not provide any array bound checking. As a programmer it is your job to make sure that every reference is valid (falls within the boundary of the array).

Access (1)

1. point[1] // the 2nd element of array point is accessed
2. point[9] = 20; // the 10th element of array point is assigned
   // the value 20
3. Want to process all of the elements of an array?

   Example: Adding the values of all array elements

   Two alternative style for loops

```
for ( i = 0; i < arraySize; i++)  sum += a[i];
for ( i = 0; i <= arraySize-1; i++)  sum += a[i];
```

- Note : The array element is a single valued variable of the corresponding type and can be manipulated as a variable of that type.

Access (2)

```
int x[5];       // declare an integer array of size 5
int i = 2;

x[0] = 20;     // valid
x[2.3] = 5;    // Invalid, index is not int
x[6] = 10;     // valid, but dangerous
x[2*i – 3] = 3; // valid, assign 3 to x[1]
x[i++];        // access x[2] and then assign 3 to i
x[(int) x[1]]; // access x[3]
```

- Referencing to an array element outside of the created bounds is possible but rather not recommended.

Arrays and Pointers in C (1)

- Arrays can also be accessed with pointers in C.
- Pointers do not have to point to single/scalar variables. They can also point at individual array elements.

```
int * intptr;
int arr[10];
intptr = &arr[2];
```

- Pointers can be manipulated by "+" and "-":
- Adding 1 to a pointer is the same as adding the size of the type it was declared to be pointing at.

  The pointer intptr-1 points to arr[1] and intptr+2 points to arr[4]. Pointer intptr++ points at arr[3].

Arrays and Pointers in C (2)

- The name of the array is the address of the first element of the array.
- In other words – a name of the array is actually a pointer to the element of the array that has a subscript equal to 0.

```
int * intptr;
int arr[10];
intptr = arr; // same as intptr = &arr[0]
```

- Note: the name of the array ("arr') is a constant. We can't force this pointer to point at something else.

  arr+1 must be the same as &arr[1]

  arr+2 must be the same as &arr[2]

Arrays and Pointers in C (3)

10	0
20	1
30	2
40	3
50	4
60	5
70	6
80	7

```
int x[8], *aptr;
aptr = x;

printf("%d\n", x[5]);
printf("%d\n", *(x+5));
printf("%d\n", aptr[5]);
printf("%d\n", *(aptr+5));
```

The output is 60 in every case

Algorithm for Searching an Array

1. Assume target has not been found
2. Start with the intial array element
3. Repeat while the target is not found and there are more
   4. if the current element matches array
      5. set flag true
      6. remember array index
   7. else
      8. advance to next array element
9. If flag set true
   10. return the array index
11. else
    12. return -1 to indicate not found


```
int found = 0, i = 0;
int arr[10];

while ( !found && (i <10) ) {
      if ( arr[i] == target ) {
            found = 1;
            index = i;
      }
      else
            i++;
}

if ( found )
      return index;
else
      return -1;
```