Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2018/2019

Arrays and Parameter passing

Dr. Amin Allam

1 Arrays

Suppose we need to declare and define several variables, without assigning a specific identifier to each variable. For example, suppose a business man is interested in sales of 7 devices during the last year, which are: phones, cameras, keyboards, monitors, printers, scanners, and routers. He could do something like that:

```
int b, c, d, e, f, g, h; // A variable for sales of each device during the last year
b=5; c=7; d=3;           // He sold 5 phones, 7 cameras, 3 keyboards,
e=4; f=6; g=2; h=8;      //  4 monitors, 6 printers, 2 scanners, and 8 routers
```

where `b` represents the number of sold phones, `c` represents the number of sold cameras, and so on. These variables will be allocated in memory as follows (the dots mean that variables may not be contiguous in memory):

	b		c		d		e		f		g		h	
	5	...	7	...	3	...	4	...	6	...	2	...	8	

Instead of defining a separate variable for each device sales, we will define a group of variables (we call it an array). Each variable identifier consists of two components:

1. *Array name* which is the same for all variables in this group. The array name must correspond to the rules of identifiers (a sequence of english letters, digits and underscores which does not start by a digit).
2. *Variable index*, inside square brackets, which is different for each variable in this group. The variable index ranges from `0` to `n-1`, where `n` is the number of variables.

In the previous example, we could define an array of integer variables as follows:

```
int a[7];   // An array of 7 variables, each one for the sales of a specific device
a[0]=5; a[1]=7; a[2]=3; // He sold 5 phones, 7 cameras, 3 keyboards,
a[3]=4; a[4]=6; a[5]=2; a[6]=8; // 4 monitors, 6 printers, 2 scanners, 8 routers
```

The variables will share the *array name* `a`. In this example the *index* ranges from `0` to `6` since the number of variables is `7`. The variables will be *contiguously* allocated in memory as follows:

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	
	5	7	3	4	6	2	8	

Note that when defining an array, the array size must be constant. For example, the following codes are correct:

```
int a[7];       ✔
```

```
#define N 7
int a[N];       ✔
```

However, the following code is not consistent with the C/C++ standard:

```
int n=7;
int a[n];       ✖
```

The index inside the square brackets can be either constant or variable. In the previous examples, we used constant index. The following example uses a variable index `i` to set array values.

```
int a[7]; int i=0;
a[i]=5; i++; a[i]=7; ++i; a[i]=3; i++;
a[i++]=4; a[i]=6; a[++i]=2; a[i+1]=8;
```

An array can be initialized as follows:

```
int a[7]={5,7,3,4,6,2,8};
```

When initializing an array, it is possible to use empty brackets. The compiler will count the initialized values and allocates enough space for the array, as follows:

```
int a[]={5,7,4};     // The same as:  int a[3]={5,7,4};
```

However, it is not possible to define array using empty brackets without initialization:

```
int a[];       ✖
```

In general, if any variable is global or static, its value is set to `0`. Otherwise, it is not initialized (contains garbage values). The same rule applies to uninitialized arrays such as:

```
int a[3];
```

However, initialized arrays have a different rule: If the array is initialized to a number of values less than its size, the remaining values will be set to `0` as follows:

```
int a[7]={5,7,3};
```

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	
	5	7	3	0	0	0	0	

In all previous examples, we used an array of `int`. It is possible to use other data types such as `float`, `double`, `char`, `short`, and so on.

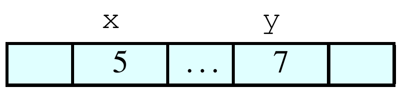
```
float y[4]={0};    // Construct 4 float variables, each variable is initialized to 0
double z[3]={1.2, 5.3};    // Construct 3 double variables, initialized to 1.2, 5.3, 0
```

2 Parameter passing

The main types of parameter passing are by value and by reference. Passing by value does not affect the value of actual parameters. The following example illustrates passing by value:

```
void Fun(int y)  // Initially y=5 which is the value of x when calling Fun(x)
{
   y=7;                 // y value becomes 7, while the value of x does not change
}
int main()
{
   int x=5;
   Fun(x);
   cout << x << endl;   // Prints 5 on screen, because x value did not change
   return 0;
}
```

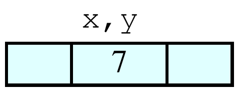
In the previous example, we call `x` the *actual parameter*, while we call `y` the *formal parameter*. The memory contains `x` and `y` as two separate variables:



Passing by reference may affect the value of actual parameters. Passing a parameter by reference is indicated by putting `&` after the type of the formal parameter. The following example illustrates passing by reference:

```
void Fun(int& y)  // Initially y=5 which is the value of x when calling Fun(x)
{
   y=7;                 // y value becomes 7, x value becomes 7 as well
}
int main()
{
   int x=5;
   Fun(x);
   cout << x << endl;   // Prints 7 on screen, because x value changed
   return 0;
}
```

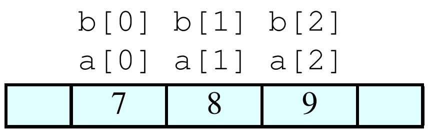
Note that `x` and `y` must have the same type. The memory contains `x` and `y` as one variable:



To understand the parameter passing type of arrays, we need more information that we will know after studying pointers in the coming lectures. For now, we can indicate that the effect on the array variables is similar to passing by reference as follows:

```
void Fun(int b[3])
{
   b[0]=7; b[1]=8; b[2]=9;
}
int main()
{
   int a[3]={1,2,3};
   Fun(a);
   cout<<a[0]<<" "<<a[1]<<" "<<a[2]<<endl; // Prints 7 8 9 on screen
   return 0;
}
```

The memory contains `a[3]` and `b[3]` as one array:



When passing an array as a parameter, it is possible to use empty brackets for the formal parameters, such as follows:

```
void Fun(int b[])
{
   b[0]=7; b[1]=8; b[2]=9;
}
```

Note that a function can take several parameters with similar or different types of parameter passing in any order, as follows:

```
void Fun(float y, int b[3], short& z)
{
   z=7;
   y=7;
   b[0]=7; b[1]=8; b[2]=9;
}
int main()
{
   int a[3]={1,2,3};
   float x=5.2;
   short t=5;
   Fun(x, a, t);
   cout << x << " " << t << endl;    // Prints 5.2 7 on screen
   cout<<a[0]<<" "<<a[1]<<" "<<a[2]<<endl; // Prints 7 8 9 on screen
   return 0;
}
```

3 Examples

1. Write a function that takes an array of integers and its size as parameters, and returns the maximum integer value in this array.

```
int GetMax(int a[], int n)
{
   int i, m=a[0];      // Initialize the current maximum value to the first value
   for(i=1;i<n;i++)    // Iterate over all values after the first value
   {
      if(a[i]>m)       // If the current value is larger than the
         m=a[i];       //   current maximum, update the current maximum
   }
   return m;
}
int main()
{
   int arr[7]={4,6,3,8,5,9,1};
   cout << GetMax(arr, 7) << endl;   // Prints 9 on screen
   return 0;
}
```

2. Write a function that takes an array of integers and its size as parameters, and returns the *index* of the maximum integer value in this array.

```
int GetMaxInd(int a[], int n)
{
   int i, index=0;     // Initialize the current maximum value to the first value
   for(i=1;i<n;i++)    // Iterate over all values after the first value
   {
      if(a[i]>a[index])   // If the current value is larger than
         index=i;         //  the current maximum, update index
   }
   return index;
}
int main()
{
   int arr[7]={4,6,3,8,5,9,1};
   int ind=GetMaxInd(arr, 7);
   cout << "The maximum value is " << arr[ind]
        << " located at index " << ind <<endl;
   return 0;       // The maximum value is 9 located at index 5
}
```