Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department



# 1 Enumerations

In a previous example, we constructed this array:

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

We used the index `0` for phone, `1` for camera and so on. Alternatively, we could use constant integers in order to avoid memorizing which index integer represents which device, as follows:

```
1 const int PHONE=0; const int CAMERA=1; const int KEYBOARD=2;
2 const int MONITOR=3; const int PRINTER=4; const int SCANNER=5;
3 const int ROUTER=6;
4 int a[7];   // An array of 7 variables, each one for the sales of a specific device
5 a[PHONE]=5; a[CAMERA]=7; a[KEYBOARD]=3; // He sold 5 phones ... etc
6 a[MONITOR]=4; a[PRINTER]=6; a[SCANNER]=2; a[ROUTER]=8;
```

An equivalent code to the previous example uses *unnamed enumerations* as follows (note that it is possible to use capital or small letters for constants, however we use capital letters to differentiate between constants and variables just to improve readability):

```
1 enum{PHONE,CAMERA,KEYBOARD,MONITOR,PRINTER,SCANNER,ROUTER};
2 // By default PHONE=0, CAMERA=1 ... etc
3 int a[7];   // An array of 7 variables, each one for the sales of a specific device
4 a[PHONE]=5; a[CAMERA]=7; a[KEYBOARD]=3; // He sold 5 phones, etc
5 a[MONITOR]=4; a[PRINTER]=6; a[SCANNER]=2; a[ROUTER]=8;
```

Enumerations can be named. *Named enumerations* are *user-defined data types* from which we can instantiate new objects. Each object can hold any of the enumerated values.

```
1 enum Device{PHONE,CAMERA,KEYBOARD,MONITOR,
2             PRINTER,SCANNER,ROUTER};
3 Device d, f, b[2];    // very similar to: int d, f, b[2];
4 d=1;                  // CAMERA
5 f=KEYBOARD;           // 2
6 b[0]=6;               // ROUTER
7 b[1]=SCANNER;         // 5
8 cout<<f<<endl;        // Prints 2
```

Although the `Device` data type is actually very similar to `int`, the readability of the code is much improved than just declaring `d` and `f` as `int`. It is also possible to construct array of `Device`, such as `b` in the previous example. Also, the `Device` data type can be used in similar ways to all built-in data types which we studied previously, such as `int`.

Usually we construct an array of strings that textually represents the enumerated values, to be used for output as follows:

```
enum Device{PHONE,CAMERA,KEYBOARD,MONITOR,
            PRINTER,SCANNER,ROUTER};
const char* device_str[]={"Phone", "Camera", "Keyboard",
                "Monitor", "Printer", "Scanner", "Router"};
Device d=PRINTER;
cout<<d<<" "<<device_str[d]<<endl;   // Prints 4 Printer
```

Enumerated values start from 0 by default. It is also possible to assign different values. The uninitialized values will take the previous value plus one.

```
enum{PHONE,CAMERA,KEYBOARD=6,MONITOR,PRINTER=9,SCANNER,ROUTER};
//  PHONE=0, CAMERA=1, MONITOR=7, SCANNER=10, ROUTER=11
```

## 2 Structures

In the previous section we defined `Device` which is a user-defined data type which can hold an enumerated value. In this section we study a more general way to create a user-defined data type.

A *structure* is a user-defined data type which is composed from other data types. A structure can be composed from built-in data types, user-defined data types, or a combination from them as follows:

```
#define MAX_NAME 14
enum Gender{MALE,FEMALE};
struct Date{int year; int month; int day;};

struct Student
{
   int    id;
   char   first_name[MAX_NAME+1];
   char   last_name[MAX_NAME+1];
   Date   birth_date;
   Gender gender;
   double gpa;
};
```

In the previous example, we define one enumeration `Gender` and two structures, `Date` structure which contains 3 `int`s, and `Student` structure which contains one `int`, two C-strings, one `Date` object, one `Gender` object, and one `double`.

An object from a user-defined data type can be treated in the same way we treat an object from a built-in data type. We use the dot operator (`.`) to access member variable of an object from a struct as follows:

```
Student s;
s.id=20190999;
strcpy(s.first_name, "Aly");
strcpy(s.last_name, "Tamer");
s.birth_date.year=1990;
s.birth_date.month=8;
s.birth_date.day=26;
s.gender=MALE;
s.gpa=3.4;
```

An arrow operator (`->`) can be used to simplify a sequence of contents-of and dot operators:

```
Student* p=new Student;
p->id=20190999;
strcpy((*p).first_name, "Aly");
strcpy(p->last_name, "Tamer");
p->birth_date.year=1990;
(*p).birth_date.month=8;
p->birth_date.day=26;
p->gender=MALE;
(*p).gpa=3.4;
delete p;
```

It is also possible to create a static array of `Student` objects as follows:

```
Student a[10];
a[0].id=20190999;
strcpy(a[3].first_name, "Aly");
a[4].birth_date.year=1990;
a[2].gender=MALE;
a[5].gpa=3.2;
// ... etc
```

It is also possible to create a dynamic array of `Student` objects as follows:

```
Student* a=new Student[10];
a[0].id=20190999;
strcpy(a[3].first_name, "Aly");
a[4].birth_date.year=1990;
a[2].gender=MALE;
a[5].gpa=3.2;
// ... etc
delete[] a;
```

# 3 Sorting

There are several ways to sort an array of elements. We will study two algorithms for sorting: selection sort and insertion sort. In other courses, we will study more efficient ways for sorting.

## 3.1 Selection sort

Selection sort starts by finding the minimum element and swapping it with first position. Then, it finds the minimum element among all elements starting from second position, and swaps it with second position, and so on.

```
1  // Swap values of a and b
2  void swap(int& a, int& b)
3  {
4      int t=a; a=b; b=t;
5  }
6
7  // Get the index of the minimum value of all values
8  //   among a[j ]... a[n-1] where n is the array size
9  int GetMinInd(int* a, int j, int n)
10 {
11     int i, ind=j;
12     for(i=j+1;i<n;i++)
13         if(a[i]<a[ind])
14             ind=i;
15     return ind;
16 }
17
18 void SelectionSort(int* a, int n)
19 {
20     int j;
21     for(j=0;j<n-1;j++)
22     {
23         // Get the index of the minimum value among a[j ]... a[n-1]
24         int ind=GetMinInd(a, j, n);
25
26         // Swap a[j] and the minimum value among a[j ]... a[n-1]
27         swap(a[ind], a[j]);
28
29         // Now, the values a [0]... a[j] are sorted
30         // But the values a[j +1]... a[n-1] are not sorted
31     }
32 }
```

### 3.2 Insertion sort

For each array position `j` in order, insertion sort moves the element at position `j` to left as far as required such that `a[0]...a[j]` are sorted.

```
1  // Starting with a [0]... a[j-1] are sorted,
2  // Move the element a[j] to the left as far such that a [0]... a[j] are sorted
3  void MoveLeft(int* a, int j)
4  {
5     int i, t=a[j];
6     for(i=j-1;i>=0;i--)
7     {
8        if(t<a[i]) a[i+1]=a[i];
9        else break;
10    }
11    a[i+1]=t;
12 }
13
14 void InsertionSort(int* a, int n)
15 {
16    int j;   // Initially, assume that a[0] is a small sorted array of one element
17    for(j=1;j<n;j++)
18       MoveLeft(a, j);   // after this step, a[0]...a[j] are sorted
19 }
```

## 4 Pointers to functions

In order to apply insertion sort on an array of students, we need to define the less than operator < in order to be able to compare two students, and then to sort them. We can do this by defining a function like this to compare students according to id.

```
1  // Return true if a.id < b.id
2  bool LessThan(Student* a, Student* b){return a->id<b->id;}
3  void SelectionSort(Student* a, int n)
4  {
5     for(int j=0;j<n;j++)
6     {
7        int ind=j;
8        for(int i=j+1;i<n;i++)
9           if(LessThan(&a[i], &a[ind]))
10             ind=i;
11       Student t=a[ind]; a[ind]=a[j]; a[j]=t;
12    }
13 }
```

Now suppose we need to sort students according to different criteria, such as name or gpa. We can rewrite the previous function with a different name and use another function than `LessThan()` inside it. However, this approach requires duplicating a lot of sorting code which is not good in software engineering. Another approach is to write one sorting procedure which accepts a function pointer as parameter, as follows:

```
1  void SelectionSort(Student* a, int n,
2                     bool (*LessThan)(Student*,Student*))
3  {
4     for(int j=0;j<n;j++)
5     {
6        int ind=j;
7        for(int i=j+1;i<n;i++)
8           if(LessThan(&a[i], &a[ind]))
9              ind=i;
10       Student t=a[ind]; a[ind]=a[j]; a[j]=t;
11    }
12 }
13
14 bool IdLess(Student* a, Student* b){return a->id<b->id;}
15 bool GpaGreater(Student* a, Student* b){return a->gpa>b->gpa;}
16 bool NameLess(Student* a, Student* b)
17 {
18    if(strcmp(a->first_name,b->first_name)!=0)
19       return strcmp(a->first_name,b->first_name)<0;
20    return strcmp(a->last_name,b->last_name)<0;
21 }
22
23 int main()
24 {
25    int n; cin>>n;
26    Student* s=new Student[n];
27    // Fill student array with data
28    SelectionSort(s, n, IdLess); // Sort according to increasing id
29    SelectionSort(s, n, GpaGreater); // Sort according to decreasing gpa
30    SelectionSort(s, n, NameLess); // Sort according to name
31    delete[] s;
32    return 0;
33 }
```