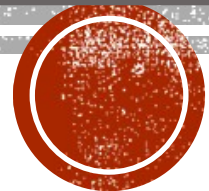


**Bule Hora University
College of Informatics
Department of Computer Science**

: Fundamental Programming II

CHAPTER 1: ARRAYS, STRINGS AND STRUCTURES



What Is Array?

- ❑ A collection of identical data objects, which are stored in consecutive memory
- ❑ locations under a common Variable name.
- ❑ In other words, an array is a group or a table of values referred to by the same name.
- ❑ The individual values in array are called elements. Array elements are also variables.
- ❑ Set of values of the same type, which have a single name followed by an index.
- ❑ In C++, square brackets appear around the index right after the name
- ❑ A block of memory representing a collection of many simple data variables stored in a separate array element,
- ❑ and the computer stores all the elements of an array consecutively in memory

Properties Of Array

- ❑ Arrays in C++ are zero-bounded; that is the index of the first element in the array is 0
- ❑ the last element is $N-1$, where N is the size of the array.
- ❑ It is illegal to refer to an element outside of the array bounds, and your program
- ❑ will crash or have unexpected results, depending on the compiler.
- ❑ Array can only hold values of one type

Array Declaration

- ❑ Declaring the name and type of an array and setting the number of elements in an array is called **dimensioning the array**.
- ❑ The array must be declared before one uses it like other variables. In the array declaration one must define:
 1. The type of the array (i.e. integer, floating point, char etc.)
 2. Name of the array,
 3. The total number of memory locations to be allocated or the maximum value of each subscript. i.e. the number of elements in the array.
- ❑ So the general syntax for the declaration is:

`DataTypename arrayname [array size];`

Cont.

- ❑ **An** array size cannot be a variable whose value is set while the program is running.
- ❑ Thus to declare an integer with size of 10 having a name of num is:

```
int num [10];
```

- ❑ This means: ten consecutive two byte memory location will be reserved with the name num.

Initializing Arrays

- ❑ When declaring an array of local scope (within a function), if we do not specify
the array variable will not be initialized.
- ❑ so its content is undetermined until we store some values in it.
- ❑ If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

```
int day [5];
```

Cont.

- Every element of day will be set initially to **0**:

Index []	[0]	[1]	[2]	[3]	[4]
day	0	0	0	0	0

- But additionally, when we declare an array, we have the possibility to assign

initial values to each one of its elements using curly brackets { }. For example:

```
int day [5] = { 16, 2, 77, 40, 12071 };
```

Index[]	[0]	[1]	[2]	[3]	[4]
day	16	2	77	40	12071

- The number of elements in the array that we initialized within curly brackets { } must be equal or less than the length in elements that we declared for the array enclosed within square brackets [].
- If we have less number of items for the initialization, the rest will be filled with zero.

Cont.

- You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. I.e.

```
int arr [] = {16, 2, 77, 40, 12071};
```

```
int ar [4];
```

```
ar[]={1,2,3,4};//not allowed
```

```
arr=ar;//not allowed
```

- Note: when initializing an array, we can provide fewer values than the array elements. E.g. `int a [10] = {10, 2, 3};` in this case the compiler sets the remaining elements to zero.

Accessing And Processing Array Elements

- The ***first element*** in an array always has the ***index 0***, and if the array has ***n*** elements the ***last*** element will have the ***index n-1***.
- An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets.
- Thus to set the 15th element of the array above to 1.5 the following assignment is used:

□ `annual_temp[14] = 1.5;` Note that since the first element is at index 0, then the ***ith*** element is at ***index i-1***.
- Hence in the above the 15th element has index 14.

Cont.

□ We must take care of not confusing these two possible uses of brackets []

with arrays:

Eg: `int day[5];` // *declaration of a new Array*
(begins with a type name)

`day[2] = 75;` // *access to an element of the Array.*

□ Other valid operations with arrays in accessing and assigning:

`int a=1;`

`day [0] = a;`

`day[a] = 5;`

`b = day [a+2];`

`day [day[a]] = day [2] + 5;`

`day [day[a]] = day[2] + 5;`

EX: *Arrays example ,display the sum of the numbers in the array*

```
#include <iostream.h>
int day [ ] = {16, 2, 77, 40, 12071};
int n, result=0;
void main ()
{
for ( n=0 ; n<5 ; n++ )
{
result += day[n];
}
cout << result;
getch();
}
```

- ❑ At some moment we may need to pass an array to a function as a parameter.
- ❑ In C++ it is not possible to pass by value a complete block of memory as a Parameter.
- ❑ But it is allowed to pass its address, which has almost the same practical effect and is a much faster and more efficient operation.
- ❑ In order to admit arrays as parameters the only thing that we must do is declaring the function by specifying in the argument.
- ❑ the base **type** for the array that it contains, an identifier and a pair of void brackets **[]** .
- ❑ For example, the following function: void procedure (int arg[]).

- Admits a parameter of type "Array of **int** " called **arg**. In order to pass to this function an array declared as:

```
int myarray [40];
```

- It would be enough with a call like this:

```
procedure (myarray);
```

■ Here you have a complete example:

```
1. // arrays as parameters
2. #include <iostream.h>
3. #include <conio.h>
4. void printarray (int arg[], int length)
5. {
6.     for (int n=0; n<length; n++)
7.         cout << arg[n] << " ";
8.     cout << "\n";
9. }
10. void mult(int arg[], int length)
11. {
12.     for (int n=0; n<length; n++)
13.         arg[n]=2*arg[n];
14. }
```

```
1. void main ()  
2. {  
3. int firstarray[] = {5, 10, 15};  
4. int secondarray[] = {2, 4, 6, 8, 10};  
5. printarray (firstarray,3);  
6. printarray (secondarray,5);  
7. mult(firstarray,3);  
8. cout<<"first array after being doubled is\n";  
9. printarray (firstarray,3);  
10.getch()  
11.}
```

- ❑ In all programs and concepts we have seen so far, we have used only numerical variables, used to express numbers exclusively.
- ❑ But in addition to numerical variables there also exist strings of characters that allow us to represent successive characters, like words, sentences, names, texts, etc.
- ❑ In C++ there is no specific *elementary* variable type to store string of characters. In order to fulfill this feature we can use arrays of type ***char***,
- ❑ Remember that this data type (***char***) is the one used to store a single character,
- ❑ for that reason arrays of them are generally used to make strings of single characters.

- For example, the following array (or string of characters) can store a string up to 20 characters long. You may imagine it thus:



- ✓ String is nothing **but a sequence of character** in which the last character is the **null character '\0'**.
- ✓ The null character indicates the **end of the string**.

- For example a string variable s1 could be declared as follows:

```
char s1[10];
```

- The string variable s1 could hold strings of length up to nine characters since space is needed for the final null character.
- Strings can be initialized at the time of declaration just as other variables are initialized. For example:

```
char s1[] = "example";
```

```
char s2[20] = "another example"
```

- ❑ At some moment we may need to pass an array to a function as a parameter.
- ❑ In C++ it is not possible to pass by value a complete block of memory as a Parameter.
- ❑ But it is allowed to pass its address, which has almost the same practical effect and is a much faster and more efficient operation.
- ❑ In order to admit arrays as parameters the only thing that we must do is declaring the function by specifying in the argument.
- ❑ the base **type** for the array that it contains, an identifier and a pair of void brackets **[]** .
- ❑ For example, the following function: void procedure (int arg[]).

Strings representation

- ✓ For example a string variable s1 could be declared as follows:

```
char s1[10];
```

- ✓ The string variable s1 could hold strings of length up to nine characters since space is needed for the final null character.
- ✓ Strings can be initialized at the time of declaration just as other variables are initialized. For example:

```
char s1[] = "example";
```

```
char s2[20] = "another example"
```



Strings representation

✓ would store the two strings as follows:

s1 |e|x|a|m|p|l|e|\0|

s2 |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?|

✓ Note that the length of a string does not include the terminating null character.



String output

- ✓ A string is output by sending it to an output stream, for example:

```
cout << "The string s1 is " << s1 <<  
endl;
```

- ✓ would print

The string s1 is example



String Input

- ✓ When the input stream **cin** is used space characters, **newline** etc. are used as separators and terminators.
- ✓ Thus when inputting numeric data **cin** skips over any leading spaces and terminates reading a value when it finds a white-space character (**space, tab, newline** etc.).
- ✓ This same system is used for the input of strings, hence a string to be input cannot start with leading spaces, also if it has a space character in the middle then input will be terminated on that space character.



String Input

- ✓ To read a string with several words in it using **cin** we have to call **cin** once for each word.
- ✓ For example to read in a name in the form of a First_name followed by a Last_name we might use code as follows:

```
char firstname [12], lastname[12];
```

```
cout << "Enter name ";
```

```
cin >> firstname ;
```

```
cin >> lastname;
```

```
cout << "The name entered was "
```

```
  << firstname << " "
```

```
  << lastname;
```



String Input

✓ To read text containing blanks we use another function, **cin.get()**.

```
#include<iostream.h>
```

```
void main()
```

```
{
```

```
    const int max=80;
```

```
    char str[max];
```

```
    cout<<"\n Enter a string;"
```

```
    cin.get(str , max); // max avoid buffer overflow
```

```
    cout<<"\n You entered : "<<str;
```

```
}
```



String Input

Reading multiple lines

- ✓ We have solved the problem of reading strings with embedded blanks, but what about strings with multiple lines?
- ✓ It turns out that the **cin.get()** function can **take a third argument** to help out in this situation. This argument **specifies the character that tells the function to stop reading.**
- ✓ The default value of this argument is the **newline('\n')** character, but if you call the function with some other character for this argument, the default will be **overridden** by the specified character.



String Input

- ✓ In this example, we call the function with a dollar sign ('\$') as the third argument

//reads multiple lines, terminates on '\$' character

```
#include<iostream.h>

void main(){

    const int max=80;

    char str[max];

    cout<<"\n Enter a string:\n";

    cin.get(str, max, '$');      //terminates with $

    cout<<\n You entered:\n"<<str; }
```



Strings Manipulation

String constants

- ✓ You can initialize a string to a constant value when you define it. Here's an example'

```
#include<iostream.h>  
  
void main(){  
  
char str[] = "Welcome to C++ programming  
language";  
  
cout<<str;  
  
}
```



Strings Manipulation

Copying string the hard way

- ✓ The best way to understand the true nature of strings is to deal with them character by character

```
#include<iostream.h>
#include<string.h> //for strlen()
int main(){
    const int max=80;
    char str1[]=" my Captain! our fearful trip is done";
    char str2[max];
    for(int i=0; i<strlen(str1);i++)
        str2[i]=str1[i];
    str2[i]='\0';
    cout<<endl;
    cout<<str2;
}
```



Strings Manipulation

Copying string the easy way

- ✓ You can copy strings using **strcpy** or **strncpy** function.
- ✓ We assign strings by using the string copy function **strcpy**.
- ✓ The prototype for this function is in **string.h**.

strcpy(destination, source);

- ✓ **strcpy** copies characters from the location specified by **source** to the location specified by **destination**.
- ✓ It stops copying characters after it copies the terminating null character.
- ✓ The return value is the value of the destination parameter.



Strings Manipulation

Copying string the easy way

- ✓ You must make sure that the destination string is large enough to hold all of the characters in the source string (including the terminating null character). *Example:*

```
#include <iostream.h>
#include <string.h>

void main(){
    char me[20] = "David";
    cout << me << endl;
    strcpy(me, "YouAreNotMe");
    cout << me << endl ;
    return;
}
```



Strings Manipulation

- ✓ There is also another function **strncpy**, is like **strcpy**, except that it copies only a specified number of characters.

strncpy(destination, source, int n);

- ✓ It may not copy the terminating null character.



Strings Manipulation

It may not copy the terminating null character.

Example

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[] = "String test";
    char str2[] = "Hello";
    char one[10];
    strncpy(one, str1, 9);
    one[9] = '\0';
    cout << one << endl;
    strncpy(one, str2, 2);
    cout << one << endl;
    strcpy(one, str2);
    cout << one << endl;
}
```



Strings Manipulation

Concatenating strings

- ✓ In C++ the `+` operator cannot normally be used to concatenate string, as it can in some languages such as BASIC; that is you can't say: **`str3 = str1 + str2;`**
- ✓ You can use **`strcat()`** or **`strncat`**
- ✓ The function `strcat` concatenates (appends) one string to the end of another string.

`strcat(destination, source);`



Strings Manipulation

✓ *Example:*

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strcat(str1, "def");
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
}
```



Strings Manipulation

- ✓ The function `strncat` is like `strcat` except that it copies only a specified number of characters.

`strncat(destination, source, int n);`

- ✓ It may not copy the terminating null character.



Strings Manipulation

Example:

```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[30];
    strcpy(str1, "abc");
    cout << str1 << endl;
    strncat(str1, "def", 2);
    str1[5] = '\0';
    cout << str1 << endl;
    char str2[] = "xyz";
    strcat(str1, str2);
    cout << str1 << endl;
    str1[4] = '\0';
    cout << str1 << endl;
}
```



Strings Manipulation

Comparing strings

- ✓ Strings can be compared using **strcmp** or **strncmp** functions
- ✓ The function **strcmp** compares two strings.

strcmp(str1, str2);

strcmp returns: < 0 if str1 is less than str2

= 0 if str1 is equal to str2

> 0 if str1 is greater than str2



Strings Manipulation

Example:

```
#include <iostream.h>

#include <string.h>

void main() {

    cout << strcmp("abc", "def") << endl;

    cout << strcmp("def", "abc") << endl;

    cout << strcmp("abc", "abc") << endl;

    cout << strcmp("abc", "abcdef") << endl;

    cout << strcmp("abc", "ABC") << endl;
```



Strings Manipulation

- ✓ The function **strncmp** is like **strcmp** except that it compares only a specified number of characters.

strncmp(str1, str2, int n);

- ✓ **strncmp** does not compare characters after a terminating null character has been found in one of the strings.



Strings Manipulation

Example:

```
#include <iostream.h>
#include <string.h>
void main()
{
    cout << strncmp("abc", "def", 2) << endl;
    cout << strncmp("abc", "abcdef", 3) << endl;
    cout << strncmp("abc", "abcdef", 2) << endl;
    cout << strncmp("abc", "abcdef", 5) << endl;
    cout << strncmp("abc", "abcdef", 20) <<
endl;
}
```



- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
 - ✓ Sometimes called matrices or tables
- Declaration syntax:

```
dataType  arrayName[intExp1][intExp2];
```

- ✓ intExp1 and intExp2 are expressions with positive integer values specifying the number of rows and columns in the array

□ Accessing components in a two-dimensional array:

```
arrayName[indexExp1][indexExp2]
```

- Where indexExp1 and indexExp2 are expressions with positive integer values, and specify the row and column position
- Example:

sales[5][3] = 25.75;

sales	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]					
[3]					
[4]					
[5]				25.75	
[6]					
[7]					
[8]					
[9]					

sales [5] [3]

Problem with Array

- Although array greatly improved our ability to store data, there is one major drawback to their use ... Each element (each box) in an array must be of the **same data types**.
- It is often desirable to group data of different types and work with that grouped data as one entity . We now have the power to accomplish this grouping with a **new data type** called a **Structure**.
- Structure is a collection of variables of different data types under a single name. It is similar a class in that, both holds a collection of data of different data types.

Structures

- A **Structure** is a collection of related data items, possibly of different types.
- A structure type in C++ is called **struct**.
- A **struct** is **heterogeneous** in that it can be composed of data of different types.
- In contrast, **array** is **homogeneous** since it can contain only data of the same type.

Structure

- Structures hold data that belong **together**.
- Examples:
 - ✓ Student record: student id, name, major, gender, start year, ...
 - ✓ Bank account: account number, name, currency, balance, ...
 - ✓ Address book: name, address, telephone number, ...
- In database applications, structures are called records.

Structure

- ❑ Individual components of a struct type are called **members** (or **fields**).
- ❑ Members can be of **different types** (simple, array or struct).
- ❑ A struct is named as a whole while individual members are named using field identifiers.
- ❑ Complex data structures can be formed by defining **arrays of structs**.

Structure Declaration

- Definition of a structure:

```
struct <struct-type>{  
  <type> <identifier_list>;  
  <type> <identifier_list>;  
  ...  
};
```

} Each
identifier
defines a
member
of the
structure.

- Example:

```
struct Date {  
  int day;  
  int month;  
  int year;  
};
```

} The “Date”
structure
has 3 members,
day, month &
year.

Structure Declaration

■ Example:

```
struct StudentInfo{  
    int Id;  
    int age;  
    char Gender;  
    double CGA;  
};
```



The “StudentInfo”
structure has 4
members
of different types.

■ Example:

```
struct StudentGrade{  
    char Name[15];  
    char Course[9];  
    int Lab[5];  
    int Homework[3];  
    int Exam[2];  
};
```

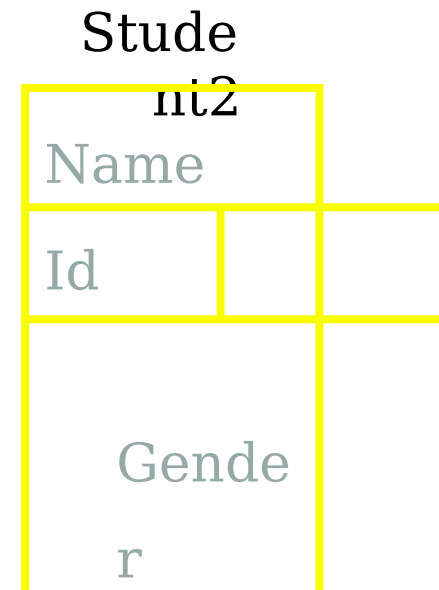
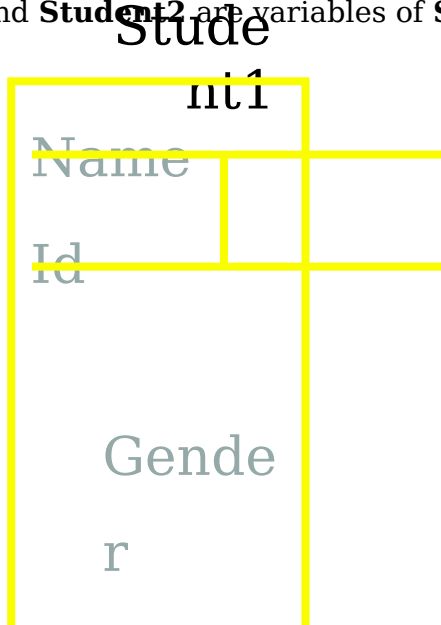


The
“StudentGrade”
structure has 5
members of
different array

Cont.

- Declaration of a variable of struct type:
<struct-type> <identifier_list>;
- Example:
StudentRecord Student1, Student2;

Student1 and **Student2** are variables of **StudentRecord** type.



Structure Example

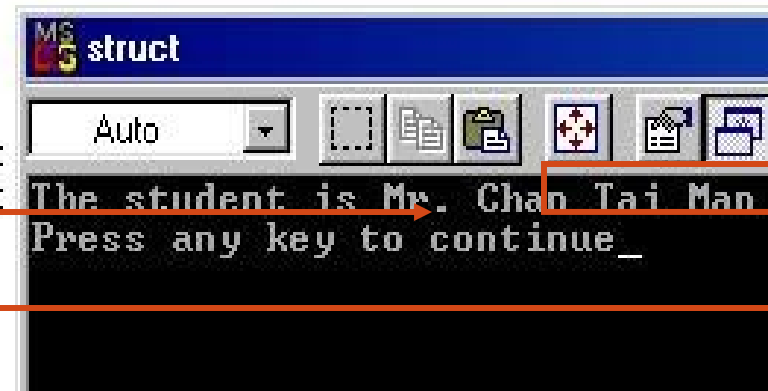
```
#include <string.h>

struct StudentRecord {
    char Name[22];
    int Id;
    char Dept[22];
    char gender;
};

int main() {
    StudentRecord Student1;

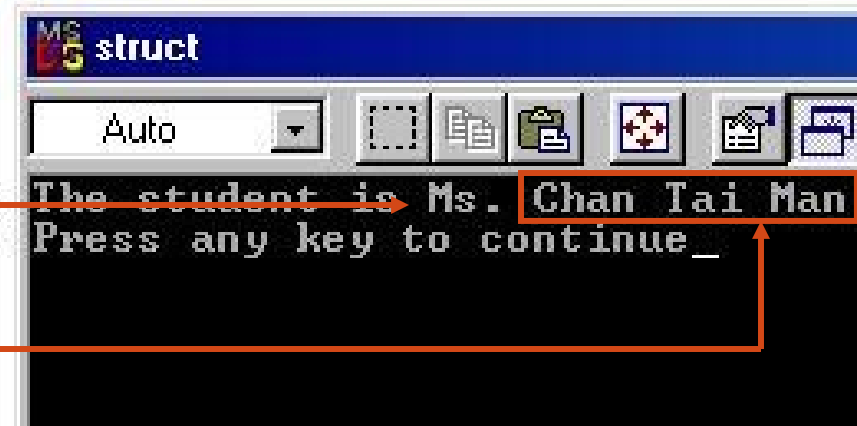
    strcpy(Student1.Name, "Chan Tai Man");
    Student1.Id = 12345;
    strcpy(Student1.Dept, "COMP");
    Student1.gender = 'M';

    cout << "The student is ";
    switch (Student1.gender){
        case 'F': cout << "Ms. "; break;
        case 'M': cout << "Mr. "; break;
    }
    cout << Student1.Name << endl;
    return 0;
}
```



Structure Example

```
struct StudentRecord {  
    char Name[22];  
    int Id;  
    char Dept[22];  
    char gender;  
};  
  
int main() {  
    StudentRecord Student1, Student2;  
  
    strcpy(Student1.Name, "Chan Tai Man");  
    Student1.Id = 12345;  
    strcpy(Student1.Dept, "COMP");  
    Student1.gender = 'M';  
  
    Student2 = Student1;  
    Student2.gender = 'F';  
  
    cout << "The student is ";  
    switch (Student2.gender){  
        case 'F': cout << "Ms. "; break;  
        case 'M': cout << "Mr. "; break;  
    }  
    cout << Student2.Name << endl;  
    return 0;  
}
```



THANK YOU!