**Chapter Five: Compound data Type**

1. **Arrays and Strings**

# What is An Array?

## A collection of identical data objects, which are stored in consecutive memory locations under a common heading or a 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 arrays:

## Arrays in C++ are zero-bounded; that is the index of the first element in the array is 0 and 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 in like other variables. In the array declaration one must define:

## The type of the array (i.e. integer, floating point, char etc.)

## Name of the array,

## 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];

## The expression array size, which is the number of elements, must be a constant such as 10 or a symbolic constant declared before the array declaration, or a constant expression such as 10\*sizeof (int), for which the values are known at the time compilation takes place.

## Note: 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.

## That means, we can store 10 values of type int without having to declare 10 different variables each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, int for example, with a unique identifier.

## 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];

## Every element of day will be set initially to 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 };

## The above declaration would have created an array like the following one:



## 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.

## For example, in the example of the day array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element. If we ignore the last initial value (12071) in the above initialization, 0 will be taken automatically for the last array element.

## Because this can be considered as useless repetition, C++ allows the possibility of leaving empty the brackets [ ], where the number of items in the initialization bracket will be counted to set the size of the array.

**int day [] = { 1, 2, 7, 4, 12,9 };**

## The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day[6])

## 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

## In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable. To access individual elements, index or subscript is used. The format is the following:

name [ index ]

## In c++ the first element has an index of 0 and the last element has an index, which is one less the size of the array (i.e. arraysize-1). Thus, from the above declaration, day[0] is the first element and day[4] is the last element.

## Following the previous examples where day had 5 elements and each element is of type int, the name, which we can use to refer to each element, is the following one:



## For example, to store the value 75 in the third element of the array variable day a suitable sentence would be:

**day[2] = 75; //**as the third element is found at index 2

## And, for example, to pass the value of the third element of the array variable day to the variable a , we could write:

**a = day[2];**

## Therefore, for all the effects, the expression day[2] is like any variable of type int with the same properties. Thus an array declaration enables us to create a lot of variables of the same type with a single declaration and we can use an index to identify individual elements.

## Notice that the third element of day is specified day[2] , since first is day[0] , second day[1] , and therefore, third is day[2] . By this same reason, its last element is day [4]. Since if we wrote day [5], we would be acceding to the sixth element of day and therefore exceeding the size of the array. This might give you either error or unexpected value depending on the compiler.

## In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can cause certain detectable problems, since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen ahead when we begin to use pointers.

## At this point it is important to be able to clearly distinguish between the two uses the square brackets [ ] have for arrays.

## One is to set the size of arrays during declaration

## The other is to specify indices for a specific array element when accessing the elements of the array

## 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;

Eg: 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(); }*

## Strings of Characters:

## What are Strings?

## 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. Until now we have only used them as constants, but we have never considered variables able to contain them.

## 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, which are successions of ****char**** elements. 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:

**char name [20];**

**name**



* This maximum size of 20 characters is not required to be always *fully* used. For example, **name** could store at some moment in a program either the string of characters "Hello" or the string "studying C++”. Therefore, since the array of characters can store *shorter strings* than its total length, there has been reached a convention to end the valid content of a string with a null character, whose constant can be written as **'\0’.**
* We could represent **name** (an array of 20 elements of type **char**) storing the strings of characters "Hello" and "Studying C++" in the following way:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| H | e | L | l | 0 | \0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | t | u | D | y | i | n | g |  | C | + | + | \0 |  |  |  |  |  |  |  |

* Notice how after the valid content it is included a null character (**'\0'**) in order to indicate the end of string. The empty cells (elements) represent indeterminate values.

## Initialization of Strings

* Because strings of characters are ordinary arrays they fulfill same rules as any array. For example, if we want to initialize a string of characters with predetermined values we can do it in a similar way to any other array:

char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };

* In this case we would have declared a string of characters (array) of 6 elements of type **char** initialized with the characters that compose **Hello** plus a null character **'\0’.**
* Nevertheless, string of characters have an additional way to initialize its values: using constant strings.
* In the expressions we have used in examples of previous chapters there have already appeared several times constants that represented entire strings of characters. These are specified enclosed between double quotes ( “ “ ), for example:

Eg: "the result is: "

is a constant string that we have probably used in some occasion.

* Unlike single quotes ( ' ) which allow to specify single character constants, double quotes ( " ) are constants that specify a succession of characters. *These strings enclosed between double quotes have always a null character (‘\0’) automatically appended at the end*.
* Therefore we could initialize the string **mystring** with values by any of these *two* ways:

char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };   
char mystring [] = "Hello";

* In both cases the Array or string of characters **mystring** is declared with a size of 6 characters (elements of type **char** ): the 5 characters that compose **Hello** plus a final null character ( '\0' ) which specifies the end of the string and that, in the second case, when using double quotes ( **"** ) it is automatically appended.
* Before going further, you should note that the assignation of multiple constants like double-quoted constants ( " ) to arrays are only valid *when initializing the array*, that is, at the moment when declared.
* The following expressions within a code are not valid for arrays

mystring="Hello";   
mystring[] = "Hello";

* neither would be: mystring = { 'H', 'e', 'l', 'l', 'o', '\0' };
* So remember: We can "assign" a multiple constant to an Array only at the *moment of initializing* it. The reason will be more comprehensible when you know a bit more about pointers, since then it will be clarified that an array is simply a constant pointer pointing to an allocated block of memory. And because of this constant feature, the array itself cannot be assigned any value, but we can assign values *to each of the elements of the array*.
* At the moment of initializing an Array it is a special case, since it is not an assignation, although the same equal sign (=**)** is used. Anyway, have always present the rule previously underlined.

**Assigning Values to Strings**

* Just like any other variables, array of character can store values using assignment operators. But the following is not allowed.

***mystring=”Hello”;***

* This is allowed only during initialization. Therefore, since the *lvalue* of an assignation can only be an element of an array and not the entire array, what would be valid is to assign a string of characters to an array of **char** using a method like this:

**mystring[0] = 'H';   
mystring[1] = 'e';   
mystring[2] = 'l';   
mystring[3] = 'l';   
mystring[4] = 'o';   
mystring[5] = '\0';**

* But as you may think, this does not seem to be a very practical method. Generally for assigning values to an array, and more specifically to a string of characters, a series of functions like **strcpy** are used. **strcpy** ( **str** ing **c** o **py** ) is defined in the ( string.h ) library and can be called the following way:

**strcpy (** *string1* **,** *string2* **);**

* This does copy the content of *string2* into *string1* . *string2* can be either an array, a pointer, or a constant string , so the following line would be a valid way to assign the constant string **"Hello"** to **mystring** :

***strcpy (mystring, "Hello");***

For example:

*#include <iostream.h>   
#include <string.h>     
int main ()   
{   
char szMyName [20];   
strcpy (szMyName,"Abebe");   
cout << szMyName;   
return 0;   
}*

* Look how we have needed to include **<string.h>** header in order to be able to use function **strcpy.**
* Although we can always write a simple function like the following **setstring** with the same operating than cstring's **strcpy** :

// setting value to string  *#include <iostream.h>     
#include<conio.h>*

*void namecopy(char dest[], char source[])*

*{*

*int c = 0;*

*while(source[c] != ‘\0’)*

*{*

*dest[c] = source[c];*

*c++;*

*}*

*dest[c] = ‘\0’;*

*cout<< “\n your name after copying : ”<<dest;*

*}*

*void main()  
{ clrscr();*

*char name[10],dest[10];*

*cout<< “\n enter your name : ”;*

*cin>>name;*

*namecopy(dest,name);*

*getch();*

*}*

* Another frequently used method to assign values to an array is by using directly the input stream (**cin)**. In this case the value of the string is assigned by the user during program execution.
* When **cin** is used with strings of characters it is usually used with its **getline** method, that can be called following this prototype:
* **cin.getline ( char** buffer **[], int** length **, char** delimiter **= ' \n');**
* where **buffer** is the address where to store the input (like an array, for example), **length** is the maximum length of the buffer (the size of the array) and **delimiter** is the character used to determine the end of the user input, which by default - if we do not include that parameter - will be the newline character ( **'\n'** ).
* The following example repeats whatever you type on your keyboard. It is quite simple but serves as example on how you can use **cin.getline** with strings:

// cin with strings  *#include <iostream.h>    
#include<conio.h>  
int main ()   
{ char mybuffer [100];   
 cout << "What's your name? ";  
 cin.getline (mybuffer,100);   
 cout << "Hello " << mybuffer << ".\n";  
 cout << "Which is your favourite team? ";   
 cin.getline (mybuffer,100);  
 cout << "I like " << mybuffer << " too.\n";   
 getch();*

return 0;   
 }

* Notice how in both calls to **cin.getline** we used the same string identifier (mybuffer**)**. What the program does in the second call is simply step on the previous content of **buffer** by the new one that is introduced.
* If you remember the section about communication through console, you will remember that we used the extraction operator ( **>>** ) to receive data directly from the standard input. This method can also be used instead of **cin.getline** with strings of characters. For example, in our program, when we requested an input from the user we could have written:

cin >> mybuffer;

* this would work, but this method has the following limitations that **cin.getline** has not:
* It can only receive single words (no complete sentences) since this method uses as delimiter any occurrence of a blank character, including spaces, tabulators, newlines and carriage returns.
* It is not allowed to specify a size for the buffer. What makes your program unstable in case that the user input is longer than the array that will host it.
* For these reasons it is recommendable that whenever you require strings of characters coming from **cin** you use **cin.getline** instead of **cin >>** .

**Comparision among Cin to gets() and cout to puts()**

##### 

##### Converting strings to other types

* Due to that a string may contain representations of other data types like numbers it might be useful to translate that content to a variable of a numeric type. For example, a string may contain **"1977”,** but this is a sequence of 5 chars not so easily convertible to a single integer data type. The **cstdlib** ( **stdlib.h** ) library provides three useful functions for this purpose:
* **atoi:** converts string to **int** type.
* **atol:** converts string to **long** type.
* **atof:** converts string to **float** type.
* All of these functions admit one parameter and return a value of the requested type (int, long or float). These functions combined with **getline** method of **cin** are a more reliable way to get the user input when requesting a number than the classic **cin>>** method:

// cin and ato\* functions  *#include <iostream.h>*

*#include <stdlib.h>*

*#include<conio.h>*

*int main()*

*{ clrscr();*

*char mybuffer[100];*

*float price;*

*int quantity;*

*cout << "Enter price: ";*

*cin.getline (mybuffer,100);*

*price = atof (mybuffer);*

*cout << "Enter quantity: ";*

*cin.getline (mybuffer,100);*

*quantity = atoi (mybuffer);*

*cout<<"\nafter conversion :\n";*

*cout<<"\nprice is : "<<price;*

*cout<<"\nquantity is : "<<quantity;*

*cout << "\nTotal price: " << price\*quantity;*

*getch();*

*return 0;*

*}*

**Functions to manipulate strings**

* The **cstring** library ( string.h ) defines many functions to perform some manipulation operations with C-like strings (like already explained strcpy). Here you have a brief with the most usual:

1. **String length**

* Returns the length of a string, not including the null character (\0).

**strlen (const char\*** string **);**

1. **String Concatenation:**

* Appends *src* string at the end of *dest* string. Returns *dest*.
* The string concatenation can have two forms, where the first one is to append the whole content of the source to the destination the other will append only part of the source to the destination.
  + Appending the whole content of the source

**strcat (char\*** dest **, const char\*** src **);**

* + Appending part of the source

**strncat (char\*** dest **, const char\*** src, **int** size **);**

Where size is the number characters to be appended

1. **String Copy:**

* Overwrites the content of the *dest* string by the *src* string. Returns *dest*.
* The string copy can have two forms, where the first one is to copying the whole content of the source to the destination and the other will copy only part of the source to the destination.
  + Copy the whole content of the source

**strcpy (char\*** dest **, const char\*** src **);**

* + Appending part of the source

**strncpy (char\*** dest **, const char\*** src, ***int*** size **);**

Where size is the number characters to be copied

1. **String Compare:**

* Compares the two string *string1* and *string2*.
* The string compare can have two forms, where the first one is to compare the whole content of the two strings and the other will compare only part of the two strings.
  + Compare the whole content of *string1* and *string2*

**strcmp (const char\*** string1 **, const char\*** string2 **);**

* + Compare part of *string1* and *string2*

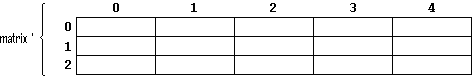
**strncmp (const char\*** string1 **, const char\*** string2, ***int*** size **);**

Where size is the number of characters to be compared

* Both string compare functions returns three different values:
  + Returns ***0*** is the strings are equal
  + Returns ***negative*** value if the first is less than the second string
  + Returns ***positive*** value if the first is greater than the second string

###### Multidimensional Arrays

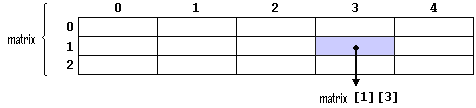
* Multidimensional arrays can be described as arrays of arrays. For example, a bi-dimensional array can be imagined as a bi-dimensional table of a uniform concrete data type.



* Matrix represents a bi-dimensional array of 3 per 5 values of type **int** . The way to declare this array would be:

int matrix[3][5];

* For example, the way to reference the second element vertically and fourth horizontally in an expression would be:

**matrix[1][3]**   
  
(remember that array indices always begin by **0** )

* Multidimensional arrays are not limited to two indices (two dimensions). They can contain so many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example:

char century [100][365][24][60][60];

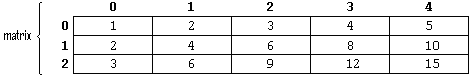
* Assigns a **char** for each second contained in a century, that is more than 3 billion **chars!** What would consume about 3000 megabytes of RAM memory if we could declare it?
* Multidimensional arrays are nothing else than an abstraction, since we can simply obtain the same results with a simple array by putting a factor between its indices:

**int matrix [3][5];**   is equivalent to   
**int matrix [15];**   (3 \* 5 = 15)

* With the only difference that the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bi-dimensional arrays and the other using only simple arrays:

// multidimensional array  *#include <iostream.h>     
#define WIDTH 5   
#define HEIGHT 3     
int matrix [HEIGHT][WIDTH];  
int n,m;     
int main ()   
{ for (n=0;n<HEIGHT;n++)   
   for (m=0;m<WIDTH;m++)   
   {   
      matrix [n][m]=(n+1)\*(m+1);   
   }   
return 0;   
}*

* None of the programs above produce any output on the screen, but both assign values to the memory block called **matrix** in the following way:



* We have used defined constants ( **#define** ) to simplify possible future modifications of the program, for example, in case that we decided to enlarge the array to a height of **4** instead of **3** it would be enough by changing the line:

#define HEIGHT 3

by the following code

#define HEIGHT 4

* 1. **Pointers**
* We have already seen how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 *byte* cells (the minimum size for a datum), each one with a unique address.
* A pointer is a variable which stores the address of another variable. The only difference between pointer variable and regular variable is the data they hold.
* There are two pointer operators in C++:

& the address of operator

\* the dereference operator

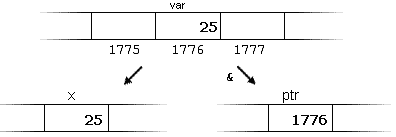
* Whenever you see the & used with pointers, think of the words “address of.” The & operator always produces the memory address of whatever it precedes. The \* operator, when used with pointers, either declares a pointer or dereferences the pointer’s value. The dereference operator can be literally translated to **"value pointed by”.**
* A **pointer** is simply the address of an object in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it is called **dereferencing** the pointer. Pointer variables are defined to point to objects of a specific type so that when the pointer is dereferenced, a typed object is obtained.
* At the moment in which we declare a variable this one must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system on runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored.
* This can be done by preceding the variable identifier by an ampersand sign (&), which literally means, **"address of”.** For example:

*ptr=* &*var;*

* This would assign to variable **ptr** the address of variable **var** , since when preceding the name of the variable **var** with the ampersand ( & ) character we are no longer talking about the content of the variable, but about its address in memory.
* We are going to suppose that **var** has been placed in the memory address **1776** and that we write the following:

var=25;   
x=var;   
ptr = &var;

* The result will be the one shown in the following diagram:



* We have assigned to **x** the content of variable **var** as we have done in many other occasions in previous sections, but to **ptr** we have assigned the address in memory where the operating system stores the value of **var** , that we have imagined that it was **1776** (it can be any address). The reason is that in the allocation of **ptr** we have preceded **var** with an ampersand ( & ) character.
* The variable that stores the address of another variable (like **ptr** in the previous example) is what we call a **pointer.**

## Declaring Pointers:

* Is reserving a memory location for a pointer variable in the heap.

Syntax:

**type \* pointer\_name ;**

* to declare a pointer variable called p\_age, do the following:

*int \* p\_age;*

* Whenever the dereference operator, \*, appears in a variable declaration, the variable being declared is always a pointer variable.

## Assigning values to pointers:

* p\_age is an integer pointer. The type of a pointer is very important. p\_age can point only to integer values, never to floating-point or other types.
* To assign p\_age the address of a variable, do the following:

*int age = 26;*

*int \* p\_age;*

*p\_age =* &*age;*

*OR*

*int age = 26;*

*int \* p\_age =* & *age;*

* Both ways are possible.
* If you wanted to print the value of age, do the following:

*cout<<age;//prints the value of age*

Or by using pointers you can do it as follows

*cout<<\*p\_age;//dereferences p\_age;*

* The dereference operator produces a value that tells the pointer where to point. Without the \*, (i.e cout<<p\_age), a cout statement would print an address (the address of age). With the \*, the cout prints the value at that address.
* You can assign a different value to age with the following statement:

age = 13; //assigns a new value to variable age

\*p\_age = 13 //assigns 13 as a value to the memory p\_age points at.

**N.B:** the \* appears before a pointer variable in only two places: when you declare a pointer variable and when you dereference a pointer variable (to find the data it points to).

* The following program is one you should study closely. It shows more about pointers and the pointer operators, & and \*, than several pages of text could do.

*#...*

*#...*

*int main()*

*{*

*int num = 123; // a regular integer variable*

*int \*p\_num; //declares an integer pointer*

*cout<< “num is ”<<num<<endl;*

*cout<< “the address of num is ”<<*&*num<<endl;*

*p\_num =* &*num;// puts address of num in p\_num;*

*cout<< “\*p\_num is ”<<\*p\_num<<endl; //prints value of num*

*cout<< “p\_num is ”<<p\_num<<endl; //prints value of P\_num*

*}*

## Pointer to void

* Note that we can’t assign the address of a float type variable to an integer pointer variable and similarly the address of an integer variable can not be stored in a float or character pointer.

*flaot y;*

*int x;*

*int \*ip;*

*float \*fp;*

*ip =* &*y; //illegal statement*

*fp =* &*x; //illegal statement*

* That means, if a variable type and pointer to type is same, then only we can assign the address of variable to pointer variable. And if both are different type then we can’t assign the address of variable to pointer variable but this is also possible in C++ by declaring pointer variable as a void as follows:

*void \*p;*

* Let us see an example:

*void \*p;*

*int x;*

*float y;*

*p =* &*x; //valid assignment*

*p =* &*y; //valid assignment*

* The difficulty on void pointers is that, void pointers can not be de referenced. They are aimed only to store address and the dereference operator is not allowed for void pointers.

## Arrays of Pointers

* If you have to reserve many pointers for many different values, you might want to declare an array of pointers.
* The following reserves an array of 10 integer pointer variables:

*int \*iptr[10]; //reserves an array of 10 integer pointers*

* The above statement will create the following structure in RAM

|  |
| --- |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |
| **.** |

iptr[4] = &age;// makes iptr[4] point to address of age.

## Pointer and arrays

* The concept of array goes very bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

*int numbers [20];   
int \* p;*

* the following allocation would be valid:

*p = numbers;*

* At this point **p** and **numbers** are equivalent and they have the same properties, with the only difference that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int with which it was defined. So, unlike **p,** that is an ordinary variable pointer, **numbers** is a constant pointer (indeed that is an Array: a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

*numbers = p;*

* Because **numbers** is an array (constant pointer), and no values can be assigned to constant identifiers.
* **N.B:** An array name is just a pointer, nothing more. The array name always points to the first element stored in the array. Therefore , we can have the following valid C++ code:

*int ara[5] = {10,20,30,40,50};*

*cout<< \*(ara + 2); //prints ara[2];*

* The expression \*(ara+2) is not vague at all if you remember that an array name is just a pointer that always points to the array’s first element. \*(ara+2) takes the address stored in ara, adds 2 to the address, and dereferences that location.
* Consider the following character array:

char name[] = “C++ Programming”;

* What output do the following cout statements produce?

*cout<<name[0]; // \_\_\_C\_\_*

*cout<<\*name; // \_\_\_C\_\_*

*cout<<\*(name+3); //\_\_\_\_\_\_\_\_\_*

*cout<<\*(name+0); //\_\_\_\_C\_\_\_\_*

## Pointer Advantage

* You can’t change the value of an array name, because you can’t change constants. This explains why you can’t assign an array a new value during a program’s execution: eg: if Cname is array of characters then:

*Cname = “Football”; //invalid array assignment;*

* Unlike arrays, you can change a pointer variable. By changing pointers, you can make them point to different values in memory. Have a look at the following code:

*#...*

*#...*

*int main()*

*{*

*float v1 = 679.54;*

*float v2 = 900.18;*

*float \* p\_v;*

*p\_v =* &*v1;*

*cout<< “\n the first value is ”<<\*p\_v;*

*p\_v =* &*v2;*

*cout<< “\n the second value is ”<<\*p\_v;*

*}*

* You can use pointer notation and reference pointers as arrays with array notation. Study the following program carefully. It shows the inner workings of arrays and pointer notation.

*int main()*

*{*

*int ctr;*

*int iara[5] = {10,20,30,40,50};*

*int \*iptr;*

*iptr = iara; //makes iprt point to array’s first element. Or iprt =* &*iara[0]*

*cout<< “using array subscripts:\n”*

*cout<< “iara\tiptr\n”;*

*for(ctr=0;ctr<5;ctr++)*

*cout<<iara[ctr]<< “\t”<< iptr[ctr]<< “\n”;*

*cout<< “\nUsing pointer notation\n”;*

*for(ctr=0;ctr<5;ctr++)*

*cout<< \*(iara+ctr) << “\t” << \*(iptr+ctr)<< “\n”;*

*}*

* Suppose that you want to store a persons name and print it. Rather than using arrays, you can use a character pointer. The following program does just that.

*int main()*

*{*

*char \*c = “Meseret Belete”;*

*cout<< “your name is : ”<<c;*

*}*

* Suppose that you must change a string pointed to by a character pointer, if the persons name in the above code is changed to Meseter Alemu: look at the following code:

*int main()*

*{*

*char \*c = “Meseret Belete”;*

*cout<< “youe name is : ”<<c;*

*c = “Meseret Alemu”;*

*cout<< “\nnew person name is : ”<<c;*

*}*

* If c were a character array, you could never assign it directly because an array name can’t be changed.

## Pointer Arithmetic

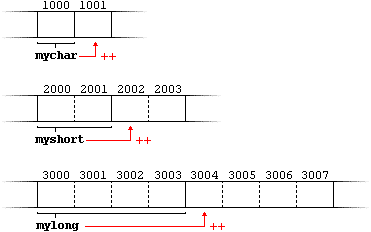
* To conduct arithmetical operations on pointers is a little different than to conduct them on other integer data types. To begin, only *addition* and *subtraction* operations are allowed to be conducted, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point to.
* When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, char occupies 1 byte, short occupies 2 bytes and long occupies 4.
* Let's suppose that we have 3 pointers:

*char \*mychar;   
short \*myshort;   
long \*mylong;*

* And that we know that they point to memory locations **1000** , **2000** and **3000** respectively. So if we write:

*mychar++;   
myshort++;   
mylong++;*

* mychar , as you may expect, would contain the value 1001 . Nevertheless, myshort would contain the value 2002 , and mylong would contain 3004 . The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.



* This is applicable both when adding and subtracting any number to a pointer.
* It is important to warn you that both increase ( ++ ) and decrease ( -- ) operators have a greater priority than the reference operator asterisk ( \* ), therefore the following expressions may lead to confusion:

*\*p++;   
\*p++ = \*q++;*

* The first one is equivalent to \*(p++) and what it does is to increase **p** (the address where it points to - not the value that contains).   
  The second, because both increase operators (++) are after the expressions to be evaluated and not before, first the value of \*q is assigned to \*p and then they are both q and p increased by one. It is equivalent to:

*\*p = \*q;   
 p++;   
 q++;*

* Now let us have a look at a code that shows increments through an integer array:

*int main()*

*{*

*int iara[] = {10,20,30,40,50};*

*int \* ip = iara;*

*cout<<\*ip<<endl;*

*ip++;*

*cout<<\*ip<<endl;*

*ip++;*

*cout<<\*ip<<endl;*

*ip++;*

*cout<<\*ip<<endl;*

*}*

## Pointer and String

* If you declare a character table with 5 rows and 20 columns, each row would contain the same number of characters. You can define the table with the following statement.

*char names[5][20] ={{“George”},{“Mesfin”},{“John”},{“Kim”},{“Barbara”}};*

* The above statement will create the following table in memory:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| G | e | o | r | g | e | \0 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| M | e | s | f | i | n | \0 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| J | o | h | n | \0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| K | i | m | \0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B | a | r | b | a | r | a | \0 |  |  |  |  |  |  |  |  |  |  |  |  |

* Notice that much of the table is waster space. Each row takes 20 characters, even though the data in each row takes far fewer characters.
* To fix the memory-wasting problem of fully justified tables, you should declare a single-dimensional array of character pointers. Each pointer points to a string in memory and the strings do not have to be the same length.
* Here is the definition for such an array:

*char \*name [5] = {{“George”},{“Mesfin”},{“John”}, {“Kim”},{“Barbara”}};*

* This array is a single-dimension array. The asterisk before names makes this array an array of pointers. Each string takes only as much memory as is needed by the string and its terminating zero. At this time, we will have this structure in memory:
* To print the first string, we should use:

*cout<<\*names; //prints George.*

* To print the second use:

*cout<< \*(names+1); //prints Mesfin*

* Whenever you dereference any pointer element with the \* dereferencing operator, you access one of the strings in the array.

## Pointer to pointer:

* As the memory address where integer, float or character is stored in can be stored into a pointer variable, the address of a pointer can also be stored in another pointer. This pointer is said to be pointer to a pointer.
* An array of pointer is conceptually same as pointer to pointer type. The pointer to pointer type is declared as follows:

*Data\_type \*\* pointer\_name;*

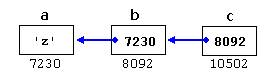
* Note that the asterisk is double here.

*int \*\*p; //p is a pointer which holds the address another pointer.*

*E.g.:*

*char a;   
char \* b;   
char \*\* c;   
a = 'z';   
b =* &*a;   
c =* &*b;*

* This, supposing the randomly chosen memory locations of **7230** , **8092** and **10502** , could be described thus:



*(inside the cells there is the content of the variable; under the cells its location)*

* Have a look at the following code:

*#...*

*int main()*

*{ int data;*

*int \*p1;*

*int \*\*p2;*

*data = 15;*

*cout<< “data = ”<<data<<endl;*

*p1 =* &*data;*

*p2 =* &*p1;*

*cout<< “data through p1 = ”<<\*p1<<endl;*

*cout<< “data through p2 = ”<< \*\*p2<<endl;*

*}*

## Dynamic memory:

* Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program. But, What if we need a variable amount of memory that can only be determined during the program execution (runtime)? For example, in case that we need a user input to determine the necessary amount of space. The answer is dynamic memory, for which C++ integrates the operators new and delete.
* Pointers are useful for creating **dynamic** objects during program execution. Unlike normal (global and local) objects which are allocated storage on the runtime stack, a dynamic object is allocated memory from a different storage area called the **heap**. Dynamic objects do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

## The New Operator

* + In C++ new operator can create space dynamically i.e at run time, and similarly delete operator is also available which releases the memory taken by a variable and return memory to the operating system.
  + When the space is created for a variable at compile time this approach is called static. If space is created at run time for a variable, this approach is called dynamic. See the following two lines:

int a[10];//creation of static array

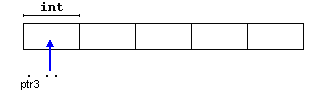
int \*a;

a = new int[10];//creation of dynamic array

* + Lets have another example:

int \* ptr3;   
ptr3 = new int [5];

* + In this case, the operating system has assigned space for 5 elements of type **int** in the heap and it has returned a pointer to its beginning that has been assigned to **ptr3** . Therefore, now, **ptr3** points to a valid block of memory with space for 5 **int** elements.



* + You could ask what is the difference between declaring a normal array and assigning memory to a pointer as we have just done. The most important one is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.
  + The dynamic memory is generally managed by the operating system, and in the multi-task interfaces can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator **new,** a null pointer will be returned. For that reason it is recommendable to always verify if after a call to instruction **new** the returned pointer is null:

*int \* ptr3;   
ptr3 = new int [5];   
if (ptr3 == NULL) {* // error assigning memory. Take measures.  *}*

* + if ptr3 is **NULL**, it means that there is no enough memory location in the heap to be given for ptr3.

## Operator delete

* + Since the necessity of dynamic memory is usually limited to concrete moments within a program, once this one is no longer needed it shall be freed so that it become available for future requests of dynamic memory. For this exists the operator **delete** , whose form is:

**delete** pointer **;**

*or*

**delete []** pointer **;**

* + The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays).
  + In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading.
  + In the following simple example, a program that memorizes numbers, does not have a limited amount of numbers that can be introduced, thanks to the concept and power of pointer that we request to the system as much space as it is necessary to store all the numbers that the user wishes to introduce.

#include <iostream.h>

#include <stdlib.h>

int main ()

{

char input [100];

int i,n;

long \* num;// total = 0;

cout << "How many numbers do you want to type in? ";

cin.getline (input,100);

i=atoi (input);

num= new long[i];

if (num == NULL)

{

cout<<"\nno enough memory!";

exit (1);

}

for (n=0; n<i; n++)

{

cout << "Enter number: ";

cin.getline (input,100);

num[n]=atol (input);

}

cout << "You have entered: ";

for (n=0; n<i; n++)

cout << num[n] << ", ";

delete[] num;

}

* + **NULL** is a constant value defined in C++ libraries specially designed to indicate null pointers. In case that this constant is not defined you can do it yourself by defining it to 0:

###### Structures

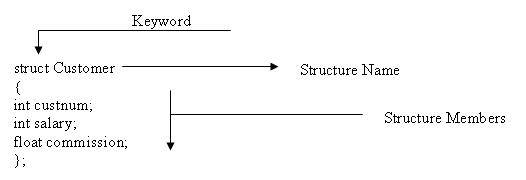
**What is a Structure?**

Structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

**Declaring a Structure*:***

The structure is declared by using the keyword struct followed by structure name, also called a tag. Then the structure members (variables) are defined with their type and variable names inside the open and close braces { and }. Finally, the closed braces end with a semicolon denoted as ; following the statement. The above structure declaration is also called a Structure Specifier.   
  
**Example:**

Three variables: *custnum* of type int, *salary* of type int, *commission* of type float are structure members and the structure name is Customer. This structure is declared as follows:



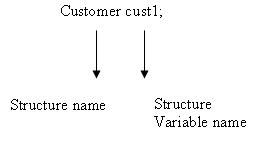
In the above example, it is seen that variables of different types such as int and float are grouped in a single structure name Customer.

Arrays behave in the same way, declaring structures does not mean that memory is allocated. Structure declaration gives a skeleton or template for the structure.   
After declaring the structure, the next step is to define a structure variable.

**How to declare Structure Variable?**

This is similar to variable declaration. For variable declaration, data type is defined followed by variable name. For structure variable declaration, the data type is the name of the structure followed by the structure variable name.

In the above example, structure variable cust1 is defined as:



Here are examples of declaring structure variable

|  |  |
| --- | --- |
| #include<iostream.h>  struct customer  {  int custnum;  char name[10];  int phonenum;  }**cust1,cust2,cust3;//**structure variable declaration  int main()  {  ….  } | #include<iostream.h>  struct customer  {  int custnum;  char name[10];  int phonenum;  }**;**  int main()  {  **struct customer cust1,cust2,cust3; //**structure variable declaration  } |

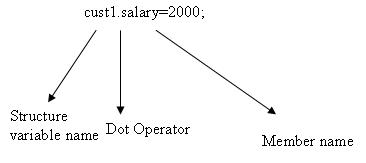
What happens when this is defined? When structure is defined, it allocates or reserves space in memory. The memory space allocated will be cumulative of all defined structure members. In the above example, there are 3 structure members: custnum, name and phonenum. Of these, two are of type in and one is of type char. If integer space allocated by a system is 2 bytes and char one bytes the above would allocate 2bytes for custnum, 2 bytes for phonenum and 1byte for name.

**How to access structure members in C++?**

* Use the dot operator to access or initialize members of structures.
* More importantly, you usually do not even know the contents of the structure variables. Generally, the user enters data to be stored in structures, or you read them from a disk file.
* A better approach to initializing structures is to use the dot operator (.). The dot operator is one way to initialize individual members of a structure variable in the body of your program. With the dot operator, you can treat each structure member almost as if it were a regular non structure variable.
* The General syntax to access members of a structure variable would be: structurevariablename.membername
* A structure variable name must always precede the dot opera- tor, and a member name must always appear after the dot operator. Using the dot operator is easy, as the following examples show.

**For example:**

A programmer wants to assign 2000 for the structure member *salary* in the above example of structure *Customer* with structure variable *cust1* this is written as:



Here are some examples

|  |  |
| --- | --- |
| #include<iostream.h>  struct customer  {  int custnum;  char name[10];  int phonenum;  }**cust1;**  int main()  {  cout<<”Enter customer number”<<endl;  **cin>>cust1.custnum;**  cout<<”Enter customer name”<<endl;  **cin>>cust1.name;**  cout<<”Enter customer phone number”<<endl;  **cin>>cust1.phonenum;**  cout<<”Customer Number:”<<**cust1.custnum**;  cout<<”\nCustomer Name:”<<**cust1.name**;  cout<<”\nCustomer Phone:”<<**cust1.phonenum**;  } | #include<iostream.h>  struct customer  {  int custnum;  char name[10];  int phonenum;  }**;**  int main()  {  **struct customer cust1;**  cout<<”Enter customer number”<<endl;  **cin>>cust1.custnum;**  cout<<”Enter customer name”<<endl;  **cin>>cust1.name;**  cout<<”Enter customer phone number”<<endl;  **cin>>cust1.phonenum;**  cout<<”Customer Number:”<<**cust1.custnum**;  cout<<”\nCustomer Name:”<<**cust1.name**;  cout<<”\nCustomer Phone:”<<**cust1.phonenum**;  } |

**Initializing structure members**

You cannot initialize individual members because they are not variables. You can assign only values to variables. The only structure variable in the above structure is cust1. The braces must enclose the data you initialize in the structure variables, just as they enclose data when you initialize arrays.

This method of initializing structure variables becomes tedious when there are several structure variables (as there usually are). Putting the data in several variables, each set of data enclosed in braces, becomes messy and takes too much space in your code.

You can initialize members when you declare a structure, or you can initialize a structure in the body of the program.

**For example**

|  |  |
| --- | --- |
| Initializing members at declaration | Initializing members at the body of the program |
| #include<iostream.h>  struct employee{  char Emp\_id[15];  char Name[20]; float Salary;  }**emp1={“DMU/001”,”Solomon”,2808};**  int main()  {  cout<<”Your id:”<<emp1.Emp\_id<<endl;  cout<<”Your name:”<<emp1.Name<<endl;  cout<<”Your salay:”<<emp1.Salary<<endl;  } | #include<iostream.h>  #include<string.h>  struct employee{ char Emp\_id[15];  char Name[20]; float Salary;  }emp1;  int main()  {  **strcpy(emp1.Emp\_id,”DMU/001/”);**  **strcpy(emp1.Name,”Solomon”);**  **emp1.Salary=2808;**  cout<<”Your id:”<<emp1.Emp\_id<<endl;  cout<<”Your name:”<<emp1.Name<<endl;  cout<<”Your salay:”<<emp1.Salary<<endl;  } |

**Arrays of Structures**

* Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format.
* Consider the following structure declaration:

*struct Company*

*{*

*int employees;*

*int registers;*

*double sales;*

*}store[1000];*

* In one quick declaration, this code creates 1,000 ***store*** structures with the definition of the ***Company*** structure, each one containing three members.
* NB. Be sure that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable information.
* You can also define the array of structures after the declaration of the structure.

*struct Company*

*{*

*int employees;*

*int registers;*

*double sales;*

*}; // no structure variables defined yet*

*#include<iostream.h>*

*…*

*void main()*

*{*

*struct Company store[1000]; //the variable* ***store*** *is array of the structure Company*

*…  
}*

***Referencing the array structure***

* The ***dot operator (.)*** works the same way for structure array element as it does for regular variables.
* Look the following example

#include<iostream.h>

struct student

{

int id;

float mark;

char name[15];

}stud[2];

int main()

{

for(int i=0;i<=1;i++)

{

cout<<"Enter the id, name and mark of student"<<i+1<<endl;

cin>>stud[i].id>>stud[i].name>>stud[i].mark;

}

for(int i=0;i<=1;i++)

{

cout<<"Id of Student "<<i+1<<"="<<stud[i].id<<endl;

cout<<"Name of Student "<<i+1<<"="<<stud[i].name<<endl;

cout<<"Mark of student "<<i+1<<"="<<stud[i].mark<<endl;

}

}

***Worksheet No 4:***

1. Define strlen function (i.e write the function body of strlen)
2. Define the strcmp function and the strncmp function
3. Define strcpy function and the strncpy function
4. Define strcat function and the strncat function
5. Write a program to store the ages of six of your friends in a single array. Store each of the six ages using the assignment operator. print the ages on the screen
6. Write a C++ program that accepts 10 integers from the user and finally displays the smallest value and the largest value.
7. Write a program that accepts ten different integers from the user and display these numbers after sorting them in increasing order.
8. Write a program to store six of your friend’s ages in a single array. Assign the ages in a random order. print the ages, from low to high, on-screen
9. Modify the program on Q8 to print the ages in descending order.
10. Write a C++ program that calculates the letter grades of 20 students. The program should accept the mid result and the final result from the students. Use the appropriate validity control mechanism to prevent wrong inputs.
11. Write a C++ program that has two functions toBinary and toDecimal. The program should display a menu prompting the user to enter his choice. If the user selects toBinary, then the function should accept a number in base ten and displays the equivalent binary representation. The reverse should be done if the user selects to Decimal.
12. Develop a C++ program that accepts a word from the user and then checks whether the word is palindrome or not. (NB a word is palindrome if it is readable from left to right as well as right to left).
13. Write a C++ program that accepts a word from the user and then displays the word after reversing it.
14. Develop a C++ program that accepts the name of a person and then counts how many vowels the person’s name have.
15. Modify the question in Q14 in such a way that it should replace vowel characters with \* in the person name.
16. Write a program in C++ which read a three digit number and generate all the possible permutation of numbers using the above digits. For example n = 123 then the permutations are – 123, 213, 312, 132, 231, 321
17. Write a program which read a set of lines until you enter #.
18. Write a program which read two matrixes and then print a matrix which is addition of these two matrixes.
19. Write a program which reads two matrix and multiply them if possible
20. Write a program which reads a 3 x 2 matrix and then calculates the sum of each row and store that in a one dimension array.