Chapter Five
Arrays and Structures

An array is:

- ✓ 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, with the first element referred by 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.
- ✓ 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:

I. The type of the array (i.e. integer, floating point, char etc.)
II. Name of the array,
III. 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 declaration is:

Typename arrayname [array size];

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

Note: array type 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];

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.

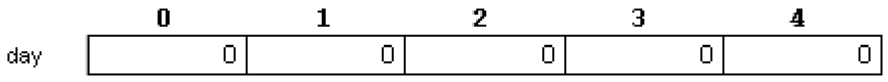
Initializing Arrays

When declaring an array of local scope (within a function), if we do not specify otherwise, it 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];

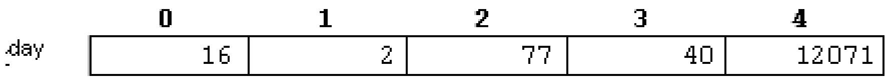
every element of *day* will be set initially to **0:**



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 };

this declaration would have created an array like the following one:



The number of elements in the array that we initialized within curly brackets { } must match the *length* in elements that we declared for the array enclosed within square brackets [ ] . 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.
Because this can be considered an useless repetition, C++ includes the possibility of leaving empty the brackets [ ] , being the size of the Array defined by the number of values included between curly brackets { } :

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

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

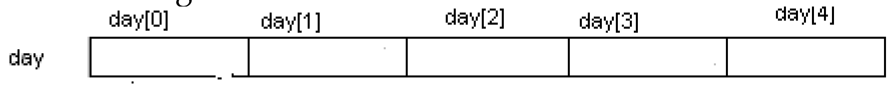
Note: when initializing an array, we can provide fewer values than the array elements. E.g. int a [10] = {10, 2, 3}; in this case the compiler sets the remaining elements to *zero*.

Accessing and processing array elements

In any point of the program in which the array is visible we can access individually anyone of its values 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. size-1) . day[0] is the first element and day[4] is the last element.
Following the previous examples in which *day* had 5 elements and each of those elements was 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 *day* a suitable sentence would be:

day[2] = 75;

and, for example, to pass the value of the third element of 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.

In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can cause certain difficultly 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 that brackets [ ] have related to arrays. They perform two different tasks: one, is to set the size of arrays when declaring them; and second, to specify indices for a concrete array element when referring to it. We must simply take care of not confusing these two possible uses of brackets [ ] with arrays:

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;
```

```
// arrays example ,display the sum of the numbers in
//the array

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

Arrays as parameters

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, even if it is ordered as an array, to a function, 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 when declaring the function is to specify 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:

```
// arrays as parameters

#include <iostream>
#include <conio.h>
void printarray (int arg[], int length)
{
     for (int n=0; n<length; n++)
         cout << arg[n] << " ";
      cout << "\n";
}

void mult(int arg[], int length)
{
    for (int n=0; n<length; n++)
      arg[n]=2*arg[n];
}

int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    mult(firstarray,3);
    cout<<"first array after doubled\n";
    printarray (firstarray,3);
    return 0;
}
```

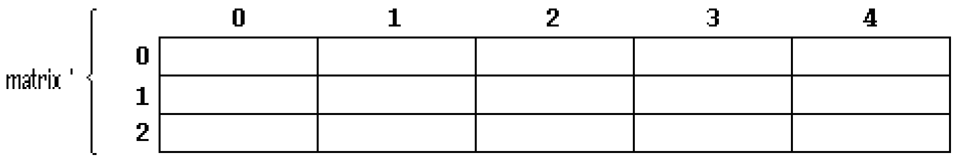
As you can see, the first argument (**int arg[]**) admits any array of type **int** , whatever its length is, for that reason we have included a second parameter that says to the function the *length* of each array that we pass to it as the first parameter so that the **for** loop that prints out the array can know the range to check in the passed array. The function mult doubles the value of each element and the firstarray is passed to it. After that the display function is called. The output is modified showing that arrays are passed by reference. *To pass an array by value, pass each element to the function*

In a function declaration it is also possible to include multidimensional arrays. Here is an example:

```
void procedure (int myarray[[3][4])
```

Multidimensional Arrays

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

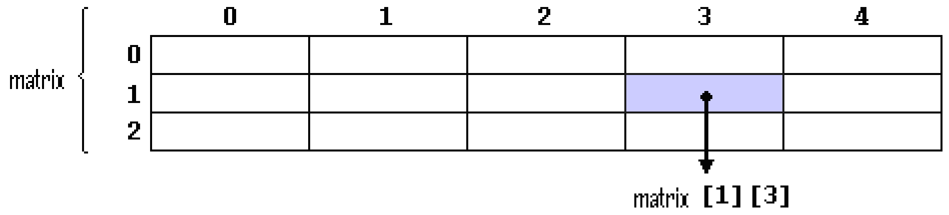


matrix represents a bidimensional array of 3 per 5 values of type **int** . The way to declare this array would be:

```
int matrix[3][5];
```

and, 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)

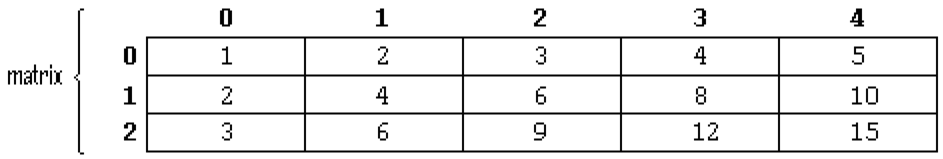
With the only difference that the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bidimensional arrays and the other using only simple arrays:

```
// multidimensional array

#include <iostream>
#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:



matrix	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

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

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Structures

- a structure is a collection of one or more variable types grouped together. You can refere to a structure as a single variable, and you also can initializae, read and change the parts of a structure (the individual variables that make it up).
- Each element (called a member) in a structure can be a different data type.
- The syntax of structures is:

```
Struct [structure tag]{
        Member definition;
        Member definition;
        …
        Member definition;
}[one or more structure variables];
```

- Let us see an example

```
                                        structure tag
Eg. struct Inventory{
        char description[15];
        char part_no[6];
        int quantity;                   members of the structure
        float cost;
}; // all structures end with semicolon
```

- Structure tag is not a variable name. unlike array names, which reference the array as variables, a structure tag is simply a label for the structure's format.

- The structure tag Inventory informs C++ that the tag called Inventory looks like two character arrays followed by one integer and one float variables.
- A structure tag is actually a newly defined data type that you, the programmer, defined.

Initializing Structure Data

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

```
#include<iostream>

void main()
{
  clrscr();
  struct cd_collection{
    char title[25];
    char artist[20];
    int num_songs;
    float price;
    char date_purchased[9];
  }cd1 = {"Red Moon Men","Sams and the Sneeds",
      12,11.95f,"08/13/93"};
  cout<<"\nhere is the info about cd1"<<endl;
    cout<<cd1.title<<endl;
    cout<<cd1.artist<<endl;
    cout<<cd1.num_songs<<endl;
    cout<<cd1.price<<endl;
    cout<<cd1.date_purchased<<endl;
  getch();
}
```

- A better approach to initialize 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. The syntax of the dot operator is :

    structureVariableName.memberName

here is an example:

```
#include<iostream>

#include<string.h>
void main()
{
    clrscr();
    struct cd_collection{
        char title[25];
        char artist[20];
        int num_songs;
        float price;
        char date_purchased[9];
    }cd1;
    //initialize members here
    strcpy(cd1.title,"Red Moon Men");
    strcpy(cd1.artist,"Sams");
    cd1.num_songs= 12;
    cd1.price = 11.95f;
    strcpy(cd1.date_purchased,"22/12/02");
    //print the data
    cout<<"\nHere is the info"<<endl;
    cout<<"Title : "<<cd1.title<<endl;
    cout<<"Artist : "<<cd1.artist<<endl;
    cout<<"Songs : "<<cd1.num_songs<<endl;
    cout<<"Price : "<<cd1.price<<endl;
    cout<<"Date purchased : "<<cd1.date_purchased;
    getch();
  }
```

<u>Arrays of Structures</u>

- 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 declariation:

```
  struct Company{
        int employees;
        int registers;
        double sales;
  }store[1000];
```

- In one quick declaration, this code creates 1,000 store structures, 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 declariation of the structure.

```
struct Company{
      int employees;
```

```
        int registers;
        double sales;
};// no structre variables defined yet

#include<iostream>
…
void main()
{
        Company store[1000];
        …
}
```

Referencing the array structure

- The sot operator works the same way for structure array element as it does for regular variables. If the number of employees for the fifth store (store[4]) increased by three, you could update the structure variable like this:

  ```
  store[4].employees += 3;
  ```

- You can also assign complete structures to one another by using array notation. To assign all the members of the 20th store to the 45th store, you would do this:

  ```
  store[44] = store[19];//copies all members from 20th store to 45th
  ```

here is a complete C++ code that shows you how to use array of structures, and how to pass and return structures to functions.

```
        #include<iostream>

#include<stdio.h>
#include<iomanip.h>
struct inventory{
   long storage;
   int accesstime;
   char vendorcode;
   float cost;
   float price;
};
void disp_menu(void);
struct inventory enter_data();
void see_data(inventory disk[125],int num_items);

void main()
```

```
{
   clrscr();
   inventory disk[125];
   int ans;
   int num_items = 0; //total number of items in the inventory

   do{
         do{
            disp_menu();
            cin>>ans;
          }while(ans<1 || ans>3);

          switch(ans)
          {
            case 1:
                  disk[num_items] = enter_data();
                  num_items++;
                  break;
            case 2:
                  see_data(disk,num_items);
                  break;
            default :
                  break;
          }
        }while(ans != 3);
        return;
}//end main

void disp_menu()
{
   cout<<"\n\n*** Disk Drive Inventory System ***\n\n";
   cout<<"Do you want to : \n\n";
   cout<<"\t1. Enter new item in inventory\n\n";
   cout<<"\t2. See inventory data\n\n";
   cout<<"\t3. Exit the program\n\n";
   cout<<"What is your choice ? ";
   return;
}

inventory enter_data()
{
  inventory disk_item;//local variable to fill with input
  cout<<"\n\nWhat is the next drive's storage in bytes? ";
  cin>>disk_item.storage;
  cout<<"\nWhat is the drive's access time in ms ? ";
  cin>>disk_item.accesstime;
  cout<<"What is the drive's vendor code (A, B, C, or D)? ";
  disk_item.vendorcode = getchar();
  cout<<"\nWhat is the drive's cost? ";
  cin>>disk_item.cost;
  cout<<"\nWhat is the drive's price? ";
  cin>>disk_item.price;
  return (disk_item);
}

void see_data(inventory disk[125], int num_items)
{
  int ctr;
  cout<<"\n\nHere is the inventory listing:\n\n";
  for(ctr=0;ctr<num_items;ctr++)
  {
    cout<<"Storage: "<<disk[ctr].storage<<"\n";
    cout<<"Access time: "<<disk[ctr].accesstime<<endl;
    cout<<"Vendor code: "<<disk[ctr].vendorcode<<"\n";
    cout<<"Cost : $ "<<disk[ctr].cost<<"\n";
    cout<<"Price: $ "<<disk[ctr].price<<endl;
  }
  return;
}
```

Chapter Six

Pointers

Introduction

- 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.

Declaring Pointers:

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 declariation, 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 //this stmt assigns 13 to the value p_age points to.
```

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 fins 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.

```
#...
#...
Void 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
   Getch();
}
```

Pointer to void

- note that we can't assign the address of a float type variable to int. eg:

```
flaot y;
int *p;
p = &y; //illegal statement
```

- similarly see the following

```
float * p;
int x;
p = &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:

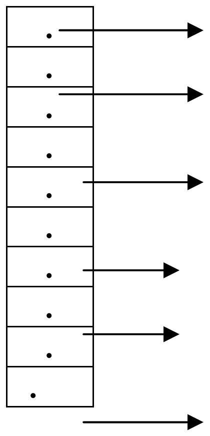
```
Void *p;
```

```
Int x;
Float y;
P = &x; //ok
P = &y; //ok
```

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 structur 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. There for , 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
  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:

```
#...
#...
Void main()
{
        Clrscr();
        Float v1 = 679.54;
        Float v2 = 900.18;
        Float * p_v;

        P_v = &v1;
        Cout<< "\nthe first value is "<<*p_v;
        P_v = &v2;
        Cout<< "\nthe second value is "<<*p_v;
        Getch();
}
```

- 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.

```
Void main()
{
        Clrscr();
        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”;
        {
                Cout<< *(iara+ctr) << “\t” << *(iptr+ctr)<< “\n”;
        }
        Getch();
}
```

- 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.

```
Void main()
{
        Clrscr();
        Char *c = “Meseret Belete”;
        Cout<< “your name is : ”<<c;
}
```

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

```
Void main()
{
        Char *c = “Meseret Belete”;
        Cout<< “youe name is : ”<<c;
        C = “Meseret Alemu”;
        Cout<< “\nnew person name is : ”<<c;
        Getch();
}
```

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

<u>Pointer Arithmetic</u>

- 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.

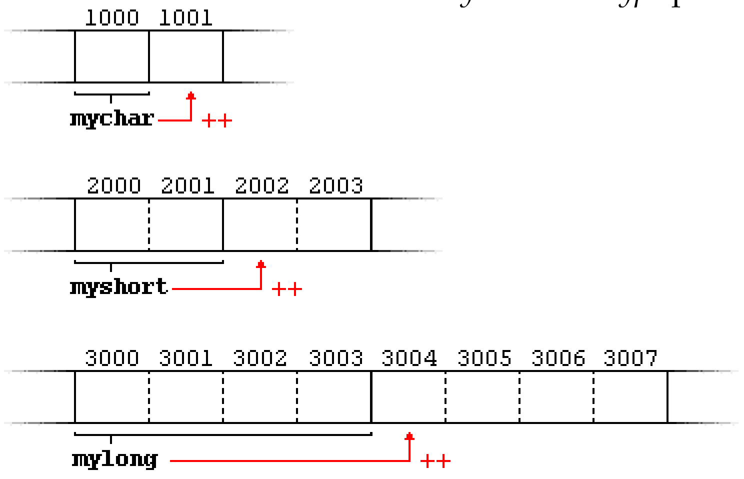
- When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, *char* occupies 1 byte, *short* occupies 2 bytes and *long* occupies 4.
- Let's suppose that we have 3 pointers:

```
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 confussion:

```
*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:

```
Void 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;
        Cout<< "\nthe integer size is "<<sizeof(int)<<
        "bytes on this machine\n\n";
}
```

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 astrix 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, you would use cout<<*names; //prints George. To print the second use cout<< *(names+1); prints Michael
- Whenever you dereference any pointer element with the * dereferencing operator, you access one of the strings in the array.

Pointer to 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;
```

Eg: int **p; where p is a pointer which holds the address another pointer.

- Have a look at the following code:

```
#...
#...
Void main()
{
        Clrscr();
        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;
        Getch();
}
```

Dynamic memory:

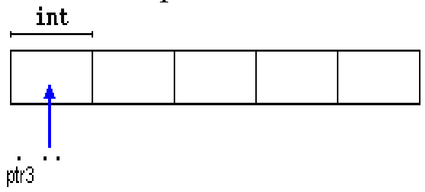
- Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them to be fixed before the execution of the program. But, What if we need a variable amount of memory that can only be 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.*
- 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 multitask 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.
  };
```

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.

```
#include <iostream>
#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!";
          getch();
          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;
    getch();
    return 0;

}
```

- This simple example that memorizes numbers does not have a limited amount of numbers that can be introduced, thanks to that we request to the system as much space as it is necessary to store all the numbers that the user wishes to introduce.
- **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: