

Chapter Four

Pointers

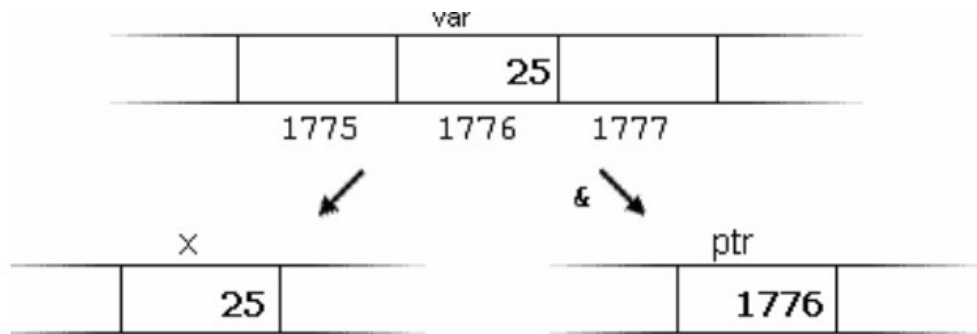
4. Introduction

- 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*. Notice that *pointer variable holds address; address of some variable or pointer variable while ordinary variables hold values of their respective types*.
- There are two pointer operators in C++:
 - & the address of operator
 - * the dereference operator
 - the value of the variable pointed by this pointer will be returned
- 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:
 - o `ptr = &var;`
 - o 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.
 - o 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;
```
 - o The result will be the one shown in the following diagram:



`ptr = var; //produces error`

`ptr = & var; //address of var is assigned to ptr`

- 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 is **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**.

4.1. Declaring Pointers:

- Is reserving a memory location for a pointer variable in the heap. Syntax:
 - **`data 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.

4.2. 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:
 - o `cout<<age; //prints the value of age`
- Or by using pointers you can do it as follows
 - o `cout<<*p_age; //dereferences p_age;`
 - o 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:
 - o `age = 13; //assigns a new value to variable age`
 - o `*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).
 - o 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.
 - o

```
#...
#...
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
        *p_num = 200;
        *p_num = *p_num + 50;
        getch();
    }

```

4.3. 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 cannot be stored in a float or character pointer.

```

    float 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:

```

    o 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 cannot be dereferenced.

They are aimed only to store address and the dereference operator is not

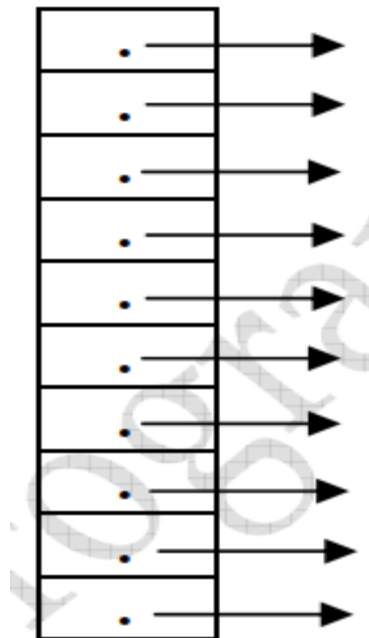
allowed for void pointers.

- Let us see an example:

```
#include <iostream.h>
#include <conio.h>
void main()
{
    void *p;
    int x=10;
    float y=10.5;
    p = &x;    //valid assignment
    p = &y;    //valid assignment
    cout<<p;   //shows the address of the variable y
    cout<<*p;  //produces error
    getch();
}
```

4.4. 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.
iptr[5] = &x; // makes iptr[5] point to address of x.

4.5. 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.

- **An array name is just a pointer that always points to the array's first element.**

- 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; //not valid
```

- Look at the following example

```
void main()
```

```
{
```

```
    int numbers [20]={10,20};
```

```
    int * p;
```

```
    p=numbers; //address of numbers is assigned to p
```

```
    numbers = p ; //error L value required
```

```
    cout<<numbers; //displays the address of numbers and  
the same as p for p=nu...
```

```
    getch();
```

```
}
```

- 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__
```

4.6. 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, it is allowed during declaration
```
- 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:


```
o #...
  #...
  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.

```

#include <iostream.h>
#include <conio.h>

void main()
{
    clrscr();
    int i;
    int array[5] = {10,20,30,40,50};
    int *ptr;
    ptr = array; //makes iprt point to array's first element. Or iprt = &array[0]
    cout<< "using array subscripts:\n" ;
    cout<< "array\tptr\n";
    for(i=0;i<5;i++)
    {
        cout<<array[i]<< "\t"<< ptr[i]<< "\n";
    }
    cout<< "\nUsing pointer notation\n";
    for(i=0;i<5;i++)
    {
        cout<< *(array+i) << "\t" << *(ptr+i)<< "\n";
    }
    getch();
}

```

```

using array subscripts:
iara    iptr
10      10
20      20
30      30
40      40
50      50

Using pointer notation
10      10
20      20
30      30
40      40
50      50

```

- 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.

```

o void main()
{
clrscr();
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:

```

o void main()
{
char *c = "Meseret Belete";
cout<< "youe name is : "<<c;
c = "Meseret Alemu";
cout<< "\nnew person name is : "<<c;
getch();
}

```

```

youe name is : Meseret Belete
new person name is : Meseret Alemu

```

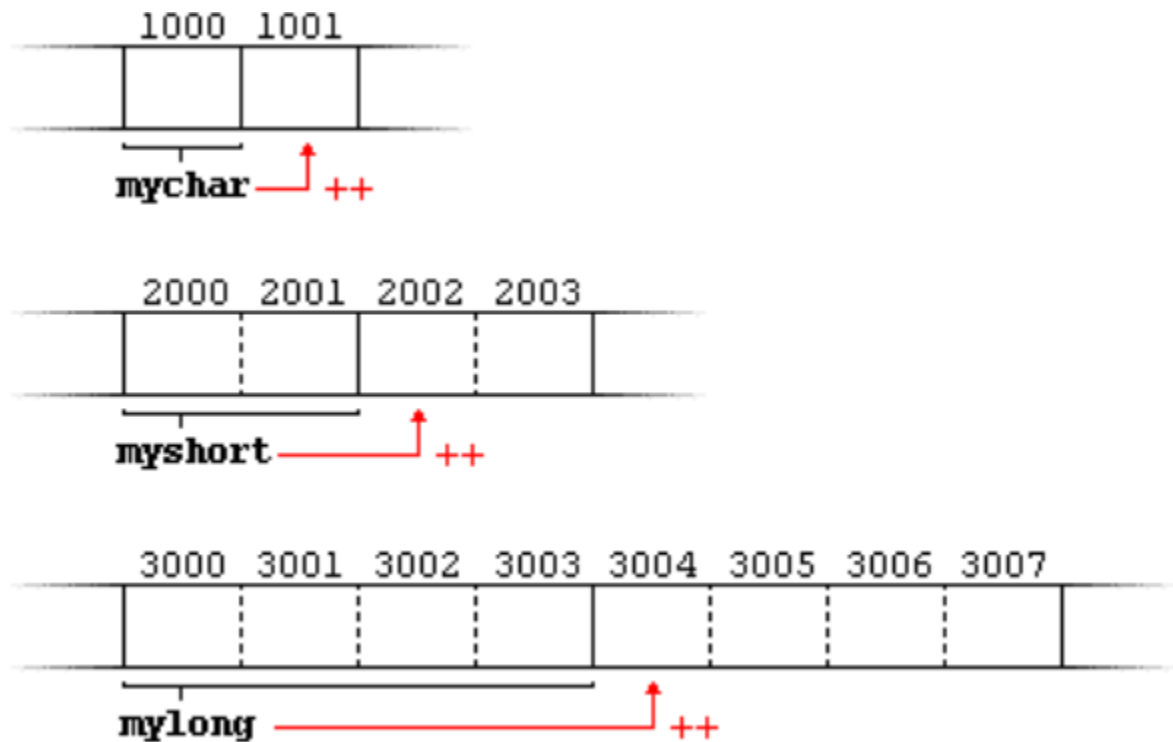
- If *c* were a character array, you could never assign it directly because an array name can't be changed.

4.7. 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:

```
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;
}

```

4.8. 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"}};`

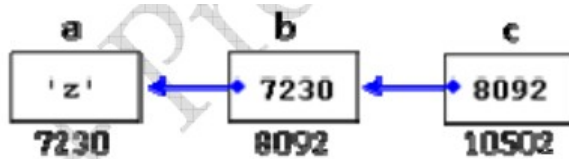
- This array is a single-dimension array. The asterisk before names makes this array an array of pointers. Each string takes only as much memory as is needed by the string and its terminating zero. At this time, we will have this structure in memory:
- To print the first string, we should use:
*cout<<*names; //prints George.*
- To print the second use:
*cout<< *(names+1); //prints Michael*
- Whenever you dereference any pointer element with the * dereferencing operator, you access one of the strings in the array.

4.9. 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:

```

#...
#...
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();
}
  
```

4.10. 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.

- a) 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:

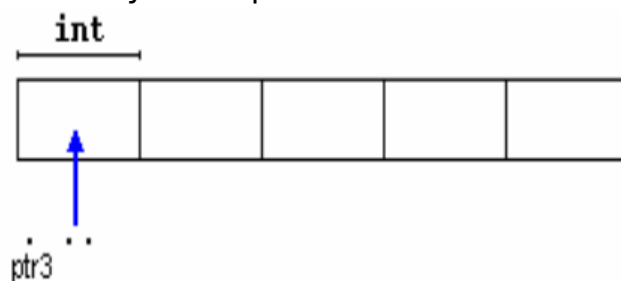
```
int * ptr3;
```

```
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.

b) 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>  
#include <conio.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;
}

```

- **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: