String

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	o	\o														

S	t	u	d	y	i	n	g		C	+	+	\o							

- 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. This 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 >>** .

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 **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:

a) String length

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

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

b) 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

c) 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

d) 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.
  - ✓ Copy the whole content of the source

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

  - ✓ Appending part of the source

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

    Where size is the number 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