

C++ STRING

- 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	\0														
---	---	---	---	---	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--

S	r	u	d	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` (string copy) 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 >>** .

Defining a String

- A string is an array of characters whose last character is \0. A typical string, such as Pacifique, is graphically represented as follows:

P a c i f i q u e \0

P	a	c	i	f	q	u	e	\0	
---	---	---	---	---	---	---	---	----	--

- The last character \0 is called the null-terminating character. For this reason, a string is said to be null-terminated. The string library ships with a lot of functions used to perform almost any type of operation on almost any kind of string. Used under different circumstances, the string functions also have different syntaxes.
- The strings that you can use in your program may be defined in various libraries depending on your compiler but most of the time, they are available once you include the string library that is defined in the **std namespace**. For example, the functions we are about to review are part of the C language. The strings that are part of the (C++) Standard Template Library (STL) are defined in the string class of the **std namespace**. Based on this, most compilers make all these functions accessible once you include the string library and the std namespace in your program.

String Manipulation Functions

- The C++ and its parent the C languages do not have a string data type. In C and C++, strings are created from arrays. Therefore, the C++ language relies on operations performed on the arrays of characters or pointers to char. The functions used for this purpose are numerous and you should know what they are used for.

As a reminder, thanks to the features of arrays of characters and the friendship between arrays of characters and strings, there are two main ways you can declare and initialize a string:

```
char *Thing = "Telephone";  
char Major[] = "Computer Sciences";
```

1. The Length of a String

In many operations, you will want to know how many characters a string consists of. To find

the number of characters of a string, use the `strlen()` function. Its syntax is:

```
int strlen(char Value);
```

The `strlen()` function takes one argument, which is the string you are considering. The function returns the number of characters of the string.

Here is an example:

```
char School[] = "Manchester United";  
int Length = strlen(School);  
cout << "The length of " << School<<"is"<<Length;
```

The length of Manchester United is 17 characters

2. The `strcat()` Function

If you have two strings, to append one to another, use the **`strcat()`** function. Its syntax is:

```
char strcat(char Destination, char Source);
```

The **`strcat()`** function takes two arguments. The second argument, called the source string, is the string you want to add to the first string; this first string is referred to as the destination.

Although the function takes two arguments. It really ends up changing the destination string by appending the second string at the end of the first string. This could be used to add two strings.

Here is an example:

```
char Make[] = {"Ford "};  
char Model[] = {"Explorer"};  
cout << "Originally, Make = " << Make;  
strcat(Make, Model);  
cout << "\n\nAfter concatenating, Make = " << Make << endl;
```

Output

Originally, Make = Ford

After concatenating, Make = Ford Explorer

3. The strncat () Function

Like the strcat() function, the strncat() function is used to append one string to another. The difference is that, while the strcat () considers all characters of the source string, the strncat() function allows you to specify the number of characters from the source string that you want to append to the destination string. This means that, if the source string has 12 characters, you can decide to append only a set number of its characters. The syntax is:

char strncat(char Destination, char Source, int Number);

Besides the same arguments as the strcat () function, the Number argument sets the number of characters considered from Source. To perform the concatenation, the compiler would count characters from left to right on the source string.

Here is an example:

```
char Make[] = {"Ford "};  
char Model[] = {"Explorer"};  
cout << "Originally, Make = " << Make;  
strcat(Make, Model,3);  
cout << "\n\nAfter concatenating, Make = " << Make << endl;
```

Output

Originally, Make = Ford

After concatenating, Make = Ford Exp

4. Copying One String into Another

The strcpy () function is used to copy one string into another string. In English, it is used to replace one string with another. The syntax of the strcpy () function is:

char strcpy (char Destination, char Source);

This function takes two arguments. The first argument is the string that you are trying to replace. The second argument is the new string that you want to replace. There are two main scenarios suitable for the strcpy () function: To replace an existing string or to initialize a string.

```
char carName1[] = {"Ford Escort"};
char carName2[] = {"Toyota 4-Runner"};
cout<<strcpy(carName2, carName1);
```

Output

Ford Escort

5. The strncpy () Function

The strncpy() function works like the strcpy() function. As a difference, the strncpy () function allows you to specify the number of characters that the compiler would copy from the source string.

Here is the syntax:

char strncpy(char Destination, char Source, int Number);

The Number argument specifies the number of characters that will be copied from the Source string. Here is an example:

```
char carName1[] = {"Ford Escort"};
char carName2[] = {"Toyota 4-Runner"};
cout<<strncpy(carName2, carName1,8);
```

Output

Ford Esc-Runner

Comparing Strings

Various routines are available for strings comparison. The C-String library is equipped with functions that perform the comparisons by characters. Alternatively, some compilers like Borland C++ Builder ships with additional multiple functions to perform these comparisons on null-terminated strings.

6. The strcmp() Function

The strcmp () function compares two strings and returns an integer as a result of its comparison. Its syntax is:

strcmp (char S1, char S2);

This function takes two strings, S1 and S2 and compares them. It returns

A negative value if S1 is less than S2

0 if S1 and S2 are equal

A positive value if S1 is greater than S2

7. The strncmp () Function

The strncmp () function compares two strings using a specified number of characters and returns an integer as a result of its findings. Its syntax is:

strncmp (const char S1, char S2, int Number);

This function takes three arguments. The first two arguments are the strings that need to be compared. The 3rd argument specifies the number of characters considered for the comparison. It returns

A negative value if S1 is less than S2

0 if S1 and S2 are equal

A positive value if S1 is greater than S2

8. The stricmp() Function

The **stricmp ()** function compares two strings without regard to their case. In other words, this function does not take into consideration if there is a mix of uppercase and lowercase characters in the strings. The syntax of the function is:

stricmp(char S1, const char S2);

This function takes two strings, S1 and S2 and compares them. It returns
A negative value if S1 is less than S2
0 if S1 and S2 are equal
A positive value if S1 is greater than S2

9. The strnicmp() Function

The strnicmp () function compares two strings without regard to their case but considers only a specified number of characters. The syntax of the function is:

int strnicmp (char S1, char S2, int n);

This function takes two strings, S1 and S2 and compares them. It returns
A negative value if S1 is less than S2
0 if S1 and S2 are equal
A positive value if S1 is greater than S2

Working with Individual Characters

10. The strchr() Function

The strchr () function looks for the first occurrence of a certain character in a string. Its syntax is:

strchr(const char S, char c);

This function takes two arguments. The second argument specifies what character to look for in the first argument which is a string. If the character c appears in the string S, the function would return a new string whose value starts at the first occurrence of c in S. If the character c does not appear in the string S, then the function would return NULL.

Working with Sub-Strings

11.The strstr() Function

The strstr () function looks for the first occurrence of a sub-string in another string and returns a new string as the remaining string. Its syntax is:

char strstr(char Main, char Sub);

The first argument of the function is the main string that would be examined. The function would look for the second argument, the Sub string appearance in the main string. If the

Sub string is part of the Main string, then the function would return a string whose value starts at the first appearance of Sub and make it a new string. If Sub is not part of the Main string, the function would return a NULL value.

Working with Character Cases

12.The **strlwr()** Function

The **strlwr()** function is used to convert a string to lowercase. Its syntax is:

char strlwr(char S);

This function takes, as argument, the string that needs to be converted. During conversion, if a Latin character were in uppercase, it would be converted to lowercase. Otherwise, it would stay “as is”. This means any symbol that is not a readable character would not be converted.

13.The **strupr()** Function

The **strupr()** function is used to convert a string to uppercase. Its syntax is:

char strupr(char S);

Each lowercase character in the function’s argument, S, would be converted to uppercase. Any character or symbol that is not in lowercase would not be changed.

C++ POINTER

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.
- 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**”.

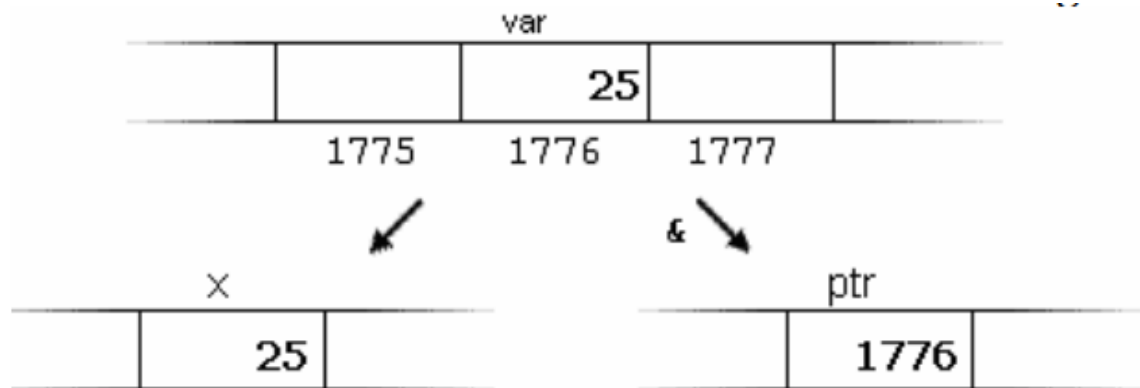
For example:

```
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:

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.

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.

```
#...
#...
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 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:

```
void *p;
```

- Let us see an example:

```
void *p;
```

```
int x;
```

```
float y;
```

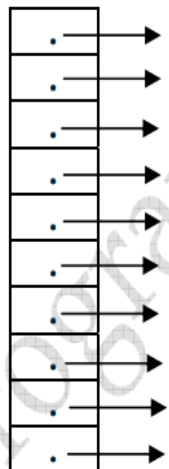
```
p = &x; //valid assignment
```

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

- The difficulty on void pointers is that, void pointers cannot be 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. 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: e.g.: 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:

```
#...
#...
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 iptr point to array's first element. Or iptr = &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";
}
getch();
}

```

- Suppose that you want to store a person's 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 change a string pointed to by a character pointer, if the person's name in the above code is changed to Meseret 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.

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;

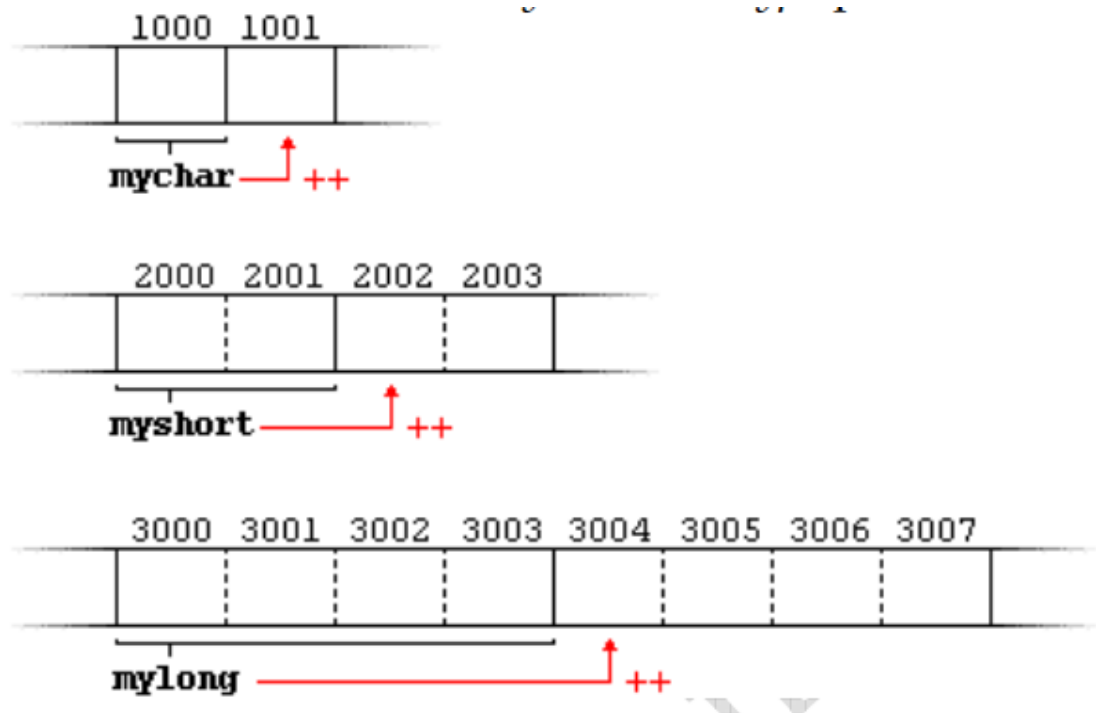
- And that we know that they point to memory locations **1000** , **2000** and **3000** respectively. So if we write:

```
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;
}
```

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.

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()
{
  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`.
- 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.

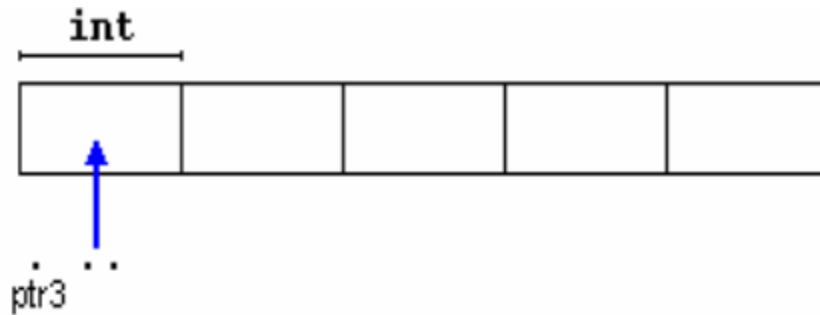
The New Operator

- 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
```

- Let's 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 the difference between is 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>
#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:

C++ STRUCTURES

- A structure is a collection of one or more variable types grouped together that can be referred using a single name (group name) and a member name. You can refer to a structure as a single variable, and you also can initialize, 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 of different data type.

- The General 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

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

- Another example might be:

```
struct Student
{
    char ID[8];
    char FName[15];
    char LName[15];
    char Sex;
    int age;
    float CGPA;
    };
```

- The above “Student” structure is aimed to store student record with all the relevant details.

- After the definition of the structure, one can declare a structure variable using the structure tag. If we need two variables to have the above structure property then the declaration would be:

```
struct Inventory inv1, inv2; //or
struct Student Stud1, stud2, Stud3;
```
- 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.

Referencing members of a structure

- To refer to the members of a structure we need to use the **dot operator (.)**
- The General syntax to access members of a structure variable would be:
VarName.Member
- Where *VarName* is the variable name of the structure variable And *Member* is variable name of the members of the structure.
- **Eg:** For the above student structure:

```
struct Student Stud; //declaring Stud to have the property of the Student structure
strcpy(Stud.FName,"Abebe"); //assigned Abebe as First Name
Stud.CGPA=3.21; //assignes 3.21 as CGPA value of Abebe
cout<<Stud.FName; //display the name
cout<<Stud.CGPA; // display the CGPA of Abebe
```

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.

```
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, 11.95, "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;
```

- 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.h>
#include<conio.h>
#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();
}

```

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. 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;`
- Unlike in the case of arrays, where the whole content of an array could not be copied to another one using a simple statement, in structures, you can 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.

```
#include <iostream.h>
#include <cstring.h>
#include <conio.h>
struct student
{
    int roll_no;
    string name;
    int phone_number;};
```

```

int main()
{
    struct student stud[5];

    for(int i=0; i<5; i++){          //taking values from user
        cout << "Student " << (i + 1) << endl;
        cout << "Enter roll no" << endl;
        cin >> stud[i].roll_no;
        cout << "Enter name" << endl;
        cin >> stud[i].name;
        cout << "Enter phone number" << endl;
        cin >> stud[i].phone_number;
    }

    for(i=0; i<5; i++) //printing values
    {
        cout << "Student " << (i + 1) << endl;
        cout << "Roll no : " << stud[i].roll_no << endl;
        cout << "Name : " << stud[i].name << endl;
        cout << "Phone no : " << stud[i].phone_number << endl;
    }
    getch();
    return 0;
}

```

- Here is a complete C++ code that shows you how to pass and return structures to functions.

```

#include <iostream.h>
#include <cstring.h>
#include <conio.h>
struct student
{
    int roll_no;
    string name;
    int phone_number;
};
void display(struct student st)
{
    cout << "Roll no : " << st.roll_no << endl;
    cout << "Name : " << st.name << endl;
    cout << "Phone no : " << st.phone_number << endl;
}

```

```
}  
int main(){  
    struct student s;  
        s.roll_no = 4;  
        s.name = "Ron";  
        s.phone_number = 888888;  
        display(s);  
    getch();  
        return 0;  
}
```

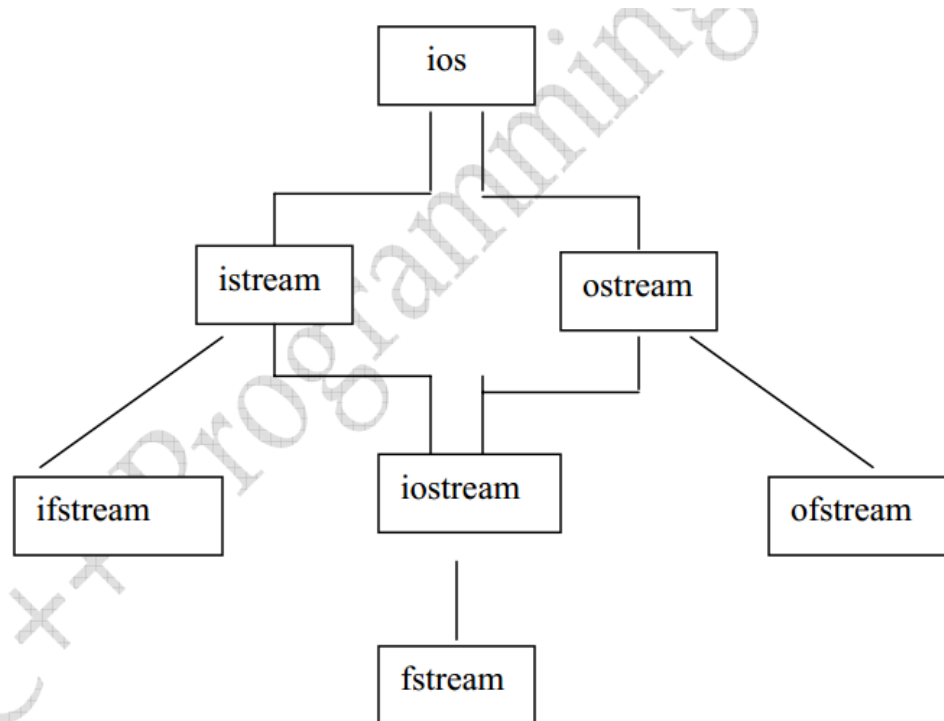
File Operations (File I/O)

Introduction

- The data created by the user and assigned to variables with an assignment statement is sufficient for some applications. With large volume of data most real-world applications use a better way of storing that data. For this, disk files offer the solution.
- When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time.

Stream

- Stream is a general name given to flow of data. In C++, there are different types of streams. Each stream is associated with a particular class, which contains member function and definition for dealing with file.
- Let's have a look at the figure:



- According to the above hierarchy, the class `iostream` is derived from the two classes' `istream` and `ostream` and both `istream` and `ostream` are derived from `ios`. Similarly the class `fstream` is derived from `iostream`.

- Generally, two main header files are used `iostream.h` and `fstream.h`. The classes used for input and output to the video display and key board are declared in the header file `iostream.h` and the classes used for disk file input output are declared in `fstream.h`.
- Note that when we include the header file `fstream.h` in our program then there is no need to include `iostream.h` header file. Because all the classes which are in `fstream.h` they are derived from classes which are in `iostream.h` therefore, we can use all the functions of `iostream` class.

Operation with File

- First we will see how files are opened and closed. A file can be defined by following class `ifstream`, `ofstream`, `fstream`, all these are defined in `fstream.h` header file.
 - if a file object is declared by `ifstream` class, then that object can be used for reading from a file.
 - if a file object is declared by `ofstream` class, then that object can be used for writing onto a file.
 - If a file object is declared by `fstream` class then, that object can be used for both reading from and writing to a file

Types of Disk File Access

- Your program can access files either in sequential manner or random manner. The access mode of a file determines how one can read, write, change, add and delete data from a file.
- A sequential file has to be accessed in the same order as the file was written. This is analogues to cassette tapes: you play music in the same order as it was recorded.
- Unlike the sequential files, you can have random-access to files in any order you want. Think of data in a random-access file as being similar to songs on compact disc (CD): you can go directly to any song you want to play without having to play or fast-forward through the other songs.

Sequential File Concepts

- You can perform three operations on sequential disk files. You can create disk files, add to disk files, and read from disk files.

Opening and Closing Sequential Files

- When you open a disk file, you only have to inform C++, the file name and what you want to do with it. C++ and your operating system work together to make sure that the disk is ready, and they create an entry in your file directory for the filename (if you are creating a file). When you close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.

- You can use either of the two methods to open a file in C++:

- using a **Constructor** or
- using the **open function**

- The following C++ statement will create an object with the name **fout** of ofstream class and this object will be associated with file name "hello.txt".

ofstream fout ("hello.txt");

- This statement uses the constructor method.
- The following C++ statement will create an object with the name **fout** of ofstream class and this object will be associated with file name "hello.txt".

ofstream fout;

fout.open("hello.txt");

- If you open a file for writing (out access mode), C++ creates the file. If a file by that name already exists, C++ overwrite the old file with no warning. You must be careful when opening files not to overwrite existing data that you want.
- If an error occurs during opening of a file, C++ does not create a valid file pointer (file object). Instead, C++ creates a file pointer (object) equal to zero. For example if you open a file for output, but use an invalid disk name, C++ can't open the file and therefore makes the file object equal to zero.

- You can also determine the file access mode when creating a file in C++. If you want to use the open function to open a file then the syntax is: **fileobject.open(filename,accessmode);**
 - File name is a string containing a valid file name for your computer.
 - Access mode is the sought operation to be taken on the file and must be one of the values in the following table.

Mode	Description
app	Opens file for appending
ate	Seeks to the end of file while opening the file
in	Opens the file for reading
out	Opens the file for writing
binary	Opens the file in binary mode

- You should always check for the successful opening of a file before starting file manipulation on it. You use the fail () function to do the task:
- Let's have an example here:

```
ifstream indata;
indata.open("c:\\myfile.txt",ios::in);
if(indata.fail())
{
//error description here
}
```
- In this case, the open operation will fail (i.e the fail function will return true), if there is no file named myfile.txt in the directory C:\
 - After you are done with your file manipulations, you should use the close function to release any resources that were consumed by the file operation.
- Here is an example

```
indata.close();
```
- The above close() statement will terminate the relationship between the ifstream object indata and the file name "c:\\myfile.txt", hence releasing any resource needed by the system.

Writing to a Sequential File

- The most common file I/O functions are `get()` and `put()`
- You can also use the output redirection operator (`<<`) to write to a file.
- The following program creates a file called `names.txt` in `C:\` and saves the name of five persons in it:

```
#include<fstream.h>
#include<stdlib.h>
ofstream fp;
void main()
{
    fp.open("c:\\names.txt",ios::out);
    if(fp.fail())
    {
        cerr<< "\nError opening file";
        getch();
        exit(1);
    }
    fp<< "Abebe Alemu"<<endl;
    fp<< "Lemelem Berhanu"<<endl;
    fp<< "Tsfaye Mulugeta"<<endl;
    fp<< "Mahlet Kebede"<<endl;
    fp<< "Assefa Bogale"<<endl;
    fp.close();
} //end main
```

Writing Characters to Sequential Files:

- A character can be written onto a file using the put () function. See the following code:

```
#include<fstream.h>
#include<stdlib.h>// for exit() function
void main()
{
char c;
ofstream outfile;
outfile.open("c:\\test.txt",ios::out);
if(outfile.fail())
{
cerr<< "\nError opening test.txt";
getch();
exit(1);
}
for(int i=1;i<=15;i++)
{
cout<< "\nEnter a character : ";
cin>>c;
outfile.put(c);
}
outfile.close();
} //end main
```

- The above program reads 15 characters and stores in file test.txt.
- You can easily add data to an existing file, or create new files, by opening the file in append access mode.
- Files you open for append access mode (using ios :: app) do not have to exist.
- If the file exists, C++ appends data to the end of the file (as is done when you open a file for write access).

- The following program adds three more names to the names.txt file created in the earlier program.

```
#include<fstream.h>
#include<stdlib.h>
...
void main()
{
    ofstream outdata;
    outdata.open("c:\\names.txt",ios::app);
    if(outdata.fail())
    {
        cerr<< "\nError opening names.txt";
        getch();
        exit(1);
    }
    outdata<< "Berhanu Teka"<<endl;
    outdata<< "Zelalem Assefa"<<endl;
    outdata<< "Dagim Sheferaw"<<endl;
    outdata.close();
} //end main
```

- If the file names.txt does not exist, C++ creates it and stores the three names to the file.
- Basically, you have to change only the open() function's access mode to turn a file-creation program into a file-appending program.

Reading from a File

- Files you open for read access (using ios::in) must exist already, or C++ gives you an error message. You can't read a file that does not exist. Open() returns zero if the file does not exist when you open it for read access.

- Another event happens when you read files. Eventually, you read all the data. Subsequently reading produces error because there is no more data to read. C++ provides a solution to the end-of-file occurrence.
- If you attempt to read a file that you have completely read the data from, C++ returns the value zero. To find the end-of-file condition, be sure to check for zero when reading information from files.
- The following code asks the user for a file name and displays the content of the file to the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{ char name[20],filename[15];
  ifstream indata;
  cout<<"\nEnter the file name : ";
  cin.getline(filename,15);
  indata.open(filename,ios::in);
  if(indata.fail())
  {
    cerr<<"\nError opening file : "<<filename;
  }
  while(!indata.eof())// checks for the end-of-file
  {
    indata>>name;
    cout<<name<<endl;
  }
  indata.close();
  getch();
}
```

Reading characters from a sequential file

- You can read a characters from a file using **get()** function. The following program asks for a file name and displays each character of the file to the screen. NB. A space among characters is considered as a character and hence, the exact replica of the file will be shown in the screen.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
char c,filename[15];
ifstream indata;
cout<<"\nEnter the file name : ";
cin.getline(filename,15);
indata.open(filename,ios::in);
if(indata.fail())// check id open succeeded
{
cerr<<"\nError opening file : "<<filename;
getch();
exit(1);
}
while(!indata.eof())// check for eof
{
indata.get(c);
cout<<c;
}
indata.close();
getch();
}
```

File Pointer and their Manipulators

- Each file has two pointers one is called input pointer and second is output pointer. The input pointer is called get pointer and the output pointer is called put pointer.
- When input and output operation take places, the appropriate pointer is automatically set according to the access mode.
- For example when we open a file in reading mode, file pointer is automatically set to the start of the file.
- When we open a file in append mode, the file pointer is automatically set to the end of file.
- In C++ there are some manipulators by which we can control the movement of the pointer. The available manipulators are:
 1. `seekg()`
 2. `seekp()`
 3. `tellg()`
 4. `tellp()`
- **seekg():** this moves get pointer i.e. input pointer to a specified location.
For eg. `infile.seekg(5);` move the file pointer to the byte number 5 from starting point.
- **seekp():** this move put pointer (output pointer) to a specified location for example:
`outfile.seekp(5);`
- **tellg():** this gives the current position of get pointer (input pointer)
- **tellp():** this gives the current position of put pointer (output pointer)
eg. `ofstream fileout;`
`fileout.open("c:\\test.txt",ios::app);`
`int length = fileout.tellp();`
- By the above statement in `length`, the total number bytes of the file are assigned to the integer variable `length`. Because the file is opened in append mode that means, the file pointer is the last part of the file.

- Now let's see the seekg() function in action

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
    clrscr();
    fstream fileobj;
    char ch; //holds A through Z
    //open the file in both output and input mode
    fileobj.open("c:\\alph.txt",ios::out|ios::in);
    if(fileobj.fail())
    {
        cerr<<"\nError opening alph.txt";
        getch();
        exit(1);
    }
    //now write the characters to the file
    for(ch = 'A'; ch <= 'Z'; ch++)
    {
        fileobj<<ch;
    }
    fileobj.seekg(8L,ios::beg);//skips eight letters, points to I
    fileobj>>ch;
    cout<<"\nThe 8th character is : "<<ch;
    fileobj.seekg(16L,ios::beg);//skips 16 letters, points to Q
    fileobj>>ch;
    cout<<"\nThe 16th letter is : "<<ch;
    fileobj.close();
    getch();
}
```

- To point to the end of a data file, you can use the `seekg()` function to position the file pointer at the last byte. This statement positions the file pointer to the last byte in the file. **`Fileobj.seekg(0L,ios::end);`**
- This `seekg()` function literally reads “move the file pointer 0 bytes from the end of the file.” The file pointer now points to the end-of-file marker, but you can `seekg()` backwards to find other data in the file.
- The following program is supposed to read “c:\alph.txt” file backwards, printing each character as it skips back in the file.
- Be sure that the `seekg()` in the program seeks two bytes backwards from the current position, not from the beginning or the end as the previous programs. The for loop towards the end of the program needs to perform a “skip-two-bytes-back”, read-one-byte-forward” method to skip through the file backwards.

```
#include<fstream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
clrscr();
ifstream indata;
int ctr=0;
char inchar;
indata.open("c:\\alph.txt",ios::in);
if(indata.fail())
{
cerr<<"\nError opening alph.txt";
getch();
exit(1);
}
```

```
indata.seekg(-1L,ios::end);//points to the last byte in the file
for(ctr=0;ctr<26;ctr++)
{
indata>>inchar;
indata.seekg(-2L,ios::cur);
cout<<inchar;
}
indata.close();
getch();
}
```