C/C++ Tutorial for Computer Graphics

Part II

The tutorial explains C/C++ basics that will be used in Computer Graphics.

Outline

1. **Character Sequences**

A character sequence is used as the string type. Different from JAVA, there is not the string type in C/C++. We can represent a string using a simple array of characters. For example,

```
char foo [10];
```

where foo is an array that contains 10 elements of type char, which can be represented as:

	0	1	2	3	4	5	6	7	8	9
foo										
	char	char	char	char	char	char	char	char	char	char

This array can store a sequence of up to 10 characters, e.g., ”Hello” and ”Hi”, since both would fit in the sequence with a capacity of 10 characters.

By convention, the end of strings represented in character sequences is signaled by a *null character*, whose literal value can be written as ”\0” (backslash, zero). An array of characters is often used to store a string shorter than its total length, and thus we often add a null character at the end of strings to indicate the end of sequences. Therefore, the array *foo* can be represented storing the character sequences "Hello" and "Hi" as:

	0	1	2	3	4	5	6	7	8	9
foo	H	e	l	l	o	\n				
	char	char	char	char	char	char	char	char	char	char

	0	1	2	3	4	5	6	7	8	9
foo	H	i	\n							
	char	char	char	char	char	char	char	char	char	char

The above diagrams show how the array *foo* is represented in memory. The individual characters that make up the string are stored in the elements of the array. The string is terminated by a null character. The panels in grey colour represent char elements with undetermined values.

A "null string" is a string with a null character as its first character:

	0	1	2	3	4	5	6	7	8	9
foo	\n									
	char	char	char	char	char	char	char	char	char	char

- **Initialization of null-terminated character sequences**

Because arrays of characters are also ordinary arrays, we can initialize an array of characters with some predetermined sequence of characters like the initialization of other arrays:

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

Here we declare an array of 6 elements of type *char* initialized with the word ”Hello” and a *null character* '\0' at the end.

In addition, arrays of characters have another way to be initialized: using *string literals*, which are specified by enclosing the text between double quotes ("). For example, ”Hello”.

This type, in fact, means that *string literals* always have a *null character* automatically appended at the end. Therefore, we can initialize the array called *mystring* with either of the two methods:

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

In above two cases, the array of characters *mystring* is declared with 6 characters (the element is type *char*): the 5 characters that make up "Hello" plus a last null character ('\0'). In the second case, when using double quotes (") the null character is appended automatically.

- **Assignment of character sequences**

Only when the array is initialized can you assign multiple values to the array at the same time, that is, when the array is declared, it is legal. The following expressions are all wrong:

```
Mystring = "Hello";                      // wrong
mystring[] = "Hello";                    // wrong
mystring[] = {'H', 'e', 'l', 'l', 'o', '\0'};   // wrong
```

Arrays cannot be assigned values, but each of its elements can be assigned a value individually. As follows:

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

2. Pointers

As we all know, a variable has a memory location which can be accessed by using the identifier (their name) whenever it needs to refer to the variable, thus the program does not need to care about the physical address of the data in memory.

For a C++ program, when a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). If a variable is defined in the program, the system allocates the memory cell to this variable when compiling. The system allocates a certain length of space according to the type of the variable. Each one byte (unit) in memory has a number, which we call memory address.

- **Address-of operator (&)**

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator. For example:

```
foo = &myword;
```

This would store the address of variable *myword* in another memory cell (variable foo), not the value. We can distinguish the value and the address of a variable by the following example.

Let's assume that *myword* is placed when compiling in memory address 216, though the actual address of a variable in memory cannot be known before compiling. Here are three simple sentences:

```
myword = 10;
foo= &myword;
val=myword;
```

- The first sentence means that we assign value 10 to the variable *myword*.
- The second sentence means that we assign the address of *myword* to the variable *foo*. The value of *foo* is the address of *myword*.
- The third sentence means that we assign the value of *myword* to variable *val*.

The main difference between the second and third statements is the appearance of the *address-of operator (&)*.

Dereference operator (*)

In C++, the variable that stores the address of another variable (like *foo*) is called a *pointer*. The pointer of a variable is the address of the variable. The variable used to store the variable address is the pointer variable, which is a special variable. Pointers can be used to access the variable they point to directly by preceding the pointer name with the dereference operator (*). The operator itself can be read as "value pointed to by". For example, *i_pointer* is a pointer variable, and **i_pointer* represents the variable pointed to by *i_pointer*.

The general form of declarations of the pointer variable is:

```
con=*foo;
```

This could be read as: "*con* equal to value pointed to by *foo*". Therefore, according to the three sentences above (assuming the address of *myword* is 216):

```
myword = 10;
foo= &myword;  val=myword;
```

- The statement would actually assign the value 10 to *con*, since *foo* is 216, and the value pointed to by 216 (following the example above) would be 10.
- Notice the difference of including or not including the *dereference operator* ”*”. The *i_pointer* refers to the address, while **i_pointer* refers to the value stored at the address.

The declaration of pointers

Just like any other variable or constant, you must declare a pointer before using the pointer to store other variable address. The declaration of pointers follows this syntax:

```
type *name;
```

where *type* is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int *ID; char  *name;
```

These are two declarations of pointers. Each one is intended to point to a different data type. Therefore, although these two variables are pointers, they actually have different types: int* and char* respectively, depending on the type they point to.

When declaring a pointer, the ”*” only represents this is a *pointer* type, which is different from the earlier *dereference operator (*)*.

Pointing a pointer variable to another variable:

```
pointer_1 = &i;  / / pointer_1 stores the address of variable i
```

In this way, *pointer_1* points to the variable *i*. If we assign 10 to the variable *i*, the value of *pointer_1 is 10.

Here is an example:

```
/ my first pointer
#include <iostream>
using namespace std;

int main()
{
     int value1, value2;
     int * mypointer;
     mypointer = &value1;
     *mypointer = 10;
     mypointer = &value2;
     value2 = 20;

     cout << "value1 is " << value1 << '\n';
     cout << "*mypointer is " << *mypointer << '\n';
     return 0;
}
```

In order to show that a pointer may point to different variables during its lifetime in a program, the example repeats the process with *value2* and that same pointer, *mypointer*.

If a pointer needs to explicitly point to nowhere and not just an invalid address, there exists a special value that any pointer type can take: the null pointer value. In C++, this value can be expressed in two ways: either with an integer value of zero, or with the nullptr keyword:

```
int *p = 0;
int *q = nullptr;
```

Here, p and q are *null pointers*. All *null pointers* compare equal to other *null pointers.* It is also quite usual to see the defined constant *NULL* be used in older code to refer to the *null pointer* value:

```
int *ptr = NULL;
```

NULL is defined in several headers of the standard library, and is defined as an alias of some *null pointer* constant value (such as 0 or *nullptr*).

Notice there are some differences between a *null pointer* and a *void pointer*. A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere", while a *void pointer* is a type of pointer that can point to somewhere without a specific type.

- **Pointer initialization**

When pointers are defined, they can be initialized to point to specific locations:

```
int myword;
int *myptr = &myword;
```

This is the same as the assignment after declaring:

```
int myword;
int *myptr;
myptr = &myword;
```

Notice *myptr* points to the address of a variable, while **myptr* represents the variable.

As pointed earlier section, the asterisk (*) in the pointer declaration only indicates that it is a pointer, and it is not the *dereference operator*.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

int myword;
int *foo = &myword; int *myptr = foo;

Pointers and arrays

Arrays behave very much like pointers to their first elements. For example, let us see an example:

int myarray[10];
int *mypointer;

The following sentence would be valid:

mypointer = myarray;

Here, *mypointer* points to the first element of *myarray*, myarray[0].

The main difference between pointers and arrays is that pointers can point to a different address, while arrays will always represent the same block (like 10 elements) of a type (like *int*). So, the following assignment would not be valid:

myarray = mypointer;

For example:

```
// my pointers
#include <iostream>
using namespace std;

int main()
{
    int numbers[5];
    int * p;
    p = numbers;  *p = 10; // p points to numbers[0]
    p++;  *p = 20;  // p  points to numbers[1] and 20 be assigned to it
    p = &numbers[2];  *p = 30; // p points to numbers[2] and 30 be assigned to it
    p = numbers + 3;  *p = 40; // p points to numbers[3] and 40 be assigned to it
    p = numbers; *(p + 4) = 50; // p points to numbers[4] and 50 be assigned to it

    for (int n = 0; n<5; n++)
        cout << numbers[n] << ", ";
    return 0;
}
```

The brackets ([]) are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:

arr[8] = 5;
*(arr+8) = 5;

The two sentences are equivalent. Notice we can use the name of an array just like a pointer point to its first element.

Pointer arithmetics

Different data types have different sizes. For example, *char* has a size of 1 byte, *short* is generally larger than that, and *int* and *long* are even larger. The exact size of them depends on the system. For example, assuming that in a given system, *char* takes 1 byte, *short* takes 2 bytes, and *long* takes 4.

There are multiple arithmetic operations that can be applied on C++ pointers: ++, --, -, +. They have a slightly different behavior applied to pointers, according to the size of the data type to which pointers point. And now we declare three pointers:

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

Assuming that they point to the memory locations 300, 500, and 800, respectively.

Just like any variable the ”++” operation increases the value of that variable.In our case here the variable is a pointer hence when we increase its value we are increasing the address in the memory that pointer points to. Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar would point to the address 301. *myshort* and *mylong* would point to 502, 804, respectively. That depends on the data type they point to. We can also write the above sentences as follows:

```
mychar = mychar+1;
myshort = myshort+1;
mylong = mylong+1;
```

For the ”++” and ”--” operators, they have a slight difference in behavior as a prefix and a suffix. As a prefix, the increment happens before the expression is evaluated, and as a suffix, the increment happens after the expression is evaluated. This also applies to ”++” and ”--” of pointers. We can also use them together with dereference operators (*). According to operator precedence rules, increment and decrement have higher precedence than prefix operators, such as ”*”. Therefore, **p++* is equivalent to **(p++)*.

There are four possible combinations of the dereference operator with both the prefix and suffix versions of the increment operator (the same being applicable also to the decrement operator):

```
*p++   // same as *(p++):increment pointer, and dereference incremented address
*++p   // same as *(++p):increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

There is an important example:

```
*p++=*q++;
```

Because ”++” has a higher precedence than *, both *p* and *q* are incremented. But because both increment operators (++) are used as suffix, the value of **q* is assigned to **p* before both *p* and *q* are incremented. And then both are incremented. Actually, it is:

```
*p = *q;
++p;
++q;
```

- **Pointers and const**

When a pointer points to a variable, it can be used to modify the value of the variable. Sometimes, we just need to read the value, not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as *const*. For example:

```
int a = 1, b;
const int *p = &a;
b = *p;   // valid, reading p
*p = b;   // invalid, *p is const-qualified
```

Here, we define *p* points to a variable in a const-qualified manner, which means that it can read the value of the variable, not modifying it.

For a function, if taking a pointer to *non-const* as a parameter, it can modify the value passed as argument, while taking a pointer to *const* as parameter cannot. For example:

```
void swap(int *a, int *b)
{
    int m   = *a;
    *a = *b;
    *b = m;
}
void swap (const int *a, const int *b)
{
    int m,n;  m = *a+1;  n=*b+2;
}
```

For the first example, pointers (a and b) are used to as a *non-const* parameter, which means *a and *b can be modified. While *a and *b for the second example cannot be modified, just reading it.

Pointers can also be themselves *const*. And this is specified by appending *const* to the pointed type (after the asterisk):

```
      int *         p1 = &x;   // non-const pointer to non-const int
const int *         p2 = &x;   // non-const pointer to const int
      int * const p3 = &x;   // const pointer to non-const int
const int * const p4 = &x;   // const pointer to const int
```

The *const* qualifier can either precede or follow the pointed type, with the exact same meaning:

```
const int * pa = &x;   // non-const pointer to const int
int const * pb = &x;   // also non-const pointer to const int
```

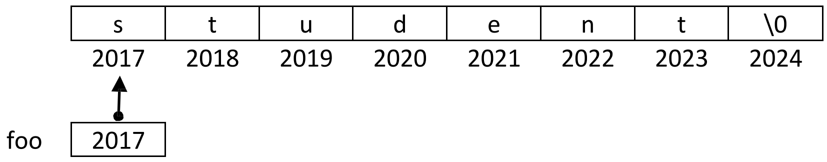
Both are exactly equivalent.

• Pointers and string literals

As earlier sections introduced, *string literals* are arrays of character sequences ending with null-terminated character, which can be accessed directly. Each element of string literals is of type *const char* (as literals, they can never be modified). For example:

```
const char *foo = ”student”;
```

This declares an array of character (”student”), and a pointer points to the first element of *foo*. We assume ”student” is stored at the memory locations that start at address 2017, and the previous declaration as follows:



Here, the value of the pointer *foo* is 2017, and not ’s’, nor ”student”.

The way that pointer access the characters is like the way arrays of null-terminated character sequences are. For example:

```
*(foo+2)  foo[2]
```

Both values of expressions are ’u’, the third element of ”student”.

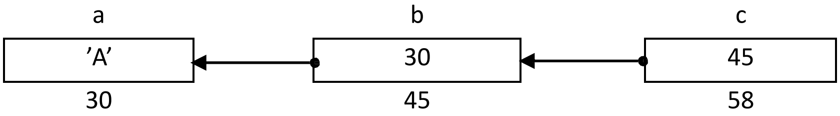
• Pointers to pointers

For a C++ program, because a pointer itself is a special variable, a pointer not only can point to a variable, but also another pointer.

A pointer to a pointer is a form of multilevel indirect addressing, or a pointer chain. When we define a pointer to a pointer, the first pointer contains the address of second pointers, and the second pointer points to the location that contains the actual value. For example:

```
char a;
char *b;
char**c;
a = ’A’;
b = &a;
c = &b;
```

We assume the address of variable *a, b* and *c* are 30, 45 and 58 respectively, as follows:



The content inside its corresponding cell is the value of each variable. The value under them is the address of variable. According to above sentences, we can notice the variable *c*, which is a pointer to a pointer.

3. Functions

A function is a set of statements that perform a task together. Each program in C++ has at least one function, that is, the main function *main()*, and all simple programs can define other additional functions. The most common syntax to define a function is:

```
type name (parameter1, parameter2...) {statements}
```

where:

- *type* is the data type of the value returned by the function.
- *name* is the identifier by which the function can be called, and this is the actual name of the function.
- *parameters* (as many as needed): Each parameter looks like a regular variable declaration (like int x), and in fact acts within the function as a regular variable which is local to the function. Each parameter is separated from the next by a comma. The purpose of parameters is to allow passing arguments to the function from the location where it is called from. The parameter is optional, that is, the function may not contain parameters.
- *statements* is the function's body. It contains a set of statements that define the function to perform the task, which is surrounded by braces { } that specify what the function actually does.

For example

```
// my function
#include <iostream>
using namespace std;

int add(int a, int b)
{
     int s;
     s = a + b;
     return s;
}

int main()
{
     int sum;
     sum = add(8, 6);
     cout << "The result is " << sum << "\n";
}
```

There are two functions in this program, *add()* and *main()*. Notice in C++, a program always starts by calling *main*(), no matter the order in which they are defined. In fact, *main* function is the only function called automatically.

For the above simple program, *main* function begins by declaring the variable *sum* of type int, and then calls *add* function. We can compare the calling to its definition:

int add(int a, int b)
↑ ↑
sum = add(8, 6);

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call, where 8 and 6 correspond to the parameters *a* and *b*.

When the program calls the *add* function in main function, execution of main is stopped, and will only resume once the *add* function ends. It passes the arguments (8,6) to variables (a and b) within *add* function. Another local variable (*s*) is defined in the *add* function. Then there is an expression *s=a+b*, where *a* is 8 and *b* is 6, means the return value of *add* function is 14. This return value would be given to *sum* in *main()*.

int add(int a, int b)
14 ↓
sum = add(8, 6);

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```
// my function
#include <iostream>
using namespace std;

int add(int a, int b)
{
    int s;
    s = a + b;
    return s;
}

int main()
{
    int sum1, sum2;
    sum1 = add(8, 6);
    sum2 = 5 + add(2, 8);
    cout << "The sum1 is " << sum1 << "\n";
    cout << "The sum2 is " << sum2 << "\n";
}
```

- ## Functions with no type - the use of void

Generally, functions always have a return value, but sometimes a function does not need to return a value. In this cases, the type to be used is *void,* which is a special type to represent the absence of value. For example, a function just prints a message:

```
void pmassage()
{
        cout << "This is a massage!";
}
```

Because there is no parameter in this function, it can be written:

```
void pmassage (void)
{
        cout << "This is a massage!";
}
```

The parentheses () that follow the function name are not optional, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses are necessary. For example:

```
pmassage ();
```

- ## The return value of main

In earlier examples, you may be wondering why the return type of *main()* is *int*, but most examples do not actually return any value from *main().*

In C + +, the *main* function is called by the operating system. And the program terminates after the *main* function has been executed. Generally speaking, when *main* function returns 0, there is no error in the running of the program, and other non-zero values may indicate that the program is out of order.

- ## Arguments passed by value and by reference

In the earlier sections, arguments have always been passed *by value*. This means that the actual value of the parameter is copied into the variables represented by the function parameters. In this case, modifying the formal parameters within the function does not affect the actual parameters. For example:

```
int m = 8, n = 6;
exchange( m, n );
void exchange(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
        return;
}
```

Here, function *exchange* is passed 8 and 6, which are copies of the values of *m* and *n*, respectively. Any modification of these variables within the function *add* has no effect on the values of the variables *m* and *n* outside it. Therefore, the result is *m=8, n=6*.

In certain cases, though, it may be useful to access an external variable from within a function. For these cases, arguments can be passed *by reference*. For example:

```
int m = 8, n = 6;
exchange( m, n );
void exchange(int & a, int & b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
        return;
}
```

The function *exchange* in this code exchanges the value of its two arguments, causing the variables used as arguments to actually be modified by the call. That is because, when a variable is passed *by reference*, what is passed is not copy of the values, but the variable itself, and any modification on their corresponding local variables within the function are reflected in the variables passed as arguments in the call. Therefore, the result is *m=6, n=8*.

- **Efficiency considerations and const references**

It may cause copies of the values to be made to call a function with parameters taken by value. If the parameter is of a large compound type, it may result on certain overhead. For example:

```
string add (string a, string b)
{
        return a + b;
}
```

Here, this function takes two strings as parameters (by value), and returns the result of concatenating them. The function copies *a* and *b* to pass to the function when it is called by passing the arguments by value.

But this copy can be avoided altogether if both parameters are made *references*:

```
string add (string& a, string& b)
{
        return a + b;
}
```

Arguments by reference do not require a copy. The function operates directly on the strings passed as arguments, and, at most, it might mean the transfer of certain pointers to the function. Therefore, the second method taking references is more efficient than the first one taking values, since it does not need to copy expensive-to-copy strings. On the other hand, functions with reference parameters can modify the arguments passed, as pointed earlier.

If we want to guarantee that its reference parameters are not going to be modified by this function, we can qualify the parameters as constant:

```
string add (const string& a, const string& b)
{
        return a + b;
}
```

The function is forbidden to modify the values of *a* and *b*, and can access the values without having to make actual copies of the strings.

• Inline functions

As we all know, it would cause a certain overhead (stacking arguments, jumps, etc...) to call a function generally, and thus it may be more efficient to simply insert the code of the very short function where it is called, instead of performing the process of formally calling a function.

A program in C++ supports inline function, and its purpose is to improve the efficiency of the function. The keyword *inline* is placed in front of the function definition to specify the function as inline function. We define an inline function:

```
inline int add (int a, int b)
{
        return a + b;
}
cout << add(a, b) << endl;
```

This informs the compiler that when *add* is called, the program prefers the function to be expanded inline, instead of performing a regular call. The *inline* is only specified in the function definition, not when it is called.

Notice most compilers already optimize code to generate inline functions when they have an opportunity to improve efficiency, even if not explicitly marked with the *inline* specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

• Default values in parameters

In C++, we can specify a default value for each parameter when defining a function. When the function is called, if the value of the parameter is not passed, the default value is used. If the value of the parameter is specified, the default value is ignored, and the passed value is used. For example:

```
#include <iostream>
using namespace std;

int add(int a, int b = 2)
{
    int s;
    s = a + b;
    return s;
}

int main()
{
    cout << add(5) << '\n';
    cout << add(12,6) << '\n';
    return 0;
}
```

In the above example, there are two calls to function add.

The first call, *add(5)*, only passes one argument to the function, even though the function has two parameters. In this case, the function uses the default value 2. Therefore, the result is 7.

The second call, *add (12,6)*, passes two arguments to the function. Therefore, the default value 2 of *b* is ignored, and it uses the value 6. The result is 18.

- **Declaring functions**

In C++, identifiers can only be used in expressions once they have been declared, such as *int a*. The same applies to functions. The functions can be called only when it is declared.

The declaration of functions shall include all types involved (the return type and the type of its arguments), which replaces the body of the function with an ending semicolon. For example:

int max(int a, int b);

In the declaration of a function, the name of the parameter is not important, but the type of the parameter is required, so the following declaration is also valid:

int max(int, int);

The function declaration is required when you define a function in a source file and call the function in another file. When you define a function and the main function in the same file, you should declare the function at the top of the file if you define the function behind *main()*, or you can define the function before *main()* and avoid the declaration. Let us see two examples:

```
#include <iostream>
using namespace std;

int add(int a, int b);  // declaration of function

int main()
{
      int a = 100, b = 200, sum;  // declaration of local  variables
      sum = add(a, b);  // call function
      cout << "sum is : " << sum << endl;
      return 0;
}

int add(int a, int b)
{
      int s; // declaration of local variables
      s = a + b;
      return s;
}
```

```
#include <iostream>
using namespace std;

int add(int a, int b) // definition of function add
{
     int s; // declaration of local variables
     s = a + b;
     return s;
}

int main()
{
     int a = 100, b = 200, sum;  // declaration of local  variables
     sum = add(a, b);  // call function
     cout << "sum is : " << sum << endl;
     return 0;
}
```

The above two examples will result the same program.

The declaration of *add* function already contain all what is necessary to call them, its name, the types of its arguments, and the return type (*int* in this case). With the prototype declaration, it can be called before it is entirely defined.

- **Recursive function**

Recursive function means call the function itself, that is, call itself directly or indirectly in the function body. It is useful for some tasks, such as calculating the factorial of numbers (like *n!*). For example, a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
     if (a > 1)
          return (a * factorial (a-1));
     else
          return 1;
}

int main ()
{
     long number = 6;
     cout << number << "! = " << factorial (number);
     return 0;
}
```

Notice that the function factorial would call itself only if the argument passed was greater than 1. If there is no restrictive condition, the function would perform an infinite recursive loop.

4. **Input/output with Files**

The standard library in C++ provides a set of input / output functions, which includes the following classes to perform output and input of characters to/from files:

- *ofstream*: Stream class to write on files (derived from *ostream*)
- *ifstream*: Stream class to read from files (derived from *istream*)
- *fstream*: Stream class to both read and write from/to files (derived from *iostream*)

These classes are derived directly or indirectly from the classes *istream* and *ostream*. In the previous examples, we have already used the objects *cin* and *cout*, which are related to our file streams.

• Open a file

In the *fstream* class, we use the member function *open ()* to implement the operation of opening a file with a stream object:

```
open(filename, mode);
```

where *filename* is the name of a file to be opened, which can be a character pointer, an array of characters, or a string type, and mode is an optional parameter used to decide how to open the file. They are as follows:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

We can use one or more of them at a time by using the bitwise operator OR (|). For example,

```
ofstream myfile;
myfile.open ( ”test.bin”, ios::out | ios::app | ios::binary);
```

The two sentences mean we can open a file called *test.bin* in binary mode to add data. Sometimes, many programs call *open()* function in default mode. For example:

class	**default mode parameter**
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For *ofstream* and *ifstream*, the default mode is automatically assumed. If a mode that is different from the default value is passed as second argument to the open member function, the default mode is used in conjunction with the passed parameter.

For *fstream*, the default value is used only if the function is called without a declared parameter. If the function is called with parameters, the default value will be fully rewritten, not combined.

Since the first operation that is performed on a file stream is usually to open a file, these classes (*ofstream, ifstream,* and *fstream*) have a constructor that calls the *open* function automatically and has the same parameters as this member. In this way, we could also have declared the previous *myfile* object and conduct the same opening operation in our previous example by writing:

ofstream myfile (”test.bin”, ios::out | ios::app | ios::binary); This sentence acts as above two sentences.

You can call the member function *is_open ()* to check if a file has been successfully opened. bool is_open ();

This member function returns a *bool* value of *true* in the case that indeed the stream object is associated with an open file, or *false* otherwise:

if(myfile.is_open()) { /* statements */ }

- **Close a file**

When our input and output operations on a file have been completed, we shall close the file to make it to be available again. The member function *close()* is responsible for discharging the data from the buffer and closing the file.

myfile.close();

Once this member function is called, the stream object can be used to open another file, and the file also can be accessed again by the other process. To prevent the stream object still associated with the open file from being destroyed, the destructor would automatically calls the member function *close()*.

- **Text files**

Text file streams are designed to store text and the *ios::binary* flag is not included in their opening mode. The read and write on text files is very simple: use the inserter (< <) to output to the file, and use the extractor (> >) to input from the file. For example, writing operations on text files are performed in the same way we operated with *cout*:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream myfile("test.txt"); // open a file called test.txt

    if(myfile.is_open()) // check if the file has been successfully opened
    {
          Myfile << "This is a line.\n"; // writing ”This is a line” on the txt file
          myfile.close(); // close the file
    }
    else cout << "Unable to open file"; return 0;
}
```

The result is that writing ”*This is a line*” on a txt file called *test*.

Reading from a file can also be performed in the same way that we did with *cin*:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
     string line;
     ifstream myfile("test.txt");
     if(myfile.is_open())
     {
            while(getline(myfile,line))
            {
                   cout << line << '\n';
            }
            myfile.close();
     }
     else cout << "Unable to open file"; return 0;
}
```

This program reads the content from a txt file called *test*, then prints it on the screen.

Notice we have created a while loop that reads the file line by line, using *getline()*. The value returned by getline() is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

Checking state flags

There are some member functions that validate the state of the stream flow (all of them return a bool value), as follows:

- *bad()*:It will return *true* if there are errors during the reading or writing operation. For example, we try to write to a file that is not open for writing.
- *fail()*: It is same as *bad()*.Besides, it will also return *true* if there is a format error. For example, an alphabetical character is extracted when we are trying to read an integer number.
- *eof()*:It returns *true* if a file open for reading has reached the end.
- *good()*:It returns *false* in the same cases in which calling any of the previous functions would return *true*. Note that *good* and *bad* are not exact opposites (*good* checks more state flags at once).

The member function *clear()* can be used to reset the state flags, without parameters.

get and put stream positioning

All input / output(I/O) streams keep at least internally one internal position:

ifstream, keeps an internal *get position* with the address of the element to be read in the next input operation, like *istream*.

ofstream, keeps an internal *put position* with the location where the next element has to be written, like *ostream*.

fstream, keeps both, the *get* and the *put position*, like *iostream*.

These internal stream positions can be observed and modified using the following member functions:

tellg() and tellp()

These two member functions do not need parameters and return a value of the member type *streampos*, which represents the current get position (in the case of tellg) or the put position (in the case of tellp). **seekg() and seekp()**.

These functions are used to change the location of the get and put positions. Both of them are overloaded with two different prototypes. The first form is:

seekg (pos_type position); seekp (pos_type position);

The stream pointer is changed to the absolute location *position* (counting from the beginning of the file) by using this prototype. The type of this parameter is *streampos*.

These functions can be written in other way:

seekg (off_type offset, seekdir direction); seekp (off_type offset, seekdir direction);

Using this prototype, the *get* or *put position* is set to an *offset* value that starts with a specific pointer determined by the parameter *direction*, where *offset* is of type *streamoff*, and *direction* is of type *seekdir*, which is an *enumerated type* that determines the point from where *offset* is counted from, that could be:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position
ios::end	offset counted from the end of the stream

Here is an example that uses the member functions above to obtain the size of a file, as follows:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main() {
     streampos begin,end; //declare position pointer
     ifstream myfile ("test.bin", ios::binary);
     begin = myfile.tellg(); //get the current position of pointer
     myfile.seekg (0, ios::end); //move the pointer to the end of file
     end = myfile.tellg(); //get the current position(end of file) of pointer
     myfile.close();
     cout << "size is: " << (end-begin) << " bytes.\n";
     return 0;
}
```

Notice *streampos* is a specific type used for buffer and file positioning and is the type returned by *file.tellg()*. Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.

- **Binary files**

For binary files, the way to open and close files is different from text files. We cannot read and write data by using the extraction and insertion operators (<<and >>) and functions like *getline*, but open the files by adding *ios: binary* to read and write files in the binary mode.

There are two member functions included in file streams to read and write binary data sequentially: *write()* and *read()*. The first one (*write*) is a member function of *ostream*, while *read* is a member function of *istream*. Objects of class *fstream* have both. Their prototypes are:

write(memory_block, size); read(memory_block, size);

where *memory_block* represents a memory address used to store or read data, which is of type char*. The parameter *size* is an integer value that represents the number of characters to be read or written from/to the *memory_block*. *read ()* reads some characters (depends on *size*) from the file to the buffer, while *write ()* writes some characters(depends on *size*) to the file from the buffer.

As the following example shows:

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    streampos size; // declare a position pointer
    char * m_block;

    ifstream myfile ("test.bin", ios::in|ios::binary|ios::ate);
    if(myfile.is_open())
    {
        size = file.tellg();
        m_block  = new char [size];
        myfile.seekg (0, ios::beg);
        myfile.read (m_block, size);
        myfile.close();

        cout << "the entire file content is in memory";
        delete[] m_block;
    }
    else cout << "Unable to open file"; return 0;
}
```

First, the program opens a file in the *ios::ate* mode, which means that the get pointer will be positioned at the end of the file. By this, we can directly get the size of the file by calling the member *tellg()*.

Then, we request the allocation of a memory block large enough to hold the entire file:

m_block = new char[size];

After that, we proceed to set the get position at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

file.seekg(0, ios::beg); file.read(memblock, size); file.close();

The content of the file is in memory and then finishes.

• Buffers and Synchronization

The file streams when they are operated are associated to an internal buffer object of type *streambuf*. This buffer object may represent a memory block, which acts as an intermediary between the stream and the physical file. For example, when the program calls the member function *put* (which writes a single character), the character may be inserted in this intermediate buffer instead of being written directly to the physical file.

When the buffer is flushed, all the data contained in it are written to the physical medium (if it is an output stream). This process is called *synchronization* and takes place under any of the following circumstances:

- **When the file is closed:** Before closing a file, all buffers that have not yet been flushed are synchronized and all pending data are written or read to the physical medium.
- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically

synchronized.

- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- **Explicitly, with member function sync():** when the program calls the stream's member function sync(), synchronization would take place immediately. This function returns an int value, and it would return -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.