P

# Cadenas en C++

Pablo R. Ramis

```cpp
#include <iostream>
int main()
{
    std::cout << "In Code We Trust";
    return 0;
}
```

## 1. INTRODUCCIÓN

Sabemos que una cadena es una secuencia de caracteres, en este apartado veremos la clase **string** de la biblioteca estándar, la cual proporciona para poder manipularlos: asignar, concatenar, busqueda de subcadenas, etc.

C++ heredó de C la noción de la cadena como un array de caracteres terminados en cero.

### 1.1. string

En primera instancia veremos el manejo básico del objeto **string** luego, profundizaremos en las clases y su naturaleza.

#### *1.1.1. Construcción.*

```
1
2  void f(char* p, vector<char> v)
3  {
4          string s0;          // la string esta vacia
5          string s00 = "";    // la string esta vacia
6
7          string s1 = 'a';    // error!!!!  no se puede convertir un caracter en string
8          string s2 = 7;      // error!!!! no hay conversion de int a string
9          string s3(7);       // error!!! no hay constructor que tome un int
10
11         string s4(7, 'a');  // crea un string con 7 copias de a, o sea, "aaaaaaa"
12         string s5("Frodo"); // copia de "Frodo"
13         string s6 = s5;     // copia de s5
14
15         string s7(s5, 3, 2);//s5[3] y s5[4] o sea copia "do"
16         string s8(p+7, 3);  //p[7], p[8] y p[9]
17         string s9(p, 7, 3); //string(string(p), 7, 3) no es recomendable
18         string s10(v.begin(), v.end()); //copiar todos los caracteres de v
19
20 }
```

Al igual que C, los caracteres se cuentan a partir de 0, y la cadena tendrá entonces de 0 a **lengh()** $-1$.

**lengh()** es sinónima a **size()** dan el largo de la cadena sin contar el terminador.

En general, la manipulación mas común como crear, copiar, comparar, no generan mayor problemas o errores, pero en el trabajo con **substrings** o de accesos determinados $at()$ es factible que surjan errores, $C++$ comprueba y lanza $out\_of\_range()$ si intentamos acceder mas alla de la cadena. Ejemplos

```
1
2  void f()
3  {
4      string s = "Snobol4";
5      string s2(s, 100, 2);     //posicion del caracter áms alla del final
6                                //de la cadena: lanza error out_of_range()
7      string s3(s, 2, 100);     //Recuento de caracteres demasiado grande:
8                                //equivalente a s3(s, 2, s.size()-2)
9      string s4(s, 2, string::npos); //los caracteres a partir de s[2]
10
11 }
```

Como se ve, se no hay que indicar posiciones demasiado grandes, pero si es posible los recuentos grandes. el ejemplo es válido para el uso de números negativos, ejemplo:

```
1  void g(string s)
2  {
3      string s5(s, -2, 3);     //posicion grande!! lanza out_of_range();
4
5      string s6(s, 3, -2);     //recuento de caracteres grande! bien!
6
7  }
```

El constructor de **string** cuando recibe el valor de posición o recuento, este es de tipo $size\_type$ el cual es de tipo **unsigned** y la conversión de esto es que sera un positivo muy grande.

```
1  #include<string>
```

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

int main(){

    string::size_type st1 = 5;
    string::size_type st2 = -1;

    cout << " st1 tiene un valor de: " << st1 << " y st2 de: " << st2 << endl;
}
```

```
$ g++ -o prueba_size size\_type.cpp
$ ./prueba_size
st1 tiene un valor de: 5 y st2 de: 18446744073709551615
$
$
```

*1.1.2. Asignaciones.* La asignaciones para cadenas funcionan de la siguiente forma:

```cpp
void g()
{
    string s1 = "Knould";
    string s2 = "Tod";

    s1 = s2;        //dos copias de "Tod"
    s2[1] = 'u';    //s2 es "Tud"

}
```

También está soportada la asignación de un simple caracter, pero no en el constructor. Ejemplo:

```cpp
void f()
{
    string s = 'a';    //Error!!!!
    s = 'a';           //Esto es correcto
    s = "a";           //Esto es correcto.

}
```

Si bien el segundo ejemplo no parece muy útil, no es tan extraño anexar un caracter: $s+ =' a'$.

*1.1.3. Comparaciones.* Las cadenas pueden compararse con cadenas de su mismo tipo y con arrays de caracteres.

La función mas común es **compare()**, la cual correspondería usarse de la siguiente forma **s.compare(s2)** retornando 0 en caso que sean iguales, un número negativo si *s* está lexicográficamente antes que *s2* y un número positivo si es el caso contrario.

```cpp
#include <string>
#include <iostream>

using namespace std;

int main ()
{
    string s = "abcde", s2 = "abcdf", s3 = "abc";

    cout << "usamos compare..." << endl;
    if (!s.compare(s2)) // como retorna 0 si son iguales, usamos el !
        cout << "son iguales" << endl;
    else
        cout << "son distintas" << endl;

    cout << "usamos == ..." << endl;
    if (s == s2)
        cout << "son iguales" << endl;
    else
```

```
20          cout << "son distintas" << endl;
21      cout << "usamos compare para comparar los 3 primeros\n"
22                  << "caracteres con s2" << endl;
23      if (s.compare(0,3,s2) == 0)
24          cout << "son iguales" << endl;
25      else
26          cout << "son diferentes\n";
27      cout << "usamos compare para comparar los 3 primeros\n"
28                  << "caracteres con s3" << endl;
29      if (s.compare(0,3,s3) == 0)
30          cout << "son iguales" << endl;
31      else
32          cout << "son diferentes\n";
33
34
35      return 0;
36  }
```

Como vemos, $compare()$ también toma como argumento $size\_type$ **pos** y $size\_type$ **n**, este último es la cantidad de posiciones que se compararán, y el primer argumento desde que posición comenzará. Como vemos en el ejemplo anterior, se compara 3 caracteres de s con la totalidad de s2.

```
$ ./cadenas_comparacion
usamos compare...
son distintas
usamos == ...
son distintas
usamos compare para comparar los 3 primeros
 caracteres con s2
son diferentes
usamos compare para comparar los 3 primeros
 caracteres con s3
son iguales
$
```

Por una cuestión de brevedad, omitiremos los ejemplos de uso con arryas o $char*$, pero son de uso similar, solo hay que tener en cuenta que se use correctamente el terminador de las cadenas.

Hay que tener en cuenta que la función en cuestión distingue las mayúsculas de las minúsculas. Para evitar esto tenemos un par de opciones, armar un método propio utilizando **toupper()** o utilizar $lexicographical\_compare()$

```
1
2   int cmp_nocase(string s, string s2)
3   {
4       string::const_iterator p = s.begin();
5       string::const_iterator p2 = s2.begin();
6
7       while(p != s.end() && p2 != s2.end()){
8           if (toupper(*p) != toupper(*p2) )
9                   return (toupper(*p) < toupper(*p2)) ? -1 : 1;
10
11          ++p;
12          ++p2;
13      }
14
15      return (s2.size() == s.size()) ? 0 : (s.size() < s2.size()) ? -1 : 1;
16  }
```

$lexicographical\_compare()$ tiene dos prototipos, el primero tomando cuatro parámetros, inicio y fin de los dos strings a comparar, y el segundo con cinco parámetros: los cuatro anteriores y una función binaria que acepta dos argumentos de tipo iteradores y retorna verdadero o falso, la comparación es realizada usando al operador $<$ .Ejemplo de uso:

```
1
2   // lexicographical_compare
3
4   #include <iostream>     // std::cout
5   #include <algorithm>    // std::lexicographical_compare
6   #include <cctype>       // std::tolower
7
```

```
using namespace std;


bool mi_comparador (char c1, char c2)
{
    return tolower(c1) < tolower(c2);
}

int main () {
    string s = "Apple";
    string s2 = "apartment";
    string::iterator i = s.begin();
    string::iterator e = s.end();
    string::iterator i2 = s2.begin();
    string::iterator e2 = s2.end();

    cout << boolalpha; // setea la salida en pantalla para mostrar true o false;

    cout << "Comparando s y s2 lexicographicamente (s < s2):\n";

    cout << "usando la ócomparacin por default (operador <): ";
    cout << lexicographical_compare(i,e,i2,e2) << endl;


    cout << "Usando la funcion mi_comparador: ";
    cout << std::lexicographical_compare(i,i+5,i2,i2+9, mi_comparador) << endl;

    return 0;
}
```

```
$  g++ lexicographical.cpp -o lexicographical
$
$ ./lexicographical
Comparando s y s2 lexicographicamente (s < s2):
usando la ócomparacin por default (operador <): true
Usando la funcion mi_comparador: false
$
```