Programación en C++ (12): strings

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMATICÁS

Octubre 2009

Outline

Outline

La librería estándar C++

La librería estándar provee una colección impresionante de estructuras, algoritmos para resolver problemas clásicos de programación:

- strings, para la manipulación de cadenas de caracteres,
- patrones de contenedores implementando datos abstractos de datos,
- patrones de algoritmos,
- flujos I/O iostreams.

Es importante preguntarse, al momento de implementar un algoritmo/contenedor que te parece "clásico" si por casualidad no estaría implementado ya en la libstdc++.

La librería estándar C++

No se puede mencionar en una clase todos métodos, funciones de la libstdc++, usar las documentaciones de esa para mas detalles:
`http://www.cplusplus.com/reference/`
`http://gcc.gnu.org/libstdc++/`
`http://msdn2.microsoft.com/en-us/library/cscc687y(VS.80).aspx`

Cadenas de caracteres en C

Ya vimos que las cadenas de caracteres pueden ser fuentes de errores, en particular por causa de consideraciones de memoria: las cadenas de caracteres en C son arreglos de datos, terminadas por un carácter nulo,

- problemas con el apuntador (acceso fuera de los limites, después de su liberación...),
- problemas con el carácter nulo.

El string encapsula los datos en un objeto cuyo uso es mucho mas fácil y que provoca a priori menos errores.

strings

El string contiene información de tipo "Smart Pointer" :

- dirección en memoria del principio de la cadena,
- tamaño de la cadena,
- ...

y conllevan los métodos necesarios para manejar la evolución de la memoria, el acceso a los caracteres individuales, las principales operaciones de edición de cadenas.

strings: implementación

Una cosa importante es que la implementación de los strings, y en particular de sus mecanismos de alocación/realocación son dependientes del compilador: el lenguaje sólo especifica un comportamiento de los objetos string para con el usuario.

Por ejemplo, varias implementaciones usan cuento de referencias para evitar duplicar versiones del string, pero eso no se ve a través de los string :

```
string a = "Cara de bala";
string b = a;
a[9] = 'o';
cout << a << endl;
cout << b << endl;
```

strings: uso

Se usan con

```
#include <string>
```

y se instancian usando la clase string

```
string s;
```

De hecho, string es una clase un poco particular porque es una especialización de un patrón:

```
typedef basic_string<char> string;
```

donde

```
template<class charT,
class traits = char_traits<charT>,
class allocator = allocator<charT> >
class basic_string;
```

strings: uso

Este patrón basic_string permite manejar otros tipos de cadenas de caracteres, por ejemplo cadenas formadas por caracteres más largos, como los necesarios para almacenar alfabetos orientales. . . o cadenas de caracteres con métricas diferentes.

Entender que el patrón sí mismo no provee maneras de hacer manipulaciones de tipo mayúscula/minúscula sobre los caracteres (que es muy dependiente del alfabeto). Entonces, por default, las operaciones sobre caracteres hacen diferencia entre mayúsculas y minúsculas.

strings: uso

Lo que en el patrón esta "encargado" de las operaciones al nivel del carácter (comparación entre dos caracteres) es la clase traits pasada en parámetro del patrón, ya que se supone que define entre otros métodos: eq (equal), ne (not equal), lt (less than), ...

Para tener un comportamiento diferente en cuanto a los string se necesita sobrecargar la especialización de los traits usados.

strings: uso

Si uno quiere usar string que no hagan diferencias entre mayúsculas y minúsculas, puede definir su propia clase traits (heredando de char_traits<char>) para re-definir una especialización de basic_string

```
struct mitraits : char_traits<char> {
  static bool eq(char a, char b) {
    return toupper(a) == toupper(b);
  }
};
```

strings: creación

- Constructor por default,

```
string s1;
```

- Constructor por cadena de caracteres C,

```
string s2("Hola");
string s3 = "Qiubo ?";
```

- Constructor por copia,

```
string s4(s3);
```

strings: creación

Notar que se puede cortar una cadena de caracteres literal en el código mismo,

```
string s5 = "Que onda"
  ", guey ?";

 cout << s5 << endl;
```

Que nos da :

```
Que onda, guey ?
```

strings: creación

Constructor por selección de una subcadena dentro de una cadena literal o dentro de otro string,

```
string s6("Parangaricutirimicuaro",5,4);
string s7(s5,4,4);
cout << s6 << endl;
cout << s7 << endl;
```

```
gari
onda
```

strings: creación

El método substr también permite crear string a partir de objetos existentes: indicas el índex del primer carácter, el numero de caracteres (máximo) que quieres copiar y genera la sub-cadena correspondiente:

```
string s8 = s6.substr(1,12);
cout << s8 << endl;
```

```
ari
```

strings: creación

Remarca: al usar malos indices, se genera una excepción out_of_range

```
string s8 = s6.substr(10,12);
```

```
terminate called after throwing an instance of
'std::out_of_range'
  what():  basic_string::substr
  Abort
```

strings: creación por operador +

El operador + permite combinar por concatenación todo tipo de cadenas de caracteres:

```
string s9 = s1 + "---" + s6.substr(1,12) + "---"
+ s7;
cout << s9 << endl;
```

```
Que onda, guey ?---ari---onda
```

strings: creación

- También se puede ver los string como casos particulares de contenedores ! Proveen iteradores y iteradores especiales begin() y end()

```
string s10(s9.begin(),s9.end());
cout << s10 << endl;
```

- En fin, se pueden definir a partir de caracteres solos

```
string s11(8,'u');
cout << s11 << endl;
```

uuuuuuuu

strings: pasar a cadenas C

Si quieres manipular una cadena de caracteres "a la C" (con funciones de cstring), existe un método especial c_str() que hace la conversión:

```
const char *cs = s5.c_str();
```

Regresa un apuntador const char *.

strings: pasar a cadenas C

Si regresa un const char *, no es nada inocente : la responsabilidad de la memoria, la tiene el objeto string, entonces evitar absolutamente todo tipo de manipulación directa de la cadena de caracteres C.

strings: acceder a caracteres individuales

Se puede ver el string:

- Como un apuntador, con operador corchete:

```
string s("Guacala");
char c = s[2];
cout << c;
```

a

- Como un contenedor de tipo vector:

```
string s("Guacala");
char c = s.at(2);
cout << c;
```

Sólo el segundo lanza una excepción al usar indices fuera del rango.

strings: acceder a caracteres individuales

Recordar que para la mayoría de los contenedores de la STL, el operador [] y el método $\mathrm{at}()$, se cumple la misma funcionalidad, excepto que el segundo hace un check (entonces es un poco mas lento).

Por eso:

- el primero es peligroso,
- el segundo está susceptible de enviar una excepción.

strings: edición de cadenas de caracteres

En C, hay dos aspectos críticos que cuidar cuando se trata de hacer edición de cadenas de caracteres (inserción, concatenación,...) : la memoria (tenemos suficiente o no ?) y el carácter nulo. En C++ con los string todo eso es transparente !

strings: inserción

Método insert:

```
s5.insert(4,"diablos ");
cout << s5 << endl;
```

```
Que diablos onda, guey ?
```

Hace la inserción al índex dado de una cadena dada. El tamaño es adaptado en consecuencia!

strings: size() y capacity()

```
cout << s5.size() << endl;
cout << s5.capacity() << endl;
s5.insert(4,"diablos ");
cout << s5.size() << endl;
cout << s5.capacity() << endl;
```

```
16
16
24
32
```

size() es el tamaño efectivo de la cadena, mientras capacity() representa la capacidad de almacenamiento (que, en este caso, dobla). Esta capacidad la puedes controlar indicando cuanto tamaño piensas usar, a través del método reserve(int).

strings: size() y capacity()

- Dos sinónimos para el tamaño efectivo de la cadena,

```
cout << s5.size() << endl;
cout << s5.length() << endl;
```

- la capacidad la puedes cambiar tú, pero se maneja generalmente de manera automática (se aumenta sólo, según una política que depende de la implementación).

strings: size() y capacity()

El reserve() permite evitar múltiples re-dimensionalizaciones del contenedor

```
string s = "a";
cout << s.size() << " " << s.capacity() << endl;
s.reserve(400);
while (s.size()<150) s.insert(0,"a ");
cout << s.size() << endl;
cout << s.capacity() << endl;
```

strings: size() y capacity()

El resize() es un método que permite cambiar el tamaño efectivo de la cadena:

- cortando los últimos caracteres si el nuevo tamaño es inferior al previo tamaño,
- completando con espacios sino.

strings: concatenación

Muy similar a insert es el método append que concatena al final de la cadena que llama:

```
s6.append(" cooper ");
cout << s6 << endl;
```

```
gari cooper
```

strings: concatenación

Los operadores de concatenación permiten un manejo relativamente fácil de las concatenaciones

```
string a("Cara ");
string b("de ");
string c("bola.");
a = a + b + c;
a += b+c;
a += c + c[4] + ".";
cout << a;
```

strings: concatenación

Otra manera es la de todos los contenedores: el método push_back(), que agrega un elemento del contenedor (aquí, un carácter):

```
a.push_back('a');
a.push_back('d');
a.push_back('e');
```

strings: remplazar caracteres

Se puede querer remplazar los caracteres o sub-cadenas: para eso sirve la función replace() :

```
s5.replace(0,8,"Como vas");
cout << s5 << endl;
```

```
Como vas, guey ?
```

Remplaza dentro de la cadena llamando, a partir del índex pasado como argumento y por máximo el segundo numero en argumento, los caracteres de la cadena que llama por los caracteres de la cadena del tercer argumento (que puede ser const char * o string).

strings: buscar caracteres

En un uso típico, métodos como replace() van de conjunto con funciones como find() que buscan una sub-cadena dada dentro de una cadena string

```
size_t s = s6.find("coo",1);
cout << s << endl;
```

5

El find() toma como parámetro una subcadena que buscar y un entero que indica a partir de qué índex buscar. En caso de fallo, regresa un valor especial string::npos.

strings: buscar y remplazar caracteres

Un ejemplo:

```
void replaceSubString(string& s,
  const string& pattern, const string& newstring) {
  // Search for substring
  size_t t = s.find(pattern, 0);
  // If found, replace it
  if(t != string::npos)
    s.replace(t, pattern.size(), newstring);
}
```

strings: buscar y remplazar caracteres

Notar que si la cadena pasada en parámetros y remplazada dentro de la cadena original “cabe”, el tamaño de la cadena original no cambia, pero que eventualmente sí podría cambiar si no “cabe”.

```
cout << s5 << "␣" << s5.size() << endl;
size_t s = s5.find("guey",0);
s5.replace(s,8,"mi␣amigo");
cout << s5 << "␣" << s5.size() << endl;
```

```
Que onda, guey ? 16
Que onda, mi amigo 18
```

strings: buscar y remplazar caracteres

Otro ejemplo: remplazar todas las ocurrencias de una cadena

```
string& replaceAll(string& text,
  const string& tobereplaced,
  const string& toreplace) {
  size_t curStart = 0;
  size_t curFound;
  while((curFound = text.find(tobereplaced, curStart)
    != string::npos) {
    text.replace(curFound, tobereplaced.size(),
        toreplace);
    curStart = curFound + toreplace.size();
  }
  return context;
}
```

strings: buscar y remplazar caracteres

Un punto importante: la clase string no tiene todos los algoritmos implementados; de hecho, muchos de esos algoritmos están ubicados en la librería estándar aparte, generalmente expresados en términos de iteradores sobre string,

```
#include <algorithm>
cout << s5 << endl;
replace(s5.begin(), s5.end(), 'a', 'e');
cout << s5 << endl;
```

```
Que onda, mi amigo
Que onde, mi emigo
```

Eso, para que los mismos algoritmos se puedan aplicar a otros tipos de contenedores de caracteres: algoritmos genéricos.

strings: buscar caracteres

Recordar que la STL tiene un montón de sobrecargos de todos los métodos citados. Además, versiones especiales de find

- find() busca un grupo de caracteres a partir del principio, y regresa el índex de la primera ocurrencia,
- rfind() lo hace al reves, empezando desde el final de la cadena,
- find_first_of( ) y find_last_of( ) permiten buscar las primeras y ultimas ocurrencias de un carácter entre los que están pasando en argumento,
- find_first_not_of( ) y find_last_not_of( ) buscan caracteres no entre los que no están en la cadena argumento.

strings: remplazar caracteres

```
cout << s6.find('e') << endl;
cout << s6.find("e") << endl;
cout << s6.rfind("e") << endl;
cout << s6.find_first_of("ubp") << endl;
cout << s6.find_last_of("ubp") << endl;
cout << s6.find_first_not_of("Carambola") << endl;
cout << s6.find_last_not_of("Carambola") << endl;
```

strings: remplazar caracteres

El resultado:

9
9
9
8
8
0
9

strings: buscar caracteres

Para cambiar el comportamiento con respeto a las minúsculas/mayúsculas

```
struct mitraits : char_traits<char> {
  static bool eq(char a, char b) {
    return toupper(a) == toupper(b);
  }
  static const char*
  find(const char* s1, size_t n, char c) {
    while(n-- > 0)
      if(toupper(*s1) == toupper(c))
        return s1;
      else
        ++s1;
    return 0;
  }
  ...
```

strings: buscar caracteres

```
typedef basic_string<char, mitraits> mstring;

inline ostream& operator<<(ostream& os,
                           const mstring& s) {
  return os << string(s.c_str(), s.length());
}
...
 mstring m1 = "CaRaMbOlA";
 cout << m1.find("o") << endl;
 string r1 = "CaRaMbOlA";
 cout << r1.find("o") << endl;
```

strings: remplazar caracteres

- De la misma manera, se puede cambiar el comportamiento para cadenas de caracteres mas largos (en memoria) como los wchar_t,
- son de 8, 16 (Windows) o 32 (Unix) bits,
- están generalmente asociados a los codigos UTF-16 o UTF-32.

strings: quitar caracteres

El método erase() permite quitar caracteres:

```
cout << s6 << endl;
s6.erase(6,1);
cout << s6 << endl;
```

```
gari cooper
gari coper
```

Toma como argumento el índex de principio y el numero (máximo) de caracteres que quitar (default: quita todo los caracteres de un string).

strings: quitar caracteres

Otros métodos útiles:

```
s6.clear(); //
if (s6.empty()) {
...
```

El $\mathrm{clear}()$ reduce una cadena a nada, el $\mathrm{empty}()$ hace el test de que si es vacía o no.

strings: comparación

El orden lexicográfico es el mas clásico para su uso en operadores usados normalmente con números : $\leq$, $<$, $\geq$, $>$, $!=$, $==$:

```
string t1("Tutu");
string t2("Tata");
if (t1>t2)
        cout << "t1 > t2";
else
         cout << "t1 <= t2";
```

Uso más práctico que strcmp... Aceptan también cadenas "a la C".

strings: comparación

Por fin, existe un método compare que puede comparar con mas flexibilidad cadenas o sub-cadenas de caracteres,

```
s1.compare(pos1, n1, s2, pos2, n2);
```

No olvidar que por default, se diferencian mayúsculas y minúsculas (para cambiar este comportamiento, usar char_traits) diferente.