Clases y Memoria Dinámica

Juan F. Huete

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Metodología de la Programación, 2018

Índice

Esquema

Clases ...

- Permiten el encapsulamiento: Un objeto aglutina en un único paquete datos y funciones:
  - Los datos representan las características de una entidad
  - Los métodos determinan su comportamiento. Son el único mecanismo para acceder a los datos
- Permiten el ocultación de información. Nos proporciona una especificación que es independiente de cualquier implementación (podemos cambiar la implementacion sin modificar la especificación)
- Cuando trabajamos con memoria dinámica podemos encapsular toda la información en una clase, asegurándonos la coherencia del objeto en todo momento.

Para ilustrarlo consideraremos una implementación alternativa (aunque muy parecida) de la clase string
... pero antes debemos entender un poco como maneja C, y no C++, las cadenas de caracteres.

Esquema

cadenas C

cadena C

Una cadena de caracteres estilo C es una secuencia de elementos de tipo char con un tamaño determinado acabado en un carácter especial, '\0' (carácter nulo), que marca el fin de la cadena.
El tamaño real del bloque utilizado por una cadena C es uno mas que el número de caracteres de la cadena.

```
1 char A[5]={'H','o','l','a','\0'}; // OK, reserva y asigna
2 char B[] ={'H','o','l','a','\0'}; // OK, reserva y asigna
3 char C[] ="Hola";  // OK,  se reserva y copia contenido + \0
4 char MAL[4]={'H','o','l','a'};  // MAL es un array de 4 char,
5                                 //pero no una cadena
```

literal C

Un literal de cadena de caracteres es un array constante de char con un tamaño igual a su longitud más uno.

```
1   const char L[] = "Hola";    // literal,
2   L[2] = 'X';                 // Error es constante
```

cadenas C

cadena: char *

Recordemos que un array lo podemos tratar con punteros
.... pero hay algunas diferencias, por ejemplo al inicializar.

```
1 char *A={'U','n','a','\0'}; //ERROR A no tiene memoria asignada
2 char *C ="Hola"; //ERROR(1) const char * no se convierte a char
3                  //ERROR(2) C no tiene memoria asignada
```

literal: const char *

Realmente, C++ considera que un literal es de tipo const char *
Copia la dirección de memoria de la constante literal en el puntero.
No es posible modificar caracteres de la cadena.

```
1 const char *L = "Hola"; //OK, L toma direc. memoria del literal
2                         //  \0 esta implicito
3 L[2] = 'X';             // Error es constante
```

Cadenas: inicialización

Se suele reservar mas memoria de la realmente utilizada, para permitir posible cambios en el tamaño de la cadena, pero puede ser peligroso

```
char nombre[10] = "Juan";
    // nombre = | J | u | a | n | \0 | ? | ? | ? | ? | ? |

nombre[4] = 'a' ; // PELIGRO en este momento no es cadena
    // nombre = | J | u | a | n | a | ? | ? | ? | ? | ? |
nombre[5] = '\0' ; // Ahora si
    // nombre = | J | u | a | n | a | \0 | ? | ? | ? | ? |
```

La biblioteca `cstring`

La biblioteca `cstring` (*http://www.cplusplus.com/reference/cstring*) proporciona funciones de manejo de cadenas de caracteres de C. Entre otras:

```
char * strcpy(char *destination, const char *source)
```

Copies the C string pointed by source into the array pointed by destination, including the terminating null character (and stopping at that point).
To avoid overflows, the size of the array pointed by destination shall be long enough to contain the same C string as source (including the terminating null character), and should not overlap in memory with source.

```
int * strlen(const char *str)
```

Returns the length of the C string str.

```
char * strcat(char *destination, const char *source)
```

Appends a copy of the source string to the destination string. The terminating null character in destination is overwritten by the first character of source, and a null-character is included at the end of the new string formed by the concatenation of both in destination.

La biblioteca cstring

```
const char * cadena = "Hola"; char dest[5]; char * p;
strcpy(dest,"Hola"); // OK       | H | o | l | a  | \0 |
strcpy(dest,"Adios"); // ERROR   | A | d | i | o  |  s |
strcpy(cadena,"uno"); // ERROR cadena es constante, literal
strcpy(p,cadena);     // ERROR no hay memoria reservada
p = new char[100];
strcpy(p,"uno");  //OK      | u | n | o | \0 | ? | .... | ? |
delete []p;  // No olvidemos liberar memoria reservada
```

Ejercicios: Implementar las funciones anteriores ....

Todas ellas utilizan \0 para identificar el final de la cadena. Analiza qué pasa en strcat o strcpy cuando dest no es lo suficientemente grande

Como vemos, con cstring tenemos que tener cuidado de todos los detalles que, en gran parte, son ignorados cuando trabajamos con string

```
string cadena = "Hola", dest;
dest = "Hola";          // OK    dest.size() es 4
dest = "adios amigo"; // OK    dest.size() es 11
cadena = uno;           // OK    cadena.size() es 3
```

Índice

Clase string

C++ proporciona una clase específica para en tratamiento de cadenas de caracteres, la clase string

- un string es una cadena de longitud variable, que se adapta a las necesidades. Por tanto, está implementada con memoria dinámica

string
string::string
string::~string
member functions:
string::append
string::assign
string::at
string::back
string::begin
string::capacity
string::cbegin
string::cend
string::clear
string::compare
string::copy
string::crbegin
string::crend
string::c_str
string::data
string::empty
string::end
string::erase
string::find
string::find_first_not_of
string::find_first_of
string::find_last_not_of
string::find_last_of
string::front
string::get_allocator
string::insert
string::length
string::max_size
string::operator+=
string::operator=
string::operator[]
string::pop_back
string::push_back
string::rbegin
string::rend
string::replace
string::reserve
string::resize
string::rfind
string::shrink_to_fit
string::size
string::substr
string::swap
member constants:
string::npos
non-member overloads:
getline (string)
operator+ (string)
operator<< (string)

cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return length of string (public member function)
length	Return length of string (public member function)
max_size	Return maximum size of string (public member function)
resize	Resize string (public member function)
capacity	Return size of allocated storage (public member function)
reserve	Request a change in capacity (public member function)
clear	Clear string (public member function)
empty	Test if string is empty (public member function)
shrink_to_fit	Shrink to fit (public member function)

Element access:

operator[]	Get character of string (public member function)
at	Get character in string (public member function)
back	Access last character (public member function)
front	Access first character (public member function)

Modifiers:

operator+=	Append to string (public member function)
append	Append to string (public member function)
push_back	Append character to string (public member function)
assign	Assign content to string (public member function)
insert	Insert into string (public member function)
erase	Erase characters from string (public member function)
replace	Replace portion of string (public member function)
swap	Swap string values (public member function)
pop_back	Delete last character (public member function)

String operations:

c_str	Get C string equivalent (public member function)
data	Get string data (public member function)
get_allocator	Get allocator (public member function)
copy	Copy sequence of characters from string (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)

http://www.cplusplus.com/referenc

Esquema

Clase string, representación

string.h

representación

```
#ifndef __STRING__
#define __STRING__

class string {
  public:
      .....
  private:
  //inicio c-string
    char *datos;
  // sig ultimo caracter
    char *fin;
  // tama bloque
    int tam;
};

#endif
```

```
string a;   // cadena vacia
```

```
a.datos ->  | \0 |
a.fin   ------^
a.tam = 1
```

```
a = "Hola"; // reserva bloque
```

```
a.datos ->  | H | o | l | a | \0 |
a.fin ----------------------^
a.tam = 5
```

```
a += " tu"; //aumenta tam -> reserva
```

```
a.datos ->  | H | o | l | a |  | t | u | \0 |
a.fin  ------------------------------^
a.tam = 8
```

```
a = "tu"; // no ajustamos, evita frag.
```

```
a.datos ->  | t | u | \0 | ? | ? | ? | ? | ? |
a.fin ---------------^
a.tam = 8
```

Esquema

La clase string

Implementación del TDA string: métodos

Consideraremos ahora los diferentes métodos que deberían completar la definición del TDA **string**. Conviene seguir el siguiente orden:

- constructores (construyen un string válido)
- operaciones naturales sobre string (deberían ser métodos públicos)
- destructores (liberan la memoria utilizada)

Asumimos que cada vez que se agregue un método debe incorporarse a la declaración de la clase, en el archivo **string.h** y su implementación en **string.cpp**

es posible que aparezcan otros métodos que resulten convenientes como métodos auxiliares (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). Estos métodos deberían ser privados.

Esquema

Los constructores de la clase

Los constructores se encargan de inicializar de forma conveniente los datos miembro. En este caso deben además reservar la memoria dinámica que sea necesaria.

Constructor por defecto

Es el constructor sin parámetros. Una clase lo puede tener mediante:

- El compilador lo crea implícitamente cuando la clase no define ningún constructor.
  - Tal constructor no inicializa los datos miembro de la clase.
  - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
  - Un dato miembro no inializado probablemente contendrá un valor basura.
- Definiéndolo explícitamente en la clase.

En nuestro caso el constructor construido implícitamente no es útil ya que no genera un string válido, por lo que debemos sobreescribirlo

Los constructores de la clase string

Constructor por defecto de la clase string

Crea espacio para un string nulo `""`. Cabe plantearse qué valores dar a los datos miembro:

- datos debe tener memoria reservada para alojar un char: `'\0'`
- fin apunta al inicio de la zona reservada
- tam debe ser 1

```
string a;
```

```
a.datos ->  | \0 |
a.fin   ------^
a.tam = 1
```

Los constructores de la clase string

string.h

representación

```
#ifndef __STRING__
#define __STRING__

class string {
  public:
    string();
      .....
  private:
    char *datos;
    char *fin;
    int tam;
};

#endif
```

string.cpp

implementación

```
string::string( ) {
  datos = new char;
  *datos = '\0';
  fin = datos;
  tam = 1;
}
```

Los constructores de la clase string

string (size_t n, char c)

(cplusplus/reference) Fills the string with n consecutive copies of character c.

- datos debe tener memoria reservada para alojar n+1 caracteres ya que debemos incluir el `'\0'`
- fin apunta a la dirección donde está el `'\0'`
- tam debe ser n+1

```
string a(5,'x');
```

```
a.datos ->  | x | x | x | x | x | \0 |
a.fin       ------------------------^
a.tam = 6
```

size_t

Tipo definido en <cstddef> <cstdio> <cstdlib> <cstring> <ctime> <cwchar> como unsigned int, utilizado para representar el tamaño de cualquier objeto.

Los constructores de la clase string

string.h

representación

```
#ifndef __STRING__
#define __STRING__
#include <cstring> // size_t
class string {
  public:
    string();
    string(size_t n,char c);
      .....
  private:
    char *datos;
    char *fin;
    int tam;
};

#endif
```

string.cpp

implementación

```
string::string(size_t n, char c) {
  datos = new char[n];
  int i=0;
  for (  ; i<n ;i++){
    datos[i] = c;
    }
  datos[i] = '\0';
  fin = datos+n;
  tam = n+1;
}
```

Los constructores de la clase string

string (const char *s)

(cplusplus/reference) from c-string. **Copies** the null-terminated character sequence (C-string) pointed by s.

- datos apunta a un bloque de memoria dinámica con capacidad para copiar todos los caracteres de s (length), incluyendo el `'\0'`
- fin apunta a la dirección donde está el `'\0'`
- tam debe ser length+1

```
char cad[10]="Adios";

string a("Hola");
string b(cad); // char [ ] == char * -> const char *
```

```
a.datos ->  | H | o | l | a | \0 |
a.fin       ---------------------^
a.tam = 5
```

Los constructores de la clase string

string.h

representación

```
#ifndef __STRING__
#define __STRING__
#include <cstring>
class string {
  public:
    string();
    string (size_t n, char c);
    string (const char *s)
      .....
  private:
    char *datos;
    char *fin;
    int tam;
};

#endif
```

string.cpp

implementación

```
string::string(const char *s) {
  int l;
  l = strlen(s); //<cstring>
  datos = new char[l+1]; //\0
  strcpy(datos,s);
  fin = datos+l;
  tam = l+1;
}
```

Esquema

Constructor de copia

- Es posible crear un constructor de copia que haga una copia correcta de un objeto de la clase en otro.
- Al ser un constructor, tiene el mismo nombre que la clase.
- No devuelve nada y tiene como único parámetro, constante y por referencia, el objeto de la clase que se quiere copiar.
- Copia el objeto que se pasa como parámetro en el objeto que construye el constructor.

Se llama:

1. Explícitamente por el programador
2. Al hacer un paso por valor para copiar el parámetro actual en el formal.
3. Cuando se devuelve un objeto de la clase por valor

Constructor de copia por defecto

- Es esencial para controlar el paso y retorno de variables por valor.
- Es tan importante que el compilador crea automáticamente un constructor de copia en caso de que programador no lo implemente.

Llama a los constructores de copia de todos los miembros almacenados en la clase (o estructura)

Cada atributo será una copia “exacta” de los atributos del objeto pasado como parámetro

- Esta copia no tiene sentido cuando consideramos memoria dinámica

La copia por defecto de un puntero hace copia bit a bit de la dirección de memoria a la que apunta, pero no del contenido almacenado en ella.

Constructor de copia por defecto

Copia blanda

- Efecto no deseado: Los dos objetos, el origen y la copia, compartirán el mismo bloque de memoria del heap

```
struct E1 {  int x; };
E1 a1; a1.x = 10;

E1 b1(a1);
```

```
struct E2{  int *pX; };
E2 a2; a2.pX= new int[2];
a2.pX[0] = 1; a2.pX[1]=2;
E2 b2(a2);
```

```
a1.x = 10                    a2.px = 0x0010----------v
                                                     | 1 | 2 |
b1.x = 10                    b2.px = 0x0010 ----------^
```

La modificación de los valores en dicho bloque para un objeto, implica la modificación en el otro.

```
a1.x = 5;
```

```
a2.pX[0] = 0;
```

```
a1.x = 5                     a2.px = 0x0010----------v
                                                     | 0 | 2 |
b1.x = 10                    b2.px = 0x0010----------^
```

Necesidad del constructor de copia

Si el efecto anterior no es el deseado, es necesario implementar el constructor de copia para garantizar un control total sobre las variables.

Copia dura

- Al hacer la copia se copia el contenido del bloque al que apunta
- Implica reserva de memoria y copiar contenidos.

  Las variables puntero `(a2.pX y b2.pX)` *apuntarán a direcciones de memoria (bloques) distintas, pero el contenido de los bloques será idéntico.*

- Tenemos dos objetos, con la misma información, pero independientes

```
struct E1 {  int x; };                  struct E2{  int *pX; };
E1 a1; a1.x = 10;                       E2 a2; a2.pX= new int[2];
                                        a2.pX[0] = 1; a2.pX[1]=2;
E1 b1(a1);                              E2 b2(a2);
```

```
a1.x = 10                       a2.px = 0x0010 ------------> | 1 | 2 |
b1.x = 10                       b2.px = 0x0028 ------------> | 1 | 2 |
```

Los constructores de la clase string

string (const string & str)

(cplusplus/reference) copy constructor. Constructs a copy of str.

- datos apunta a un bloque de memoria dinámica con capacidad para copiar todos los caracteres del string str, incluyendo el `'\0'`
- fin apunta a la dirección donde está el `'\0'`
- tam debe tener el mismo tamaño de str

```
string funcion( string x ){
   string cad;
   .....
   return cad; // <<<<<< 3  Llamada al constructor de copia
}
string a("Hola");
string b(a);   // <<<<<<<<< 1 Llamada al constructor de copia
funcion(a);    // <<<<<<<<< 2 Llamada al constructor de copia
```

```
b.datos ->  | H | o | l | a | \0 |
b.fin   ---------------------^
b.tam = 5
```

Los constructores de la clase string

string.h

representación

```
#ifndef __STRING__
#define __STRING__

class string {
  public:
    string();
    string (size_t n, char c);
    string (const char *s)
    string (const string & s)
      .....
  private:
    char *datos;
    char *fin;
    int tam;
};

#endif
```

string.cpp

implementación

```
string::string(const string &s)
      {
  int l =s.size(); //s.length()
  datos = new char[l+1]; // \0
  strcpy(datos,s.datos);
  fin = datos+l;
  tam = s.tam;
}
```

Esquema

Conversiones implícitas

Conversión implícita

Cualquier constructor (excepto el de copia) de un sólo parámetro puede ser usado por el compilador de C++ para hacer una conversión automática de un tipo al tipo de la clase del constructor.

```
void funcion(const string a, double b){
  for (int .....)
}

int main(){
   string a;
   prueba(a,2.5); // llamada con string,
                  // convierte (string &) -> (const string &)
   prueba("Hola",3.8); // se hace casting implicito,
                       // (const char*) ->  (const string &)
}
```

Esquema

Destrucción automática de objetos locales

- Como se ha visto en temas anteriores, las variables locales (gestionadas por el compilador) se destruyen automáticamente al finalizar la función (o bloque) en la que se definen.
- El **destructor por defecto** generado por el compilador llama uno a uno a los destructores de cada miembro
- Sin embargo, las memoria gestionada por el programador NO se libera

```
1  float funcionA( ){
2    int v[4]={1,2,3,4};
3    string s("Hola mundo");
4    ....
5    return valor;
6  }
7  int main(){
8    ...
9    a = funcionA()
0  }
```

En funcionA se reservan

```
stack:   dir     var    tipo bytes valor
         0x0001  v       pI      8  0x0010
         0x0010                4x4  |1|2|3|4|
         0x0026  s.datos pC      8  0x0B00
                 s.fin   pC      8  0x0B11
                 s.tam    I      4  11

Heap:   dir        tipo bytes  valor
        0x0B00     pC   11x1   |H|o|...|\0|
```

Al salir de funcionA

```
1  stack: [  ] men liberada
2
3  "Heap:   dir       tipo bytes  valor
4           0x0B00    pC   11x1   |H|o|...|\0|
5  queda bloqueado, perdemos toda referencia "
```

El destructor de una clase

- Es un método que permite liberar toda la memoría asociada a un objeto, automatizando el proceso de destrucción
- El destructor es único y se llama con el nombre de la clase precidido por el símbolo $\sim$
- No lleva parámetros y no devuelve nada, por ejemplo $\sim$**string( )**
- Se ejecuta de forma automática, al finalizar el ámbito en el que está definido el objeto
  - Los objetos que son locales a una función o trozo de código, justo antes de acabar la función o trozo de código.
  - Los objetos variable global, justo antes de acabar el programa.

Si NO hemos reservado memoria dinámica NO tenemos que implementarlo, nos vale con el

Sólo debemos de implementarlo si la clase usa reserva de memoria dinámica, debiendo liberar exclusivamente la memoria dinámica reservada.

Desctructor de la clase string

string.h

representación

```
#ifndef __STRING__
#define __STRING__

class string {
  public:
    string();
    string(const string & s);
    .....
    ~string();
  private:
    char *datos;
    char *fin;
    int tam;
};

#endif
```

string.cpp

implementación

```
string::~string( ) {
   delete []datos;
   // libera memoria heap
   // fin no tiene asociada
    memoria, no hay nada que
    liberar
}
```

Ejemplo de llamadas al destructor

Al ejecutar el siguiente ejemplo puede verse en qué momento se llama el destructor de la clase.

```
#include <iostream>
using namespace std;
class Prueba{
 public:
   Prueba();
   ~Prueba();
};

Prueba::Prueba(){
   cout<<"Constructor"<<endl;
}
Prueba::~Prueba(){
   cout<<"Destructor"<<endl;
}
void funcion(){
   Prueba local;
   cout<<"funcion()"<<endl;
}

Prueba varGlobal;
int  main(){
   cout<<"Comienza main()"<<endl;
   Prueba ppal;
   cout<<"Antes de llamar a funcion()"<<
         endl;
   funcion();
   cout<<"Despues de llamar a funcion()"<<
         endl;
   cout<<"Termina main()"<<endl;
}
```

En la traza se han agregado comentarios para aclarar en qué momento se genera cada línea.

```
Constructor          "// Construccion objeto varGlobal"
Comienza main()      "// Inicio ejecucion main"
Constructor          "// Construccion objeto ppal"
Antes de llamar a funcion()
Constructor          "// Construcc objeto funcion::local"
funcion()            "// Ejecucion funcion"
Destructor           "// Destruye objeto funcion::local"
Despues de llamar a funcion() "// De vuelta en main"
Termina main()       "// Finaliza ejecucion main"
Destructor           "// Se destruye objeto ppal"
Destructor           "// Se destruye objeto varGlobal"
```

Clases con datos miembro de otras clases

```
1  class alumno {
2    private:
3        string nombre;
4        int *notas;
5        int cuantas;
6    public:
7       alumno();
8       ~alumno()
9   };
0  alumno::alumno(){
1    notas = new int[5];
2    cuantas = 5;
3  }
4  alumno::~alumno(){
5     delete []notas;
6  }
```

Constructor

Un constructor de una clase:

- Llama al constructor por defecto de cada miembro (string,int*,int)
  nombre será el string por defecto (“ ”)
- Ejecuta el cuerpo del constructor.
  (reserva memoria y asigna cuantas=5)

Destructor

El destructor de una clase:

- Ejecuta el cuerpo del destructor de la clase del objeto. (libera notas)
- Luego llama al destructor de cada dato miembro. (libera nombre string y cuantas)

Esquema

Consultores de una clase: Métodos const

- Un consultor es cualquier método que permita obtener información almacenada en un objeto de la clase.
  Por ejemplo en string
  - Devolver el tamaño de la cadena

  Como tal, un consultor debería tener prohibido modificar el objeto.
- C++ nos permite garantizar esta restricción y en caso de violarla de forma accidental (al programar) obtener un error de compilación.

Métodos const

Declarar un método como constante, poniendo la palabra reserva **const** al final de la declaración, evitará cualquier intento de modificar los datos miembro en el método (de forma directa o indirecta)

```
class string {
  ....
  size_t size() const;
  size_t length() const;
  size_t capacity() const;
  size_t find(.....) const;
```

Uso de métodos const

Un objeto declarado como const **sólo** puede llamar a métodos declarados const.

```
1  class string {
2    size_t size() const;
3    size_t tama();  // Metodo imaginario
4  };
5  void pintaLongitud( const string & s) { //s es ref constante
6     cout <<"long"<< s.size(); // OK el metodo es const
7     cout <<"long"<< s.tama(); // ERROR comp. metodo NO es const
8  }
9  int main(){
10  string a("Hola");
11  pintaLongitud(a);  // se le pasa referencia constante
12  cout << a.size() << endl; // OK conversion implicita a const
13  cout << a.tama() << endl; // OK ambos no const
14  const string b("adios");
15  cout << b.size() << endl; // OK ambos son const
16  cout << b.tama() << endl; // ERROR b es un objeto constante
```

Consultores de la clase string

- size: Devuelve el tamaño del string, número de caracteres sin el '\0'
- length: Devuelve el tamaño del string, número de caracteres sin el '\0'
- capacity: Devuelve el número de bytes reservados para el string en este momento, sin el '\0'

string.h

representación

```
class string {
  public:
    .....
    size_t size() const;
    size_t length( )const;
    size_t capacity() const;
  private:
    char *datos;
    char *fin;
    int tam;
};
```

string.cpp

implementación

```
size_t string::size() const {
    return fin-datos;
}
size_t string::length() const {
    return fin-datos;
}
size_t string::capacity() const{
    return tam-1;
}
```

Consultores de la clase string

- data: Returns a pointer to an array that contains a \0-terminated sequence of chars (i.e., a C-string) representing the current value of the string. The pointer returned points to the internal array currently used

Al devolver **const char *** nos aseguramos que el string NO se puede modificar desde fuera de la clase.

- c_str: Equivalente.

string.h

```
class string {
  public:
    const char* data() const;
    const char* c_str() const;
  private:
    char *datos;
    char *fin;
    int tam;
};
```

string.cpp

```
const char* string::data()
    const {
    return datos;
}
const char* string::c_str()
    const {
    return datos;
}
```

Ejercicio

Implementar los consultores:

```
size_t find (char c, size_t pos = 0) const
```

Searches the string for the first occurrence of the character c, pos is the position of the first character in the string to be considered in the search. The position of the first character of the first match. If no matches were found, the function returns `string::npos`.

```
string cad ("la blanca paloma");
cout << cad.find('b',0); // 3
cout << cad.find('b');   // 3
cout << cad.find('p',5); //10
```

```
size_t find (const string& str, size_t pos = 0) const;
```

Searches the string for the first occurrence of the string str, pos is the position of the first character in the string to be considered in the search.

```
string cad ("La blanca paloma");
cout << cad.find("la");   // 0
cout << cad.find("la",2); // 4
```

Esquema

Modificadores de una clase

- Un modificador es cualquier método que permite modificar la información almacenada en un objeto.
- Cuando trabajamos con memoria dinámica un modificador puede necesitar realizar nuevas peticiones de memoria para permitir alojar nueva información, como por ejemplo cuando el tamaño del objeto varía. En estos casos se recomienda realizar una copia dura del objeto.

  Por ejemplo en string
  - `clear` Borra todo el contenido de la cadena
  - `push_back` Añadir un elemento al final de la cadena
  - `resize` Aumentar el tamaño de la cadena

```
class string {
    void push_back (char c);
    void clear();
    void resize (size_t n, char c); //aumenta hasta n char
```

```
// posiciones                          0123456789*123456789*12345
string str("esto es un ejemplo"); // "esto es un ejemplo"
str.push_back('!');               // "esto es un ejemplo!"
str.resize(22,'.');               // "esto es un ejemplo!...."
str.clear();                      // ""
```

Modificadores de la clase string: clear

- `clear` Erases the contents of the string, which becomes an empty string (with a length of 0 characters).

Realojo: Any pointers and references related to this object may be invalidated.

string.h

```
class string {
  public:
    void clear();
  private:
    char *datos;
    char *fin;
    int tam;
};
```

string.cpp

```
void string::clear(){
    delete []datos;
    datos = new char;
    *datos='\0';
    fin = datos;
    tam = 1;
}
```

Modificadores de la clase string: push_back

- push_back: Appends character c to the end of the string, increasing its length by one. The object is modified.

Realojo: Any pointers and references related to this object may be invalidated.

```
1 string b("hola");
2 cout << b.size() <<" "<< b.capacity() <<endl;
3 b.push_back('!');
4 cout << b.size() <<" "<< b.capacity() <<endl;;
5 b.push_back('!');
6 cout << b.size() <<" "<< b.capacity() <<endl;;
```

Salida

```
4 4
5 8
6 8
```

```
b.datos 0x001->  | H | o | l | a | \0 |
b.fin   0x005 ---------------------^
b.tam = 5
```

```
b.datos 0x011->  | H | o | l | a | ! | \0 | ? | ? | ? |
b.fin   0x016 -------------------------^
b.tam = 9
```

```
b.datos 0x011->  | H | o | l | a | ! | ! | \0 | ? | ? |
b.fin   0x017 ---------------------------^
b.tam = 9
```

Modificadores de la clase string: push_back

```
class string {
  public:
    void push_back (char c);
    ...
```

```
  private:
    char *datos;
    char *fin;
    int tam;
```

```
1  void string::push_back(char c){
2    if(tam == (fin-datos+1) ){       // vector lleno
3      int ntam = 2*capacity()+1;  // doblamos tam y '\0'
4      char *aux = new char[ntam]; // reservamos
5      strcpy(aux,datos);  // copiamos datos
6      delete []datos;     // liberamos
7      datos = aux;        // asignamos a datos nuevo bloque
8      fin = datos+tam-1;  // actualizamos fin
9      tam = ntam;
10     }
11   *fin = c;             // estamos seguros que entra c
12   ++fin;
13   *fin = '\0';
14 }
```

Esquema

Devolviendo referencias

Hay muchas situaciones en las que podemos estar interesados en devolver una referencia al objeto en si o a algún atributo del mismo

- Muchos modificadores devuelven una referencia al objeto ya modificado, por ejemplo `string & string::replace(...)`. Devolver la referencia permite:
  - encadenar llamadas a métodos de forma eficiente sin necesidad de llamar al constructor de copia del objeto (evita costos innecesarios)

```
cd.replace(9,5,str2) // es una referencia al objeto cd,
cout<< cd.replace(9,5,str2).replace(1,3,str3).size();
cout << 234 << 'a' << x << endl; // << encadenando refs.
```

- Tanto consultores como modificadores pueden devolver una referencia a algún atributo, lo que puede evitar copias innecesarias, por ejemplo `char & string::at( )`.

```
string cd="this is a test string.";
cout << cd.at(0); // usamos la ref, no copia el char 't'
cd.at(0) = 'T';   // en la ref [cd.at(0)] se escribe 'T'
```

Puntero this

Cómo devolver una referencia al propio objeto 

Todo objeto de una clase contiene un puntero oculto llamado **this** que apunta a dicho objeto, esto es, **this** contiene la dirección del objeto al que referencia. No hay que declararlo, C++ lo genera automáticamente

```
class string {
  private:
     char *datos;
     char *fin;
     int tam;
     // string * const this; // Se crea implicitamente
```

- **this** es un puntero al objeto
- ***this** es el objeto es sí
- **(*this).tam** o **this->tam** es el atributo tam(año)

Al estar definido como `T * const this` podemos cambiar el contenido al que apunta pero no su valor

Uso de this: lo debemos utilizar cuando

- Para distinguir atributos de la clase de parámetros formales

```
void string::diferenciar(char * datos, int x){
   this->datos[0]=datos[0] // parametros con el mismo nombre
   this->tam = x;          // o solo por distinguir
}
```

- Cuando devolvemos una referencia al objeto

```
string & string::devReferencia( ){
      ...
     return *this; // devolvemos la ref al objeto en si
}
```

- Comparar si el objeto y parámetro pasado por referencia son el mismo

```
void string::elMismo(string & otro ){
   if (this==&otro)
      cout << "Mismo objeto " << endl
   else cout << "Objetos distintos " << endl;
}
       string a("hola"), b("hola");
       a.elMismo(b);  // Objetos distintos
       a.elMismo(a);  // Mismo objeto
```

Modificadores de la clase string: erase

```
string& erase (size_t pos = 0, size_t len = npos);
```

- erase: Erases the portion of the string value that begins at the character position pos and spans len characters (or until the end of the string, if either the content is too short or if len is string::npos.

Realojo: Any pointers and references related to this object may be invalidated.

```
1 string b("Hola mundo");
2 b.erase(5,3);
3 b.erase(0,string::npos); //equiv. b.erase(0), borra todo
```

```
b.datos 0x001->  | H | o | l | a |  | m | u | n | d | o | \0 |
b.fin  0x00b-> ------------------------------------^
b.tam = 11
```

```
b.datos 0x001->  | H | o | l | a |  | d | o | \0 | ? | ? | ? |
b.fin  0x008-> ----------------------------^
b.tam = 11
```

```
b.datos 0x00c->  | \0 |
b.fin  0x00c ------^
b.tam = 1
```

Modificadores de la clase string: erase

string.h

```
class string {
 public:
  string & erase(size_t pos=0,
                 size_t len=npos);
```

```
private:
  char *datos;
  char *fin;
  int tam;
```

string.cppp

```
1 string & string::erase(size_t pos, size_t len){
2   if(pos==0 && len==string::npos)
3       clear(); // this->clear(); No es el standard reducir tam
4   else {
5     strcpy(datos+pos,datos+pos+len);   //OJO
6     fin = fin-len;
7   }
8   return *this;
9 }
```

Fijaros en la declaración de los parámetros por defecto

Devolviendo referencias a atributos de una clase

- Cuando devolvemos una referencia a un atributo de la clase tenemos que diferenciar entre los dos posibles contextos en los que el objeto puede ser utilizado
  - Contexto Constante: El objeto, y por tanto sus atributos, no se pueden modificar. Por tanto, actúa como consultor.

```
void pinta(const string & sc){
   // sc es constante, no se puede modificar
   cout << sc.size() << endl;
 }
int main(){
 const string x("aaa");    // x es string constante
```

  - Contexto No Constante: Tenemos permiso para modificar el objeto y por tanto tiene sentido devolver una referencia al atributo de nuestro interés.

```
void modifica(string & s){
   s.push_back('a');  // s es ref y puede modificarlo
 }
int main(){
   string x("aaa");    // x es string constante
   x.clear();
```

Devolviendo referencias a atributos de una clase

Necesitamos espeficar el tipo de referencia teniendo en cuenta el contexto en que se puede utilizar el método

- Contexto Constante: Tanto la referencia devuelta como el propio método deben ser declarados como constantes

```
class objeto {
  public:
    const Tipo & metodoC ( ... ) const;   // Version
    constante
  private:
    Tipo dato;
```

- Contexto No Constante: Permite devolver referencia simple. El método no puede ser constante, pues en ese caso sólo devuelve refs constantes.

```
class objeto {
  public:
    Tipo & metodoVNC ( ... );          // Version no constante
  private:
    Tipo dato;
```

Esquema

Acceso seguro: at

Dependiendo del contexto, nos puede interesar tener un "mismo"método con comportamiento distinto, actuando como consultor o modificador. Veamos por ejemplo el método at:

```
class string {
  char& at (size_t pos);                  // (1) Version no constante
  const char& at (size_t pos) const; //(2) Version constante
```

Returns a reference to the character at position pos in the string.
The function automatically checks whether pos is the valid position of a character in the string (i.e., whether pos is less than the string length), throwing an exception if it is not.
If the string object is const-qualified, the function returns a const char&. Otherwise, it returns a char&

```
void pinta(const string & s){
   for (int i=0;i<s.size();i++) cout << s.at(i); // Ver. (2)
}
void cambia(string & s){
   for (int i=0;i<s.size();i++)  s.at(i)='x';     // Ver. (1)
}
```

Acceso seguro: at

string.h

```
#include <cassert>
class string {
  char& at (size_t pos);                  // (1) Version no constante
  const char& at (size_t pos) const; //(2) Version constante
```

string.cppp

```
char& string::at (size_t pos){
   assert(0<=pos && pos<tam-1);   // Si la expresion es false
                                  // muestra mensaje y aborta
   return datos[pos];
}
const char& string::at (size_t pos) const{
   assert(0<=pos && pos<tam-1);
   return datos[pos];
}
```

Fijaros en #include <cassert>. En std::string lanza excepción

Esquema

Operadores y Clases: Introducción

- Nos podemos preguntar qué método debemos implementar para poder asignar un string a otro.
  En principio podemos pensar en algo como (existe realmente en string)

```
class string {
    string& assing (const string & s);
    string& assign (const char* s, size_t n);
```

  Assigns a new value to the string, replacing its current contents. In the second case, copies the first n chars from the array of chars pointed by s.
- Sin embargo, esto choca con lo que sería el estándar de asignación para los otros tipos, donde se utiliza el operador de asignación, $=$.
- Este hecho es causa de problemas a la hora de diseñar/utilizar algoritmos, pues debemos diferenciar entre tipos de datos para realizar la misma operación conceptual, una asignación

```
int x, y; string u,v;
   y = x;
   u.assign(v);
```

solución: sobrecarga de operadores

Operadores

- Un operador es una función con un nombre especial, `operator?`, donde `?` puede ser cualquiera entre:

Operadores que podemos implementar

```
+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<=
     == != <= >= && || ++ -- , ->* -> ( ) [ ]}
```

- C++ no impone ninguna restricción sobre que debe realizar un operador, salvo que el resultado debe ser el lógico, `operator+` se espera que añada o `operator=` se espera que asigne.
- Implementar un operador no implica que estén implementados otros que pudiesen estar relacionados. Por ejemplo, si implementamos `operator+` y `operator=` no implica tener implementado `operator+=`
- Algunos operadores se pueden implementar fuera de la clase y otros dentro, la elección depende del programador aunque la norma general es implementar como internos todos aquellos que modifiquen el objeto: Por ejemplo, en string son externos los operadores `<<`, `>>`, `+` y los relacionales `<`, `<=`, `>`, `>=`, `==`, `!=`

Operador de asignación: operator=()

- C++ nos permite definir el operador de asignación, `operator=`.

Por defecto, C++ implementa de forma automática el `operator=` para cualquier clase realizando una copia **bit a bit** de cada uno de los atributos. Recordemos que este tipo de copia es peligrosa cuando trabajamos con memoria dinámica, no siendo normalmente el comportamiento deseado. Por lo que se recomienda implementar dicho operador explícitamente.

El formato **estándar** de un operador de asignación es

```
Objeto & operator=(const Objeto & s);  (1)   // Mismo tipo
Objeto & operator=(const Otro & x);    (2)   // Conversion
```

Recibe un único parámetro que es un objeto del tipo que queremos asignar y devuelve una referencia al objeto modificado

```
Objeto o1,o2; Otro x;
  o1 = o2;   // (1) precompilador transf.:  o1.operator=(o2);
  o2 = x;    // (2)                         o1.operator=(x);
```

string::operator=

```
class string {
  string& operator= (const string & s);     // (1) string
  string& operator= (const char* s);     //(2) c-string
```

Assigns a new value to the string, replacing its current contents.

```
string a("Hola"), w, r, s;
w = a;          // (1) eqv  w.operator=(a);
a = "Buenas"; // (2) eqv  a.operator=("Hola");
r = s = w = a = "Buenas" ; // encadenamos asignacioens
                           // posible al devolver string&
```

```
a.datos ->  | H | o | l | a | \0 |        w.datos -> | \0 |
a.fin ---------------------^           w.fin  ------^
a.tam = 5                              w.tama = 1
-------------------------------------------------------
a.datos ->  | H | o | l | a | \0 |        w.datos ->  | H | o | l | a | \0 |
a.fin ---------------------^           w.fin  ---------------------^
a.tam = 5                              w.tama = 5
-------------------------------------------------------
a.datos ->  | B | u | e | n | a | s | \0 | w.datos ->  | H | o | l | a | \0 |
a.fin -----------------------------^   w.fin  ---------------------^
a.tam = 7                              w.tama = 5
-------------------------------------------------------
```

string::operator=

```
string& operator= (const string & s);  // asignar string
```

string.cppp

```
string& string::operator= (const string & s){
   if (this != &s) {     // si s=s no hace nada
     if(tam <= s.tam){  // No hay espacio, necesitamos memoria
        delete [] datos;           // liberamos
        datos = new char[s.tam]; // reservamos memoria para
    this
        tam = s.tam;
     }
      strcpy(datos,s.datos);
      fin = datos+tam;
    }
   return *this;
}
```

- Ejercicio: Implementar el otro operador de asignación. Si no se implementa se crea un string temporal, esto es, mas reservas...:

```
  s = "Hola"; (1) Se crea un string temp usando constructor
```

string::operator[]

- Consigue el mismo efecto que el operador[] en arrays estáticos, acceder a una posición del string
- Al igual que el, no hace ningún chequeo sobre la validez de la posición (al contrario que el método `at`), por lo que cualquier acceso fuera de los límites del string tiene un comportamiento no definido

```
class string {
  char& operator[] (size_t pos);      // (1) Version no constante
  const char& operator[](size_t pos) const;//(2) Vers constante
```

Returns a reference to the character at position pos in the string. If pos is equal to the string length, the function returns a reference to the null character that follows the last character in the string (which should not be modified).
If the string object is const-qualified, the function returns a const char&. Otherwise, it returns a char&

```
void pinta(const string & s){
   for (int i=0;i<s.size();i++) cout << s[i]; // Ver. (2)
}
void cambia(string & s){
   for (int i=0;i<s.size();i++)  s[i]='x';     // Ver. (1)
}
```

string::operator[]

string.h

```
#include <cassert>
class string {
  char& operator[] (size_t pos);     //   Version no constante
  const char& operator[](size_t pos) const; //Version constante
```

string.cpp

```
char& string::operator[] (size_t pos){
   return datos[pos];
}
const char& string::operator[] (size_t pos) const{
   return datos[pos];
}
```

s[s.size()+5]='x' comportamiento indefinido

operator==

- Como hemos dicho esta implementado fuera de la clase string

string.h

```
bool operator==(const string & izq, const string & dch);
```

string.cpp

```
bool operator==(const string & izq, const string & dch){
bool igual = true;
  if ( izq.size()!=dch.size() ) igual = false;
  else {
    for (int i=0; i<izq.size() && igual; i++)
       if (izq[i]!=dch[i]) igual = false;
  }
 return igual;
}
```

 Podemos declarar el `operator==` como función amiga para mejorar el rendimiento

Esquema

Intercambio de valores: swap

Supongamos que tenemos el siguiente código

```
void ordenaSeleccion(string v[], int tama){
  for (int paso = 0; paso < tama; paso++){
      size_t pm = pos_menor(v,paso+1,tama);
      string temp = v[pm];  // Intercambiamos
      v[pm]= v[paso];
      v[paso]=temp};
      }
}
...
string vs[10000]={"aaa", "ddd", "cccc", "bbb", ....};
ordenaSeleccion(vs,10000);
for (int i=0; i< 10000; i++) cout << vs[i];
```

Hay que hacer múltiples asignaciones entre string, donde en cada uno se realiza una copia dura (reservamos memoria, copiamos elementos, liberamos, ...) donde el objetivo final es intercambiar elementos.

En estos casos es conveniente implementar un método, `swap` que realice esta acción de forma eficiente

string::swap

```
void ordenaSeleccion(string v[], int tama){
  for (int paso = 0; paso < tama; paso++){
     size_t pm = pos_menor(v,paso,tama);
     v[pm].swap(v[paso]);  // Intercambiamos
     }
}
```

swap sólo intercambia los punteros al string, no realiza copias ni reservas

```
a.tam = 5
a.fin ----------------------------
a.datos 0x0010 -->                    |
                | H | o | l | a | \0 |      | U | n | a | \0 |
w.datos 0x0040 ----------------------------^
w.fin ------------------------------------------------^
w.tama = 4
=============================================================
a.tam = 4
a.fin ------------------------------------------------
a.datos 0x0040 ----------------------------->            |
                | H | o | l | a | \0 |      | U | n | a | \0 |
w.datos 0x0010 --^
w.fin -------------------------^
w.tama = 5
```

string:swap

string.h

```
  void swap(string & s);   // Exchange
```

string.cpp

```
void string::swap ( string & s){
    char *temp;
    int ttam;
    temp = s.datos;     s.datos = datos;     datos = temp;
    temp = s.fin;       s.fin   = fin;       fin = temp;
    ttam = s.tam;       s.tam   = tam;       tam = ttam;
}
```

Esquema

Métodos inline

Cuando un método/función es muy sencillo, C++ nos permite ahorrarnos la secuencia de pasos que se produce tras llamarlo: guardar estado en pila, copiar parámetros, salto, reserva de var. locales y retorno.
Sería más eficiente, si compilador pudiese sustituir las llamadas al método por el bloque de sentencias que lo componen (limitado a métodos con pocas líneas de código). Para ello hay que definirlo como **inline**:

- Definirlo en la propia clase, se asume **método inline** (no recomendado)
- Declaración normal en la clase, y explícitamente tratarlo como **inline** en su definición (al implementarlo). (recomendado)

string.h

```
#include <cassert>
class string {
  size_t size ( ) const;
};

inline size_t string::size( ) const{ //en .h, el compilador
    return fin-datos;                 // debe conocer la definc
}                                    // para realizar sustituc.
```

Esquema

string.h

```
#ifndef __STRING__H
#define __STRING__H
// Includes
#include <cassert>
#include <cstring>
class string {
  public:
  // Constante npos Mayor posible valor de size_t (unsigned int)
    static const size_t npos = -1;
  // Metodos
    string();
    size_t size() const;
    string& erase(size_t pos = 0, size_t len = npos);
    ~string();
  //operadores
    string& operator+=(const string & str);
  // Funciones amigas
     friend bool operator==(const string &, const string &);
     friend ostream & operator<<(ostream &, const string &);
  private:
  // Datos
    char *datos; char *fin; int tam;
  // Funciones privadas
    void copiar(char *, const char *);
};
inline size_t string::size() const {return fin-datos;}
#endif
```

string.cpp

```
#include "string.h"
string :: string () :tam(1) {
    datos = new char;    *datos='\0'; fin = datos;
 }

string ::~ string ()  {      delete []  datos; }

string& string :: erase (size_t  pos,  size_t  len){
    ...
   return *this;
}  //  Param por defecto solo .h
string& string :: operator+=(const string & str){
    ...
   return *this;
}

bool operator==(const string & izq, const string &dch) {
  if  (  (izq . fin  – izq.datos) !=  (dch.fin –dcha.datos) ) return false;
   ....
 }

ostream & operator<<(ostream& os, const string& cad){
 for(int  i  = 0;  cad.datos[i ]  !=  '\0'; i++)
         os << cad.datos[i ];   //  Por ser  amiga tenemos acceso a datos privados
 return os;
}
```