Metodología de la Programación

Tema 4. Clases en C++ (Ampliación)

Sylvia Acid (acid@decsai.ugr.es)
derivado de las obras de Andrés Cano y de Juan Huete
Departamento de Ciencias de la Computación e I.A.

Curso 2025-26

Contenido del tema

A la hora de programar disponemos de datos de tipo:

1. elemental: int, double, char ...
2. avanzado standard STL: string, tree, set ...

A la hora de programar disponemos de datos de tipo:

1. elemental: int, double, char ...
2. avanzado standard STL: string, tree, set ...
3. diseñado por el programador

A la hora de programar disponemos de datos de tipo:

1. elemental: int, double, char ...
2. avanzado standard STL: string, tree, set ...
3. diseñado por el programador

Somos **usuarios** de los tipos 1 y 2 ej.:

```
int v; v=7+3;
```

A la hora de programar disponemos de datos de tipo:

1. elemental: int, double, char ...
2. avanzado standard STL: string, tree, set ...
3. diseñado por el programador

Somos **usuarios** de los tipos 1 y 2 ej.:

```
int v; v=7+3;
string s; s = "Hola" + " mundo";
```

Somos **diseñadores** y **programadores** del tipo 3
Además de ser **usuarios** de 3
¿Qué debería de tener un tipo propio para tener una larga vida?

Contenido del tema

Introducción: tipos de datos abstractos

Tipo de dato abstracto

Un **tipo de dato abstracto** (T.D.A.) es una colección de **datos** (posiblemente de tipos distintos) y un **conjunto de operaciones** de interés sobre ellos, definidos mediante una *especificación que es independiente de cualquier implementación* (es decir, está especificado a un alto nivel de abstracción).

Introducción: tipos de datos abstractos

TDA para polinomios

- Datos:
  - grado
  - coeficientes
  - ...

Introducción: tipos de datos abstractos

TDA para polinomios

- Datos:
  - grado
  - coeficientes
  - ...
- Operaciones:
  - sumar
  - multiplicar
  - derivar
  - ...

Introducción: tipos de datos abstractos

TDA para polinomios

- Datos:
  - grado
  - coeficientes
  - ...
- Operaciones:
  - sumar
  - multiplicar
  - derivar
  - ...

Algunos (datos/métodos) aparecen de forma natural y otros como herramientas auxiliares para facilitar la implementación o el uso....

TDA estring, lo llamaremos string con e

inspirado en *http://www.cplusplus.com/reference/string/*

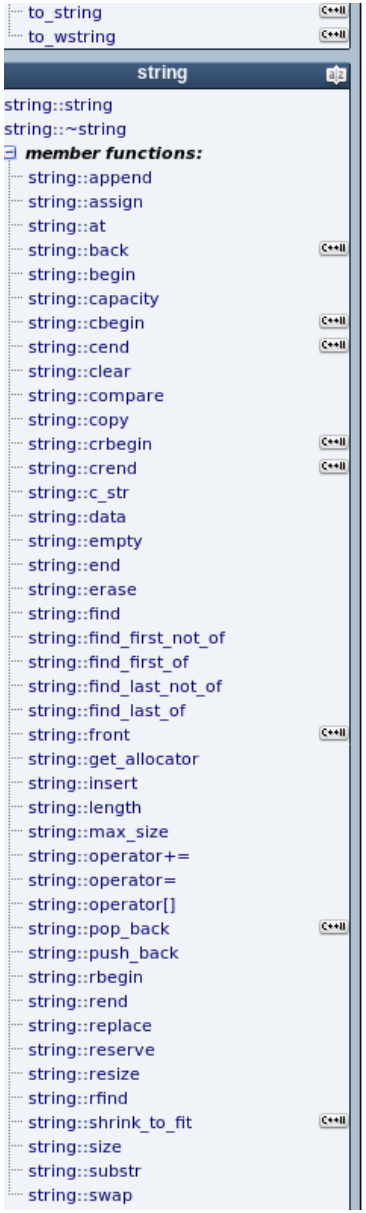
- Datos para estring:
  - char *datos;
  - char *fin;
  - tamagno
  - ...

TDA estring, lo llamaremos string con e

inspirado en *http://www.cplusplus.com/reference/string/*

- Datos para estring:
  - char *datos;
  - char *fin;
  - tamagno
  - ...
- Métodos/operaciones para estring:
  - comparar
  - concatenar
  - buscar
  - copiar
  - ...

...? cuáles más?



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 C++11	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 C++11	Access last character (public member function)
front C++11	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 C++11	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)
find_first_of	Find character in string (public member function)
find_last_of	Find character in string from the end (public member function)
find_first_not_of	Find absence of character in string (public member function)
find_last_not_of	Find non-matching character in string from the end (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

http://www.cplusplus.com/reference/string/

Introducción: tipos de datos abstractos

Implementación de un TDA

- **¿Cómo puede implementarse?**: struct y class son las herramientas que nos permiten definir nuevos tipos de datos abstractos en **C++**.

```
struct Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // OK
}
```

```
class Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // ERROR
}
```

Introducción: tipos de datos abstractos

Implementación de un TDA

- **¿Cómo puede implementarse?**: struct y class son las herramientas que nos permiten definir nuevos tipos de datos abstractos en **C++**.
- **Diferencias**: la principal, en struct, por defecto los datos miembro son públicos mientras que en class (por defecto) son privados.

```
struct Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // OK
}
```

```
class Fecha{
    int dia, mes, anio;
};
int main(){
    Fecha f;
    f.dia=3; // ERROR
}
```

Introducción

Abstracción de datos

- Podemos definir miembros privados en un struct, aunque no suele hacerse. Más adecuado usar class: por defecto los **datos miembro son privados**, se limita su acceso de forma predeterminada.

```
struct Fecha{
    private:
        int dia, mes, anio;
};
```

```
class Fecha{
    int dia, mes, anio;
};
```

Introducción

Abstracción de datos

- Podemos definir miembros privados en un struct, aunque no suele hacerse. Más adecuado usar class: por defecto los **datos miembro son privados**, se limita su acceso de forma predeterminada.

```
struct Fecha{
    private:
        int dia, mes, anio;
};
```

```
class Fecha{
    int dia, mes, anio;
};
```

Abstracción funcional

- Tanto las estructuras como las clases pueden contener métodos, aunque habitualmente las estructuras no suelen hacerlo.

Criterio para la elección:

- Los struct suelen usarse sólo para agrupar datos públicos.
- Si un struct necesita contener métodos usaremos class.

Introducción

- Los tipos de datos abstractos que se suelen definir con struct normalmente usan únicamente *abstracción funcional* (ocultamos los algoritmos, ya que los datos son públicos):

```
struct TCoordenada {
    double x,y;
};
void setCoordenadas(TCoordenada &c,double cx, double cy);
double getY(TCoordenada c);
double getX(TCoordenada c);

int main(){
    TCoordenada p1;
    setCoordenadas(p1,5,10);
    cout<<"x="<<getX(p1)<<", y="<<getY(p1)<<endl;
}
```

Introducción

- Los tipos de datos abstractos que se suelen definir con class usan además *abstracción de datos* (ocultamos la representación):

```
class TCoordenada {
    private:
        double x,y;

    public:
        void setCoordenadas(double cx, double cy);
        double getY();
        double getX();
};
int main(){
    TCoordenada p1;
    p1.setCoordenadas(5,10);
    cout<<"x="<<p1.getX()<<", y="<<p1.getY()<<endl;
}
```

TDA mediante class. Hacia la POO

Objetos creados con una clase

Un objeto ha de estar **siempre** en estado válido.

Puntos críticos de un objeto

1. Selección de los datos miembros de la clase, **sin dependencias funcionales**. (Diseño)
2. Ocultamiento de los datos miembros, solo se modifican a través de los métodos. (Diseño)
3. Creación de un objeto, en estado válido. (Implementación)
4. Control estricto de los métodos que modifican el estado de un objeto, pasan de un estado válido a otro estado válido, (Implementación).

Contenido del tema

La clase estring

El tipo **estring**, nuestro primer tipo diseñado.
Los datos y la funcionalidad que aparecen en la clase, están inspirados en el **string de la STL**.

TDA estring

Strings are objects that represent sequences of characters.

Para su implementación será necesario:

- seleccionar los datos miembro de la clase. **Su representación**.
- decidir los métodos que se van a poner a disposición. **Su funcionalidad**.

La clase estring

El tipo **estring**, nuestro primer tipo diseñado.
Los datos y la funcionalidad que aparecen en la clase, están inspirados en el **string de la STL**.

TDA estring

Strings are objects that represent sequences of characters.

Para su implementación será necesario:

- seleccionar los datos miembro de la clase. **Su representación**.
- decidir los métodos que se van a poner a disposición. **Su funcionalidad**.

```
char *datos; //inicio cstring
char *fin;   // siguiente al ultimo caracter
int tam;     // tamagno del bloque
```

Clase estring, repres

estring.h

representación

```
#ifndef __ESTRING__
#define __ESTRING__
class estring {
  public:
      .....
  private:
  //inicio c-string
    char *datos;
  // sig ultimo caracter
    char *fin;
  // tama bloque
    int tam;
};
#endif
```

```
estring 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"; //incr. 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
```

La clase estring

Implementación del TDA estring: métodos

Conviene seguir el siguiente orden:

- **constructores**: construyen un estring válido
- **operaciones naturales** sobre estring (públicos)
- **destructores**: liberan la memoria utilizada

Asumimos que, cuando se plantea un nuevo método debe incorporarse a la declaración de la clase (**estring.h**) y, a su implementación (**estring.cpp**)

otros métodos auxiliares, deberían ser **privados**.
El usuario de la clase ni los ve, ni los puede usar...

Los constructores de la clase

Inicializan de forma **adecuada** los datos miembro. Si contienen datos en memoria dinámica deben además, reservar la memoria necesaria.

Constructor por defecto

Es el **constructor sin parámetros**. Una clase lo puede tener bien por:

- El programador lo define explícitamente.
- El compilador lo crea implícitamente, cuando la clase **no tiene ninguno** definido por el programador.
  - Tal constructor no inicializa los datos miembro de la clase.
  - Cada dato miembro **no inicializado** contendrá un **valor indeterminado** (basura).
  - Llama al constructor por defecto de cada dato miembro que sea a su vez un objeto (de otra clase).

En `estring`, el constructor por defecto **implícito** provoca **error lógico** ya que, no genera objetos válidos, es necesario definir un constructor sin parámetros.

Constructores de la clase estring

Constructor sin parámetros

Debe asignar espacio a un estring nulo "".
Determinar los valores para 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

```
estring a;
```

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

Constructores de la clase estring

estring.h

representación

```
#ifndef __ESTRING__
#define __ESTRING__

class estring {
  public:
    estring();
      .....
  private:
    char *datos;
    char *fin;
    int tam;
};
#endif
```

estring.cpp

implementación

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

Constructores de la clase estring

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

```
estring 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.

Constructores de la clase estring

estring.h

representación

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

estring.cpp

implementación

```
estring::estring(size_t n, char c)
    {
  datos = new char[n+1];
  int i;
  for (i=0; ; i<n ;i++)
     datos[i] = c;

  datos[i] = '\0';
  fin = datos+n;
  tam = n+1;
}
```

Constructores de la clase estring

string (const char *s)-> *cplusplus/reference from c-string*

Copies the null-terminated character sequence (C-string) pointed by.

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

```
estring a("Hola"); // char [ ] == char * -> const char *
a.datos - > | H | o | l | a | \0 |
a.fin   ----------------------^
a.tam = 5
```

También se invoca con llamadas como:

```
char cad[10]="Adios";
estring b(cad);
```

Constructores de la clase estring

estring.h

representación

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

estring.cpp

implementación

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

Constructor de copia

- Es necesario un constructor de copia, para obtener una copia correcta de un objeto de la clase en otro objeto.
- 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.

Cuándo se invoca ¿?

1. Al hacer un **paso por valor** se hace una copia del parámetro actual en el formal.
2. Cuando se **devuelve un objeto** de la clase **por valor**.
3. Explícitamente por el programador.

Constructor de copia por defecto

- El compilador crea automáticamente un constructor de copia en caso de que no exista uno **explícito** del programador.
- Cada atributo será una copia "literal "de los atributos del objeto pasado como parámetro.
- Llama a los constructores de copia de todos los miembros almacenados en la clase (o estructura)

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.

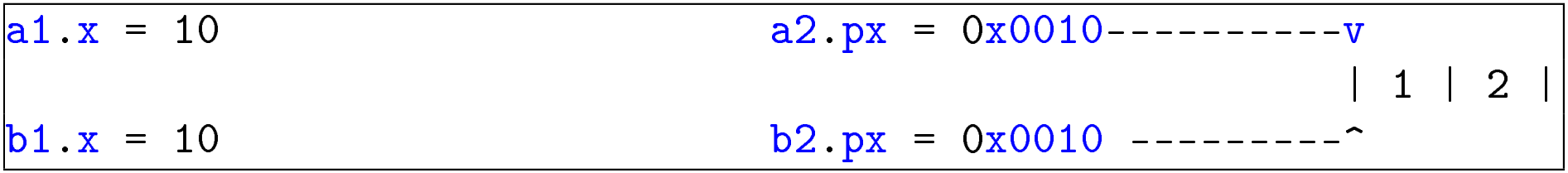
Esta copia (blanda) no tiene sentido cuando consideramos memoria dinámica.

Constructor de copia por defecto

Copia blanda

- Problema: Los dos objetos, el origen y la copia, comparten el mismo bloque de memoria del heap

```
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);   // <- Ctor copia
```



Constructor de copia por defecto

Copia blanda

- Problema: Los dos objetos, el origen y la copia, comparten el mismo bloque de memoria del heap

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

```
b1.x = 10                          b2.pX  ---------- > | 1 | 2 |
...
a1.x = 5;                          a2.pX[0] = 0;  // <- se modifica
    1 valor
```

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

Necesidad del constructor de copia

Para garantizar un control total sobre los objetos de la clase.

Copia dura

- Al hacer la copia, se copia el contenido en el heap, asociado al objeto
- Implica: reserva de memoria + 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.
- Obtenemos **dos objetos distintos** pero, con la misma información.

```
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 estring

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 estring `str`, incluyendo el `'\0'`
- `fin` apunta a la dirección donde está el `'\0'`
- `tam` debe tener el mismo tamaño de `str`

```
estring funcion( estring x ){  // x, parametro por copia
   estring cad;
    .....
   return cad; }// <<<<<<< 3  Llamada al constructor de copia
estring a("Hola");
estring 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 estring

estring.h

representación

```
#ifndef __ESTRING__
#define __ESTRING__

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

estring.cpp

implementación

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

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 funcion1(const estring a, double b){
  for (int .....)
}
int main(){
   estring a;
   funcion1(a,2.5); // llamada con estring,
                    // convierte (estring \&) -> (const estring \&)
   funcion1("Hola",3.8); // se hace casting implicito,
                         // (const char*) ->  (const estring &)
}
```

La clase estring en notación UML

Diseño de la clase (de momento)

Los datos miembro son privados $-$

Las funciones miembro públicas $+$ y privadas $-$.

La clase estring en notación UML

Diseño de la clase (de momento)

Los datos miembro son privados −
Las funciones miembro públicas + y privadas −.

estring
- char *datos - char *fin - int tam
+ estring() + estring (size_t n, char c) + estring (const char *s) + estring (const estring & s)

Destrucción automática de objetos locales

- 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, la memoria gestionada por el programador (en el heap) **no se libera**.

```
float funcionA( ){
  int v[4]={1,2,3,4};
  estring s("Hola
    mundo");
  ....
  return valor;
}
int main(){
  ...
  a = funcionA()
}
```

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

```
stack: [  ] mem liberada
"Heap:  dir       tipo bytes  valor
        0x0B00    pC   11x1   |H|o|...|\0|
queda bloqueado, perdems toda referencia"
```

Destructor de una clase

- Método que permite restituir toda la memoria asociada a un objeto, automatizando el proceso de liberación.
- El destructor, es único, y se declara con el nombre de la clase precedido por el símbolo $\sim$.
- No lleva parámetros y no devuelve nada. ej. $\sim$**estring( )**.
- Se ejecuta de forma automática, al finalizar el ámbito en el que el objeto está definido.
  - Los objetos locales a una función o trozo de código, justo antes de acabar la función o trozo de código.
  - Los objetos de ámbito global, justo antes de acabar el programa.

Si **no** hemos reservado memoria dinámica **no** hay que implementar el destructor, vale el implícito.

Sólo debemos de implementar el destructor si la clase reserva memoria dinámica, debiendo restaurar al sistema la memoria reservada.

Destructor de la clase estring

estring.h

representación

```
#ifndef __ESTRING__
#define __ESTRING__

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

estring.cpp

implementación

```
estring::~estring( ) {
   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 funcion1(){
  Prueba local;
  cout<<"funcion1()"<<endl;
}
Prueba varGlobal;
int main(){
  cout<<"Comienza main()"<<endl;
  Prueba ppal;
  cout<<"Antes de llamar a funcion1()"<<endl;
  funcion1();
  cout<<"Despues de llamar a funcion1()"<<
      endl;
  cout<<"Termina main()"<<endl;
}
```

```
Constructor      "// Construccion objeto varGlobal"
Comienza main()  "// Inicio ejecucion main"
Constructor      "// Construccion objeto ppal"
Antes de llamar a funcion1()
Constructor      "// Construcc objeto funcion1::local"
funcion1()        "// Ejecucion funcion1"
Destructor       "// Destruye objeto funcion1::local"
Despues de llamar a funcion1() "// 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

```
class alumno {
  private:
   estring nombre;//<-
     int *notas;
     int cuantas;
  public:
    alumno();
    ~alumno();
 };
alumno::alumno(){
  notas = new int[5];
  // for ...
  cuantas = 5;
}
alumno::~alumno(){
   delete []notas;
}
```

Constructor

Un constructor de una clase:

- Llama al constructor por defecto de cada miembro (estring,int*,int)
  nombre tendrá el estring por defecto ("")
- Ejecuta el cuerpo del constructor.
  (reserva memoria, inicializa, 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 estring nombre y cuantas)

Contenido del tema

Los otros métodos de la clase estring

Interfaz básico y adicional

A la hora de decidir qué métodos incluimos en la clase, debemos distinguir entre los que constituyen la **interfaz básica** y los que constituyen la **interfaz adicional**.

Los otros métodos de la clase estring

Interfaz básico y adicional

A la hora de decidir qué métodos incluimos en la clase, debemos distinguir entre los que constituyen la **interfaz básica** y los que constituyen la **interfaz adicional**.

- Los métodos de la interfaz básica:
  - Deberían ser **pocos**: definen la funcionalidad básica.
  - Deberían definir una interfaz **completa**.
  - Suelen utilizar **directamente los datos miembro** de la clase.
  - Hay métodos de consulta y métodos de mutación.

Los otros métodos de la clase estring

Interfaz básico y adicional

A la hora de decidir qué métodos incluimos en la clase, debemos distinguir entre los que constituyen la **interfaz básica** y los que constituyen la **interfaz adicional**.

- Los métodos de la interfaz básica:
  - Deberían ser **pocos**: definen la funcionalidad básica.
  - Deberían definir una interfaz **completa**.
  - Suelen utilizar **directamente los datos miembro** de la clase.
  - Hay métodos de consulta y métodos de mutación.
- Los métodos de la interfaz adicional:
  - Pueden ser métodos de la clase o funciones externas en el TDA.
  - Facilitan el uso del tipo de dato abstracto.
  - No deberían extenderse demasiado.
  - Aunque sean métodos, no es conveniente que accedan directamente a los datos miembro de la clase, ya que *un cambio en la representación del TDA* **implica** *cambiar todos los métodos adicionales*.

Consultores de una clase: Métodos const

- Un consultor es cualquier método que permite obtener información almacenada en un objeto de la clase.
  - `estring`: Devolver el tamaño de la cadena

  Un consultor debería tener prohibido modificar el objeto.
- C++ permite garantizar esta restricción y en caso de violarla resulta error de compilación.

Métodos const

Un método es constante, palabra reserva **const** al final de la declaración, evita cualquier intento de modificar los datos miembro en el método

```
class estring { ...
  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 estring {
2    size_t size() const;
3    size_t tama();  // Metodo imaginario
4  };
5  void pintaLongitud( const estring & 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
10 int main(){
11  estring a("Hola");
12  pintaLongitud(a);  // se le pasa referencia constante
13  cout << a.size() << endl; // OK conversion implicita a const
14  cout << a.tama() << endl; // OK ambos no const
15  const estring b("adios");
16  cout << b.size() << endl; // OK ambos son const
17  cout << b.tama() << endl; // ERROR b es un objeto constante
```

Consultores de la clase estring

- size(): Returns the length of the string, in terms of bytes.
- length(): Returns the length of the string, in terms of bytes. Equivalente a `size()`.
- capacity(): Returns the size of the storage space currently allocated for the string, expressed in terms of bytes.

estring.h

representación

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

estring.cpp

implementación

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

Métodos inline

Cuando un método es muy sencillo puede implementarse en la propia declaración de la clase como **método inline**.

- Los métodos **inline** se tratan de forma especial: no dan lugar a llamada a métodos (evitando el uso de la pila, etc). El compilador sustituye las llamadas al método por el bloque de sentencias que lo componen.
- Conviene limitar este tipo de métodos a aquellos que consten de pocas líneas de código.

Puede dejarse en el `estring.h`

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

Consultores de la clase estring

- 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 *** aseguramos que el estring NO se puede modificar desde fuera de la clase.

- c_str: Equivalente.

estring.h

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

estring.cpp

```
const char* estring::data()
    const {
    return datos;
}
const char* estring::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. Returns the position of the first character of the first match. If no matches were found, the function returns `string::npos`.

```
estring 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 estring& 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.

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

Modificadores de una clase

- Método que permite modificar la información almacenada en un objeto.
- Cuando tiene memoria dinámica un modificador puede hacer reajustes para liberar o alojar nueva información. Se recomienda realizar una copia dura del objeto.

  En estring: `push_back clear resize` (cambian el tamaño)

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

```
estring 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 estring: clear

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

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

estring.h

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

estring.cpp

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

Modificadores de la clase estring: push_back

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

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

```
1 estring 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
```

Modificadores de la clase estring: push_back()

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

```
1  void estring::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 }
```

Devolviendo referencias

Hasta ahora hemos devuelto objetos

```
estring estring::funcion(){..}
estring a(b.funcion());  // ok
```

Situaciones en las que podemos estar interesados en devolver una **referencia** al objeto en sí o a algún atributo del mismo

- Consultores y modificadores pueden devolver una referencia a algún atributo, lo que puede evitar copias innecesarias, por ejemplo `char & estring::at( )`.

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

- Muchos modificadores devuelven una referencia al objeto ya modificado, lo que permite encadenar llamadas a métodos (reduce llamadas a métodos)

Devolviendo referencias

Situaciones en las que podemos estar interesados en devolver una **referencia** al objeto en sí o a algún atributo del mismo

- Tanto consultores como modificadores pueden devolver una referencia a algún atributo, lo que puede evitar copias innecesarias
- Muchos modificadores devuelven una referencia al objeto ya modificado, por ejemplo `estring & estring::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)

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

1. no se percibe diferencia respecto de copia
2. encadenamiento de izda a dcha
3. encadenamiento flujo

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. **this** contiene la dirección del objeto al que referencia.
No hay que declararlo, C++ lo genera automáticamente

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

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

Al estar definido como T * const this, podemos cambiar el contenido al

Uso de this: cuándo?

- Para distinguir atributos de la clase de parámetros formales

```
void estring::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

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

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

```
bool estring::elMismo(estring & otro ){
   return (this == &otro);
}
estring a("hola"), b("hola");
ok = a.elMismo(b);  // F Objetos distintos
ok = a.elMismo(a);  // T Mismo objeto
```

Modificadores de la clase estring: erase()

```
estring & 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.

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

```
1 estring b("Hola mundo");
2 b.erase(5,3);
3 b.erase(0,estring::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 estring: erase()

estring.h

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

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

estring.cppp

```
1 estring & estring::erase(size_t pos, size_t len){
2   if(pos==0 && len==estring::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 }
```

Devolviendo referencias a objetos de una clase

- Cuando devolvemos una referencia a objeto de la clase hay que diferenciar entre dos posibles contextos en el que puede ser utilizado
  - Contexto Constante: El objeto (o $\forall$ atributo) no se pueden modificar, método **consultor**.

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

Devolviendo referencias a objetos de una clase

- Cuando devolvemos una referencia a objeto de la clase hay que diferenciar entre dos posibles contextos en el que puede ser utilizado
  - Contexto Constante: El objeto (o $\forall$ atributo) no se pueden modificar, método **consultor**.

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

  - Contexto NO Constante: Tenemos permiso para modificar el objeto, método **modificador**

```
void modifica(estring & s){
   s.push_back('a');  } // s es ref y puede modificarlo
int main(){
   estring a("aaa"), b(a); // a b no constante
   a.modifica(b);  // a b se modifican
```

Devolviendo referencias a atributos de una clase

Especializar el tipo de referencia considerando el contexto donde se puede utilizar el método

- **Contexto Constante:** Referencia devuelta y método deben ser declarados **constantes**
- **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;
```

Acceso seguro: accediendo a posiciones del estring

Dependiendo del contexto, nos puede interesar tener un "mismo" método con comportamiento distinto, actuando como **consultor** o **modificador**.
Ejemplo: el método at().
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 (whether pos is less than the string length), throwing an exception if it is not.

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

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

Acceso seguro: implementación at()

estring.h

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

estring.cpp

```
const char& estring::at (size_t pos) const{
   assert(0<=pos && pos<tam-1); // (1)
   return datos[pos];
}
char& estring::at (size_t pos){   // (2)
   assert(0<=pos && pos<tam-1);  // Si la expresion es false
                                 // muestra mensaje y aborta
   return datos[pos];
}
```

Contenido del tema

Funciones y clases amigas (friend)

Las funciones y clases amigas (`friend`) pueden acceder a la parte privada de otra clase.

¡Cuidado!

Deben usarse puntualmente, por cuestiones justificadas de eficiencia. No es conveniente usarlas indiscriminadamente ya que **rompen el encapsulamiento** que proporcionan las clases.

```
class A {
    private:
        ...
    public:
        ...
    friend class B;
        ...
    friend tipo funcion(parametros);
};
```

- B es una clase amiga de A.
  ⇒ Desde los métodos de B podemos acceder a la parte privada de **A**.
- `funcion()` es una función amiga de A.
  ⇒ Desde **funcion()** podemos acceder a la parte privada de **A**.

Funciones y clases amigas (friend): Ejemplos

```
class ClaseA {
    int x;
    ...
    public:
        ...
        friend class ClaseB;
        friend void func();
};


 void func() {
     ClaseA z;
     z.x = 6; // Acceso a z
     ...
 }
```

```
class ClaseB {
    ...
    public:
        void unmetodo();
};
void ClaseB::unmetodo() {
    ClaseA v;
    v.x = 3; // Acceso a v
    ...
}
```

Contenido del tema

Operadores y Clases. Nuevos problemas

Es lo mismo construcción por copia y asignación ¿?

- Ejemplos de uno y otro con objetos estring. Es correcta la asignación¿?

```
int main(){
estring s, t("Hola");  // Ctor
s = t;
estring u=v(" de la Mancha..."); //Ctor
u = t;
}
```

s

```
s.fin ---------------------------------v
s.datos 0x0010 -->| H | o | l | a | \0 |
t.datos 0x0040 -----^
t.fin ---------------------------------^
t.tam = 5
```

La asignación no es correcta¡!

Operadores y Clases. Nuevos problemas

- El problema de la asignación

```
int main(){
estring s, t("Hola");
s = t; ...}
```

Nuevo método `assign()`.
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.

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

Operadores y Clases. Nuevos problemas

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

- El problema de la asignación

```
int main(){
estring s, t("Hola");
s = t; // asignacion ...}
```

- Se transforma en:

```
s.assign(t); // asignacion ...}
```

- No sigue el estándar de asignación para los otros tipos, el operador =.
- Problema a la hora de diseñar/utilizar algoritmos, hay que diferenciar entre TDAs para la asignación.

Solución: la sobrecarga de operadores (proximamente...).

nuestro estring hasta el momento ...

estring
- char *datos - char *fin - int tam
+ estring() + estring (size_t n, char c) + estring (const char *s) + estring (const estring & s) + ~ estring() + void push_back (char c) + void clear() + void resize (size_t n, char c) + estring & erase (size_t pos = 0, size_t len = npos) + char & at (size_t pos) + const char & at (size_t pos) const + estring & assign (const estring & s) + estring & assign (const char* s, size_t n)

Cuestiones abiertas

- Escribe algunas razones para implementar un TDA mediante un class.
- Escribe algunas razones para no implementar un TDA mediante struct.
- Escribe más de 3 razones por las que es necesario definir un constructor sin parámetros explícito, y concretamente para clases con datos dinámicos.
- Escribe algunas razones por las que es necesario definir un constructor de copia, para clases con datos dinámicos. Porqué crees que el constructor de copia tiene un parámetro por referencia¿?
- Escribe alguna razón por la que es necesario definir un destructor para clases con datos dinámicos. Ilústralo con un ejemplo propio.

Cuestiones abiertas

- Implementar un método `assign()` para la clase `estring`.
- Asignación de objetos y constructor de copia, detalla las características comunes y diferencias entre ambas.
- Comprueba si existe o no el método `assign()` por defecto.
- Dado el siguiente programa, detalla a nivel de espacio, operaciones e incidencias de cada instrucción.

```
1 #include "estring.h"
2 estring estring::metodoM(){...}
3 main(){
4 estring a, b("oh la la");
5 a = b.metodoM();
6 }
```


creative
commons


BY NC SA