ALGORITMOS Y ESTRUCTURAS DE DATOS
SEMANA Nº 4

OBJETIVOS DE LA CUARTA SEMANA

Clase teórica

- Clases en C++ (continuación métodos: Declaración y definición, el cuerpo, llamadas a métodos el objeto actual (this), métodos especiales: sobrecargados, constructores y destructores).
- Problemas tipos.
- Funciones básicas de caracteres.

Clase práctica

- Ejercicios de clases con código y diagramas de flujo.

CLASES EN C++ (Continuación)

METODOS DE UNA CLASE

Las clases describen pues los atributos de los objetos, y proporcionan también los métodos. Un método es una función que se ejecuta sobre un objeto. No se puede ejecutar un método sin precisar el objeto sobre el que se aplica (salvo una excepción que veremos más adelante).

Los métodos de una clase definen las operaciones que un usuario puede realizar con los atributos de la clase. Desde el punto de vista de la POO, el conjunto de todas las funciones definen el conjunto de mensajes a los que los objetos de las citadas clases pueden responder.

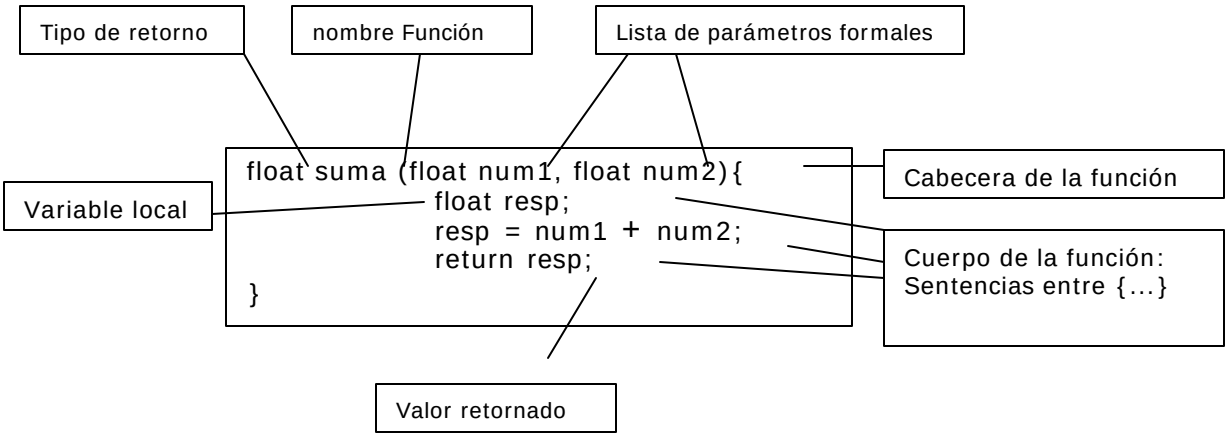
Ya que los métodos de una clase son funciones a continuación desarrollaremos el concepto general de una función en C++.

Funciones

1. Estructura

Una función es, sencillamente, un conjunto de sentencias que se pueden llamar desde cualquier parte de un programa. Su sintaxis:

tipoDeRetorno nombreFunción (lista de parámetros){ *cuerpo de la función }*



Una llamada a la función produce la ejecución de las sentencias del cuerpo de la función y un retorno a la unidad de programa llamadora después que la ejecución de la función se ha terminado; normalmente cuando se encuentra una sentencia return.

2. prototipos

C++ requiere que una función se declare o defina antes de su uso. La *declaración* de una función se denomina *prototipo.* Los prototipos de una función contienen la misma cabecera de la función, terminado en ;. No tienen cuerpo. Contiene los siguientes elementos: nombre de la función, una lista de argumentos encerrados entre paréntesis y un punto y coma. La inclusión del nombre de los parámetros es opcional.

Sintaxis: tipoRetorno nombreFunción (lista_de_declaración_parámetros);

Un prototipo declara una función y proporciona una información suficiente al compilador para verificar que la función está siendo llamada correctamente, con respecto al número y tipo de los parámetros y el tipo devuelto por la función. Es obligatorio poner un punto y coma al final del prototipo de la función con el objeto de convertirlo en una sentencia.

```
int minimo(Int v1, int v2, int v3);     //      prototipo válido
```

En C++, la diferencia entre los conceptos *declaración* y *definición* es clara. Cuando una entidad se *declara,* se proporciona un nombre y se listan sus características. Una *definición* proporciona un nombre de entidad y reserva espacio de memoria para esa entidad. Una *definición* indica que existe un lugar en un programa donde 'existe' realmente la entidad definida, mientras que una *declaración* es sólo una indicación de que algo existe en alguna posición.

Una *declaración* de la función contiene sólo la cabecera de la función y una vez declarada la función, la *definición* completa de la función debe existir en algún lugar del programa.

3. Parámetros

La lista de parámetros es la lista de nombres de variables separados por comas, con sus tipos asociados, que reciben los valores de los argumentos cuando se llama a la función.

Una función puede recibir como parámetros valores prácticamente de cualquier tipo. Estos parámetros serán usados por la función para realizar operaciones que lleven a la acción que se espera generar.

Los parámetros que va a recibir una determinada función habrá que indicarlos en la declaración, tras el identificador y entre paréntesis. La declaración propiamente dicha es similar a la que realizamos cuando definimos una variable, es decir, facilitaremos bs tipos y los nombres, usando la coma para separar unos parámetros de otros.

Características de la declaración de parámetros

- La declaración de los parámetros formales sigue una sintaxis similar a la del resto de los identificadores de C++. Por ejemplo:
  void funcion(**int a, float b**);
- En C++ los parámetros de las funciones pueden tomar *valores por defecto*, que son asignados por el compilador si no se suministran los argumentos actuales. Por ejemplo:
  void funcion(**int a=0, float b=7**);
- Si una función se declara sin parámetros, C++ la considera una función sin argumentos (void).
  void funcion();
- Los parámetros tienen el ámbito y duración de la función. Esto significa que los parámetros se podrán ver y utilizar sólo dentro de la función.

Parámetros por valor

Este método copia el valor de un argumento en el parámetro formal de la función. De esta forma los cambios realizados en los parámetros no afectan a la variable original utilizada en la llamada.

Por ejemplo:

```
class Ejemplo {
```

```
private:
	int valor;

public:
	void PorValor(int x)
	{
		x ++;			//modifica x
		valor = x;		//guarda x en valor
	}
};

void main()
{
	int aux;
	Ejemplo a;
	aux=10;
	cout << aux;			//muestra el valor de aux que es 10
	a.PorValor(aux)			//llama a PorValor de la clase ejemplo pasando el valor de
					//aux como parámetro
	cout << aux;			//muestra el valor de aux que es 10
}
```

En este ejemplo, el valor de la variable **aux** después de llamar al método PorValor no ha cambiado ya que la función recibió solo la copia de la variable pasada como parámetro.

Parámetros por referencia

La llamada por referencia es la segunda forma de pasar argumentos a una función. En este método se copia la dirección de memoria del argumento en el parámetro. Dentro de la función se usa esta dirección de memoria para acceder al contenido. Esto significa que los cambios hechos a los parámetros afectan a las variables usadas en la llamada a la función.

Para pasar un parámetro por referencia lo único que tendremos que hacer será disponer el carácter & delante del identificador, como si fuéramos a obtener la dirección de memoria de una variable. En una misma función podemos tener parámetros por valor y por referencia sin restricción alguna.

Por ejemplo:

```
class Ejemplo {

private:
	int valor;

public:
	void PorReferencia(int &x)
	{
		x ++;			//modifica x
		valor = x;		//lo guarda en valor
	}
};

void main()
{
	int aux;
	Ejemplo a;
	aux=10;
	cout << aux;			//muestra el valor de aux que es 10
	a.PorValor(aux)			//llama a PorReferencia de la clase ejemplo pasando el
					//valor de aux como parámetro
	cout << aux;			//muestra el valor de aux que es 11
}
```

En este ejemplo, el valor de la variable **aux** después de llamar a PorReferencia ha cambiado ya que la función recibió la referencia (dirección) de la variable pasada como parámetro y no su valor.

En este método el compilador pasa la dirección de memoria del valor del parámetro a la función. Cuando se modifica el valor del parámetro (la variable local), este valor queda almacenado en la misma dirección de memoria,

por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado. Para declarar una variable parámetro como paso por referencia, el símbolo & debe preceder al nombre de la variable.

4. llamadas

La llamada a una función se da cuando es referenciada con un conjunto de argumentos actuales que el sistema hace corresponder con los parámetros formales. El número de argumentos debe coincidir con los declarados en la función.

```
class ejemplo {
  private:
       int n;
  public:

       void Inicializar() {n = 10;}
       void Funcion1()           //funcion que muestra el valor de la variable n
       {  cout << n; }

       int Funcion2()                  //función que retorna el valor de la variable n
       {  return n; }

       void Funcion3(int x)            //función que muestra el valor del parámetro x
       {      cout << x; }

       int Funcion4(int x)             //función que retorna el valor del parámetro x
       {  x ++;   return x; }    //incrementado en 1
};


void main()
{
       Ejemplo a; int aux, aux2;
       a.Inicializar();
       a.Funcion1();                   //llama a Funcion1, como la función no
                                       //devuelve valor es una llamada simple
       aux = a.Funcion2();             //llama a Funcion2 y el valor que devuelve
                                       //se guarda en la variable aux
       cout << aux;                    //muestra el valor de aux que es 10
       a.Funcion3(aux)                 //llama a la Funcion3 pasando el valor de a
                                       //como parámetro
       aux2 = Funcion4(aux);           //llama a Funcion4 pasando el valor de aux
                                       //como parámetro y el valor que devuelve
                                       //se guarda en la variable aux2

       cout << aux;                    // muestra el valor de aux que es 10
       cout << aux2;                   // muestra el valor de aux que es 11
}
```

Declaración y definición de métodos

Generalmente la declaración de los métodos se realiza en el cuerpo de una clase, se define la cabecera de la función es decir el prototipo. Por ejemplo:

```
//Implementación de un contador sencillo
class Contador {
       int cnt;
public:
       int incCuenta();    //prototipos
       int getCuenta();
}
```

La definición de las funciones miembro es muy similar a la definición ordinaria ya conocida de función. Tienen una cabecera y un cuerpo y pueden tener tipos y argumentos. Sin embargo, tienen dos características especiales.

- Cuando se define una función miembro, se utiliza el operador de resolución de ámbito (::) para identificar la clase a la que pertenece la función.
- Las funciones miembro (métodos) de las clases pueden acceder a los componentes privados de la clase.

Supongamos una clase CLS con una función Fun que visualiza el contenido de los campos x e y. La función miembro f se puede declarar de dos modos:

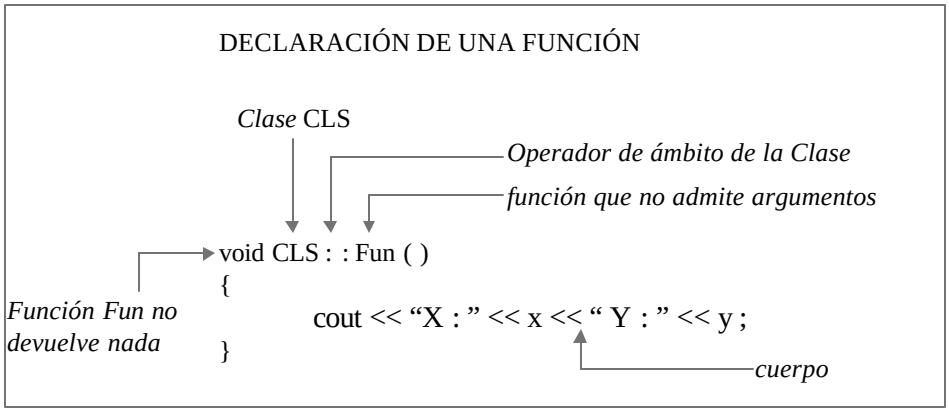
```
class CLS {
      int x :
      int y :
public :
      void Fun( ) { cout << "X : " << x << " Y : " << y ; }
};
```

O bien

```
class CLS {
      int x :
      int y :
public :
      void Fun( );
};
void CLS :: Fun ( )
{
      cout << "X : " << x << " Y : " << y ;
}
```

En el primer caso la función está declarada y definida dentro de la clase, en el segundo caso, la declaración está separada de la definición.

En el segundo caso se sabe que la función Fun es un miembro de la clase CLS ya que al definirla se ha indicado, mediante el *operador de resolución de ámbito* (::), que esa función pertenece a esa clase. En resumen, para indicar que una función pertenece a una clase, se añade como prefijo al nombre de la función el nombre de la clase a la cual está asociada la función miembro (el método). El *operador de resolución de ámbito* (::) separa el nombre de la clase del nombre de la función. Diferentes clases pueden tener funciones del mismo nombre y la sintaxis indica la clase asociada con la definición de la función.



Cuando la declaración de una función está separada de la definición, esta se puede realizar en el propio fichero fuente o en otro fichero fuente.

El cuerpo de los métodos

En Java dentro de los métodos pueden incluirse:

- Declaración de variables locales
- Asignaciones a variables
- Operaciones matemáticas
- Llamados a otros métodos

- Estructuras de control
- Excepciones (try, catch, que veremos más adelante) Las vemos????

Declaración de variables locales

Las variables locales en un método se declaran igual que en C:

```
Tipo NombreVariable [ = Valor];
```

Por ejemplo:

```
int    suma;
float  precio;
Contador laCuenta;
```

Las variables pueden inicializarse al momento de su creación:

```
int    suma=0;
float  precio = 12.3;
Contador laCuenta = new Contador() ;
```

Llamadas a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis, como se ve, entre otros, en el ejemplo siguiente:

Otra vez recordaremos la clase Contador:

```
//archivo ejemplo1.cpp
#include <iostream.h>

//Declaración de la clase
class Contador {
      int cnt;
public:
      Contador();
      int incCuenta();    //prototipos
      int getCuenta();
};

// Definición o Implementación de la clase
void Contador :: Contador() {
      cnt=0;       //inicializa
}
int Contador :: incCuenta() {
      cnt++;
      return cnt;
}
int Contador :: getCuenta() {
      return cnt;
}
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
void main() {
      Contador c;
      cout << c.incCuenta();
      cout << "El valor del contador es " << c.getCuenta();
}
```

las líneas en letra resaltada es un ejemplo de un llamado a un método de un objeto de otra clase. Notan que en este caso, es necesario llamar al método para un objeto (instancia) existente, por lo que se indica:

Nombre_del_Objeto<punto>Nombre_del_método(parámetros)

El objeto actual (puntero this)

Nunca se puede llamar una función miembro de una clase a menos que se asocie con un objeto (una instancia de la clase). ¿Cómo sabe una función miembro cuál es la instancia de una clase (el objeto específico) asociada con ella?.

El método utilizado por C++ es añadir un argumento extra oculto a las funciones miembro. Este argumento es un puntero al objeto de la clase que los enlaza con la función asociada y recibe un nombre especial denominado **this**.

Dentro de una función miembro, this apunta al objeto asociado con la invocación de la función miembro. Normalmente, el programador no necesita preocuparse por este puntero, ya que el lenguaje realiza la operación automáticamente transparente a las funciones miembro que la utilizan.

Dentro de una función miembro, se pueden hacer referencias a los miembros del objeto asociado a ella con el prefijo this y el operador de acceso ->. Sin embargo, este proceso no es necesario, ya que es redundante.

Si tomamos como ejemplo la siguiente clase:

```
public class Cliente {
       private:
        int codigo;
        float importe;
       public:
        int   getCodigo()  { return codigo; }
        float getImporte()       { return importe; }
        void setImporte(float x) { importe = x; }
};
```

Cuando escribimos en el método setImporte

```
importe = x;
```

para asignar un valor a importe, no tuvimos necesidad de indicar a qué objeto pertenecía. Cuando no se pone el objeto antes del atributo, se asume que la variable es un miembro del objeto, o es una variable local o parámetro. Si los nombres colisionan, como podría ser en el siguiente método

```
void setImporte(float importe)
{
       this.importe = importe;
}
```

usamos this para indicar al objeto actual. Esta palabra reservada siempre está apuntando al objeto actual, y puede usarse como en este caso, para resolver una ambigüedad, o puede usarse, por ejemplo, como parámetro en una llamada a una función para pasar un puntero al objeto asociado:

```
objeto.procesar(this);
```

Acceso al objeto mediante this :

- this -> *nombre miembro* apunta a un objeto miembro.
- this es la dirección del objeto apuntado.

Métodos especiales

Métodos sobrecargados

Tradicionalmente, un método o una función realizan una tarea específica para la que están programados. Java soporta la *sobrecarga de métodos,* lo que le permite definir versiones de éstos con el mismo nombre en una clase, siempre y cuando las versiones tengan diferentes firmas. Una firma incluye el nombre del método, el número, el orden y los tipos de sus parámetros formales. Como ejemplo simple, considere el reestablecimiento del saldo de cuenta de un objeto Account, esta clase (Account) ya tiene un método

```
double balanceo()
{
       return saldo;
}
```

que se utiliza para recuperar el saldo de la cuenta. Con la sobrecarga podemos definir un método:

```
void balanceo(double valor)
{
       saldo = valor;
}
```

que fija el saldo en una cantidad específica. El mismo método recuperará o establecerá el saldo, dependiendo de que se proporcione un argumento; ¡es muy útil!

Tenga en cuenta que, sólo ocurre la sobrecarga, cuando se utiliza varias veces el mismo nombre de método dentro de una clase. No hay un límite práctico para las versiones que es posible apilar en el mismo nombre de método.

Resolución de llamada a un método

Cuando se hace una llamada aun método sobrecargado C++ deduce automáticamente, a partir de los argumentos reales, la versión correcta del método que habrá de invocar. A esta actividad se le denomina *resolución de llamada,* C++ la realiza al seleccionar un método entre los accesibles *que son aplicables.*

Un método es *aplicable* si toma el mismo número de parámetros que los argumentos dados y cada uno de éstos puede transformarse por *conversión de invocación de método* al tipo del parámetro.

El compilador realiza la resolución de llamada de método al comparar el número y tipo de los argumentos reales con firmas de todos los métodos accesibles y elige un método aplicable que es el *más* especifico.

En resumen, las funciones sobrecargadas tienen el mismo nombre, pero deben tener un número diferente de argumentos o diferentes tipos de argumentos, o ambos. Por ejemplo:

```
void visualizar();
void visualizar(int cuenta);
void visualizar(int cuenta, int max);
```

Ejemplo:

Supongamos que tenemos una clase Media, que calcula la media de dos y tres números, enteros y reales. Para esto tendremos las siguientes funciones:

```
float media (float, float);      //calcula la media de dos valores tipo float
int media (int, int);            //calcula la media de dos valores tipo int
float media (float, float, float); //calcula la media de tres valores tipo float
int media (int, int, int);       // calcula la media de tres valores tipo float
```

Entonces:

```
class Media {
public:
      Media ();
      float Cal_Media (float a, float b)      { return (a+b)/2.0; }
      int Cal_Media (int a, int b)            { return (a+b)/2; }
      float Cal_Media (float a, float b, float c) { return (a+b+c)/3.0;}
      int Cal_Media (int a, int b, int c)     { return (a+b+c)/3; }
};

void main ()
{
      Media M;
      float x1, x2, x3;
      int y1, y2, y3;
      ...
      cout << M.Cal_Media (x1, x2);
      cout << M.Cal_Media (x1, x2, x3);
      cout << M.Cal_Media (y1, y2);
      cout << M.Cal_Media (y1, y2, y3);
}
```

Métodos constructores

Para cada clase, pueden definirse uno o más métodos particulares: son los constructores.
Un constructor es una función especial que sirve para construir o inicializar objetos.
En general:

- Tienen el mismo nombre de la clase que inicializa.
- No devuelven valores.
- Pueden admitir parámetros como cualquier otra función.
- Pueden existir más de un constructor, e incluso no existir.
- Si no se define ningún constructor de una clase, el compilador generará un constructor por defecto.
- Se llaman en el momento de la creación del objeto.

```
//archivo ejemplo1.cpp
#include <iostream.h>
```

```
//Declaración de la clase
class Contador {
       int cnt;
public:
       Contador();
       Contador(int c);
       int incCuenta();    //prototipos
       int getCuenta();
};

// Definición de constructores
void Contador :: Contador() {
       cnt=0;       //inicializa en 0
}
void Contador :: Contador(int c) {
       cnt = c;           //inicializa con el valor e c
}
//resto de la definición
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
void main() {
       Contador  c1, c2(10), c3(100);  // llamada al constructor

       cout << “El valor del contador1 es ” << c1.getCuenta();
       cout << “El valor del contador2 es ” << c2.getCuenta();
}
```

Cuando, desde una aplicación u otro objeto, se crea una instancia de la clase Contador, mediante la instrucción:

```
Contador c1;
```

el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto instanciado, ya que, una vez que fue creado, **no** puede "recrearse" a sí mismo).

En tiempo de compilación o ejecución, según corresponda, al encontrarse dicha instrucción, se reserva espacio para el objeto instanciado y se crea su estructura y en tiempo de ejecución se llama al método constructor.

La utilización de las instrucciones vistas, por ejemplo new(), entraña en efecto la creación física del objeto y la llamada a uno de sus constructores. Si hay varios constructores, estos difieren unos de otros por los parámetros que se les pasa mediante new().

Por ejemplo:

```
Include “iostream.h”

// una clase que tiene dos constructores diferentes
class Ejemplo {
public:
       Ejemplo (int param);
       Ejemplo (char *param);
};

Ejemplo :: Ejemplo (int param) {
       cout<< "Ha llamado al constructor\n";
       cout<< "con un parámetro entero";
}
Ejemplo :: Ejemplo (char *param) {
       cout<< "Ha llamado al constructor\n";
       cout<< "con un parámetro String";
}

// una clase que sirve de main
void main () {
       Ejemplo e1(2);
       Ejemplo e2("2");
}
```

da el resultado siguiente:

```
Ha llamado al constructor con un parámetro entero
Ha llamado al constructor con un parámetro String
```

Los constructores no tienen tipo de retorno. Atención, si por error definimos un constructor que tenga un tipo de retomo, el compilador lo considerará como un método normal. En ese caso tendremos la impresión de que el constructor no se llama en el momento de la creación del objeto. En realidad, se llamará a un constructor predeterminado, porque no habremos definido realmente un constructor.

Tipos de constructores

Constructores por defecto

El constructor por defecto es un constructor que no acepta argumentos. Por ejemplo:

```
class Punto {
        private:
                int x;
                int y;
        public:
                Punto()
                {       x = 0;
                        y = 0;
                }
};
```

Para crear objetos usando este constructor se escribiría:

```
        void main()
        {
                Punto p1;
                …
        }
```

Constructores con argumentos

El constructor con argumentos, como su nombre lo indica posee argumentos. Por ejemplo:

```
class Punto {
        private:
                int x;
                int y;
        public:
                Punto(int x, int y)
                {       this.x = x;
                        this.y = y;
                }
};
```

Para crear objetos usando este constructor se escribiría:

```
        void main()
        {
                Punto p2(2, 4);
        }
```

Constructores copiadores

Un constructor que crea un objeto a partir de uno existente se llama constructor copiador o de copias. Es un constructor que toma como único parámetro otro objeto del mismo tipo. El constructor de copia tiene sólo un argumento: una referencia a un objeto de la misma clase. Por ejemplo:

```
class Punto {
        private:
                int x;
                int y;
        public:
                Punto(Punto p)
                {       x = p.x;
                        y = p.y;
                }
```

```
};
```

Para crear objetos usando este constructor se escribiría:

```
void main()
{
      Punto p2(2,4);                  //del ejemplo anterior
      Punto p3 = p2;            //p2 sería el objeto creado en el
                                //ejemplo anterior
}
```

Métodos destructores

Es una función miembro con igual nombre que la clase, pero precedido por un carácter tilde (~). Realiza la función opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean.

Las funciones destructor tienen las siguientes propiedades:

- El nombre del destructor es el mismo de la clase, precedido por una tilde (~).
- No pueden devolver ningún tipo de retorno.
- No pueden tener argumentos.
- Cada clase tiene como máximo un destructor.
- El compilador llama automáticamente a un destructor del objeto cuando el objeto sale fuera de ámbito.

Las diferencias esenciales entre constructores y destructores son :

- Los destructores pueden ser virtuales, los constructores no pueden.
- Los destructores no se pueden pasar como argumentos.
- Sólo se puede declarar un destructor para una clase dada.

El siguiente ejemplo muestra una clase String con dos constructores y un destructor:

```
class String {
      int lon;
      char *cadena;
public :
      String (char *s);                     //constructor
      String (String &s);                   //constructor
      ~ String ()  {delete [] cadena; }     //destructor
      //otros miembros
};

//definición de funciones miembro

main () {
  String Saludos (“Hola Mundo”);
  //procesar cadenas
}
```

La instancia Saludos invoca al primer constructor para asignar el espacio del montículo para almacenar la cadena “Hola mundo”. Cuando la función main () termina, el destructor se llama automáticamente para recuperar el espacio del montículo creado por la instancia saludos.

PROBLEMAS TIPOS

Manejo de secuencias

En la mayoría de los problemas que se plantean se utilizan secuencias de elementos, ya sean números, caracteres o entidades como cliente, alumno, producto, etc. En esta clase vamos a analizar los distintos problemas que se pueden plantear y las posibles soluciones.

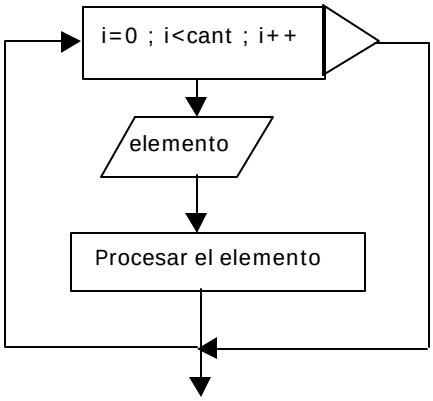
Carga de la secuencia

La carga implica ingresar los elementos desde teclado uno por uno, dicha carga está inserta en un ciclo o bucle cuya finalización dependerá del enunciado del problema.
Se pueden presentar dos casos:

1. Que se conozca el número exacto de elementos a cargar, por ejemplo Leer 100 números.
2. Que se desconozca el número de elementos a cargar, pero que se sepa que la finalización de la carga depende de una condición, que generalmente depende del dato de entrada, por ejemplo leer un número mientras sea distinto de 0.

Para resolver esto generalmente se aplica el siguiente diagrama:

Para el caso 1.

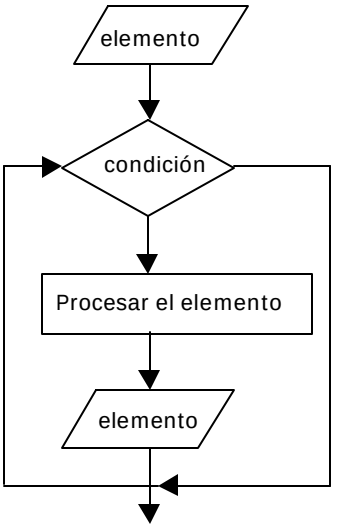


Cant es la cantidad maxima de elementos a ingresar

La condición del ciclo dependerá de cómo se inicialice la variable de control, siempre hay que tener en cuenta que el ciclo debe realizar cant iteraciones, por ejemplo:

si i = 0 entonces la condición de fin será i < cant max
si i = 1 entonces la condición de fin será i <= cant max
si i = 1 entonces la condición de fin será i <= cant max+1

Para el caso 2



La condición depende de el elemento ingresado, por ejemplo elemento !=0

Se actualiza el elemento antes de evaluar nuevamente la condición

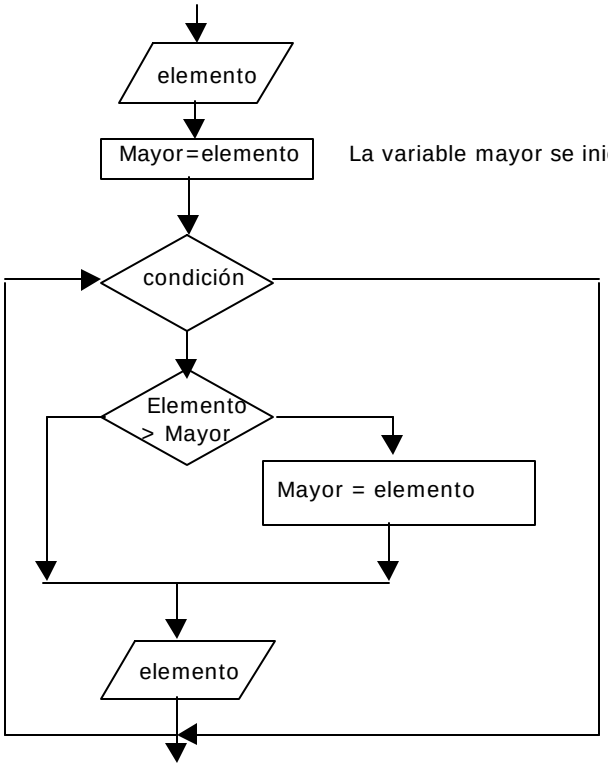
Procesar todos los elementos de la secuencia

1. Listar, sumar, acumular, etc. todos los elementos de la secuencia.
2. Calcular y mostrar totales, promedios, porcentajes, etc.
3. Buscar mayores y menores.

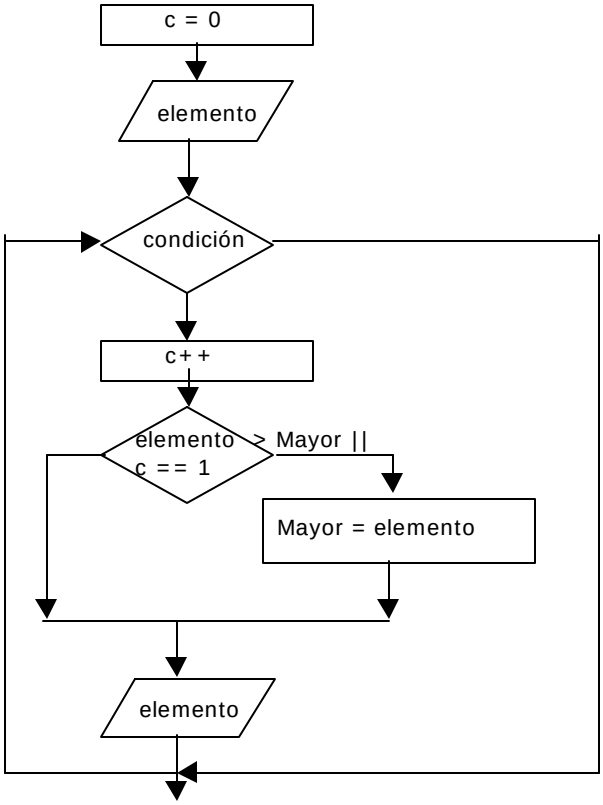
Para el caso 3
Para buscar mayores y menores en una serie hay que respetar los siguientes pasos:

1. Definir una variable auxiliar para guardar el valor mayor o menor de la serie.
2. Inicializar dicha variable con el primer valor de la serie
3. En el ciclo comparar cada elemento de la secuencia con esa variable, por ejemplo si se esta buscando un mayor: elemento > mayor
4. Si esto se cumple actualizar la variable mayor con el valor del elemento.

Un posible diagrama de una búsqueda de mayor…



o bien…

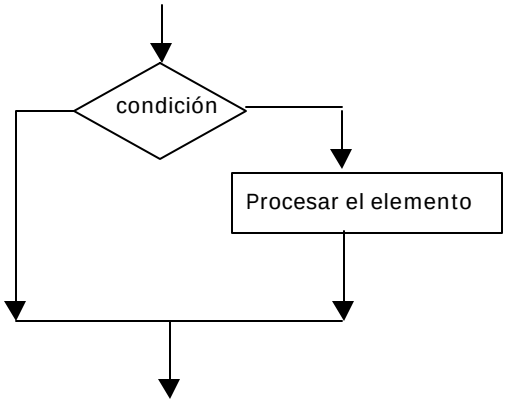


La variable c cuenta la cantidad de elementos de la serie.

La variable mayor se inicializa con el primer valor de la serie o con aquel valor mayor encontrado.

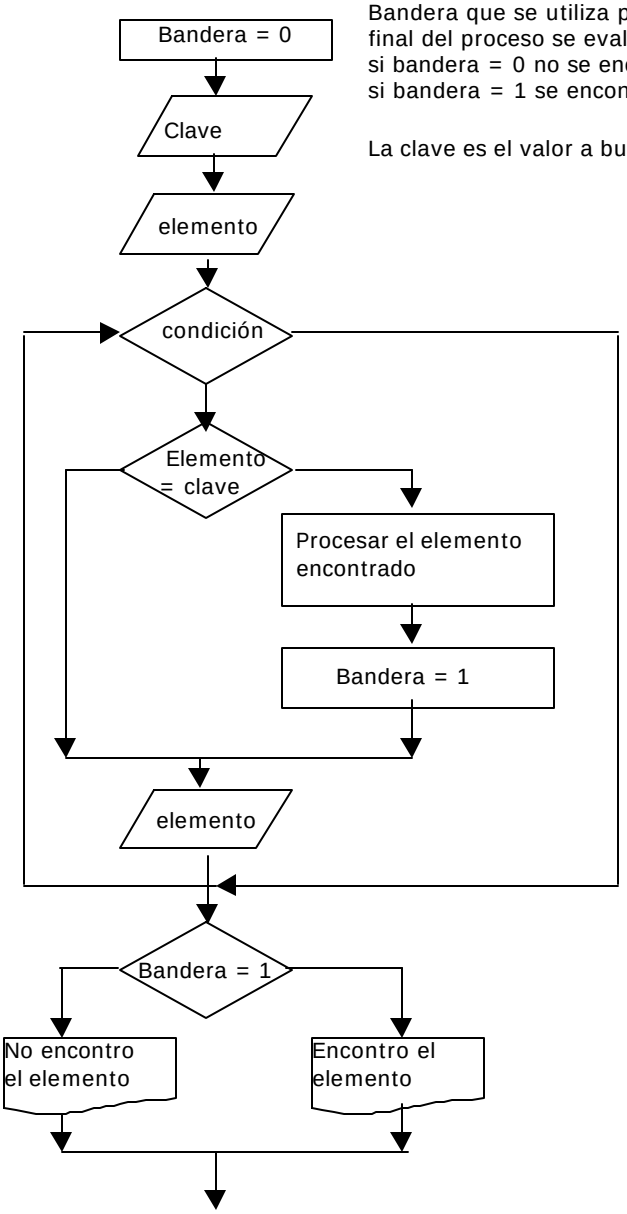
Procesar un elemento en particular de la secuencia

1. Listar, sumar, acumular, etc. un elemento según una condición.
2. Buscar un elemento según una clave de búsqueda.



Si se cumple una condición se procesa el elemento, por ejempo si el elemento es > 10 mostrarlo

Para el caso 2



Bandera que se utiliza para saber si se encontro el elemento, al final del proceso se evalua esta variable:
si bandera = 0 no se encontro el elemento
si bandera = 1 se encontro el elemento.

La clave es el valor a buscar

Si el elemento es igual a la clave

Secuencias numéricas

Enunciado común: Leer una serie de números mientras el número sea distinto de un valor.

Procesos en común:

- Contar o acumular todos los números o los que cumplan con una condición (por ejemplo: todos los pares, impares, mayores que un valor, etc.)
- Mostrar posiciones determinadas.
- Calcular porcentajes, promedios, etc.

Se utilizan contadores, acumuladores, banderas y variables auxiliares.

Secuencias de caracteres

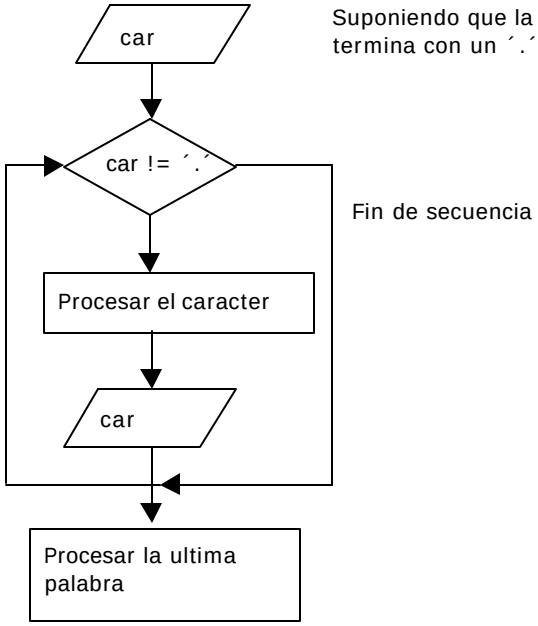
Enunciado común: Leer una serie de caracteres mientras no sea fin de texto o secuencia, dicho fin de secuencia puede ser cuando el carácter leído sea ., *, .-, etc.

Control de secuencia

El control de la secuencia está regulado por la condición del enunciado y depende del carácter leído.

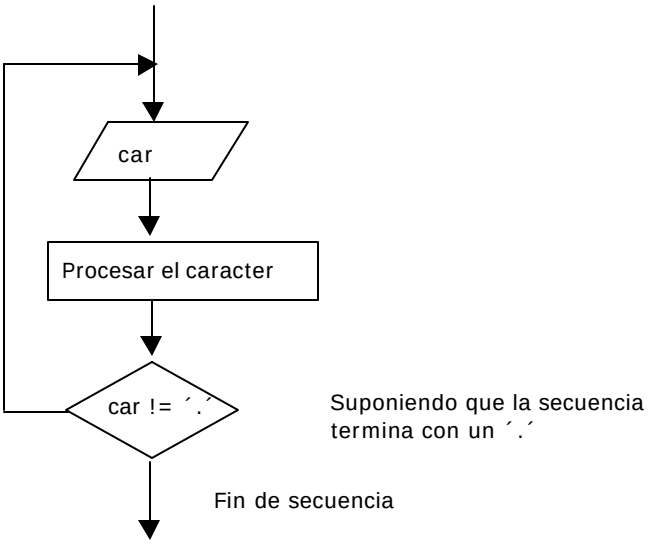
Formas de leer las secuencias.

Usando un ciclo while

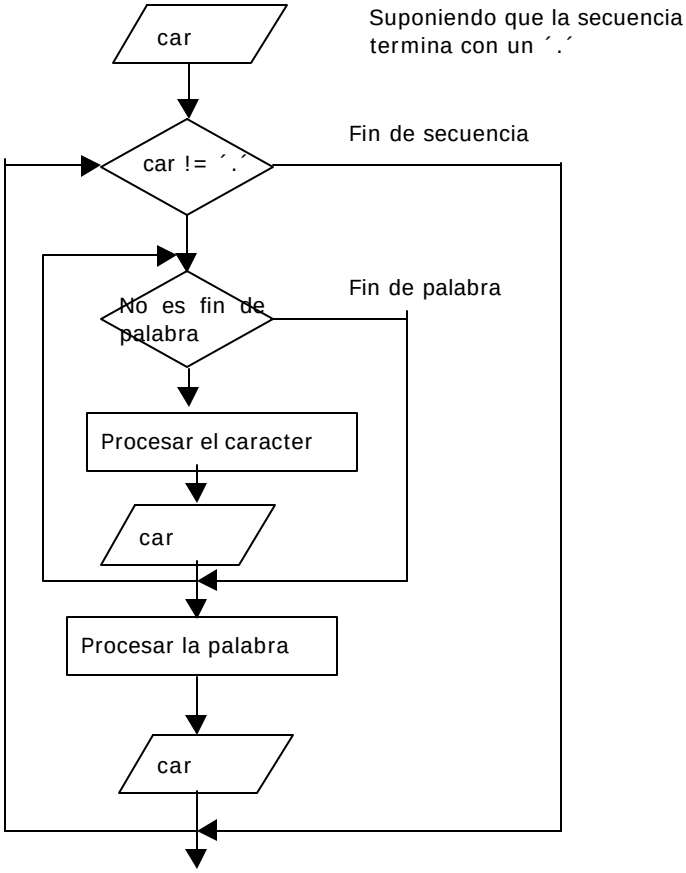


Este método no es muy recomendado, porque si la última palabra terminó con el carácter de fin de secuencia no se procesa en el ciclo, por lo tanto se debe procesar la última palabra al finalizar la secuencia, repitiendo el código por fin de palabra.

Usando un do … while



Usando un ciclo while para manejar la secuencia y otro ciclo while interno para manejar los fin de palabra.



Lectura de un carácter: Se puede utilizar la función getche(), que permite la entrada de un carácter, lo muestra por pantalla y continua sin esperar un retorno de carro.

Por ejemplo: car = getche();

Control de un carácter: se utiliza una variable (car) para guardar el valor del carácter actual
Por ejemplo: si car = ´a´ significa que el caracter leido es una a

Control de expresiones: se utilizan variables para guardar los caracteres leidos y se necesitan tantas como caracteres se necesiten procesar. Como se lee carácter a carácter siempre se tiene en la variable car el carácter actual leido y no se sabe cuales fueron los caracteres anteriores leidos, para solucionar esto hay que manejar tantas variables auxiliares como se necesiten, que guarden el valor de los caracteres leidos anteriormente.

Ej:

Si se tiene que evaluar la expresión ab, se utiliza el caracter actual (car) y el carácter anterior (carant)

```
si (carant = ´a´ && car = ´b´)
        ocurrio ´ab´
```

Si se tiene que evaluar la expresión abc, se utiliza el caracter actual (car), el carácter anterior (carant) y el carácter anterior del anterior (carant1)

```
si (carant = ´a´ && car = ´b´ &&      car = ´c´)
        ocurrio ´abc´
```

Control de palabras: Utiliza dos variables caracter actual y carácter anterior

Comienzo:
si (carant no es letra) y (car actual es letra)
Fin:
si (carant es letra) y (car actual no es letra)

Ejercicio

Se carga un texto por teclado carácter a carácter, fin del texto ´.´ se pide

1. Contar la cantidad de veces que ocurre la expresión ´la´ en el texto.
2. Contar la cantidad de palabras que empiezan con ´s´.
3. Contar la cantidad de palabras que terminan con ´r´.
4. Contar la cantidad de palabras que tienen la expresión ´se´.
5. Contar la cantidad de palabras que tienen mas de dos vocales.
6. Porcentaje de palabras del punto 2.
7. Longitud de la palabra mas corta.
8. Promedio de palabras.

FUNCIONES ESTANDAR PARA TRATAR CARACTERES

Para imprimir un caracter en pantalla :

putchar() : Escribe su argumento de caracter en la pantalla en la posición actual del cursor. y este avanza un espacio. El archivo de cabecera es stdio.h.

Para leer un caracter del dispositivo del teclado :

getche() : Espera hasta pulsar una tecla y después devuelve su valor. La función hace eco de la tecla pulsada automáticamente en la pantalla, es decir muestra su valor en la pantalla. El archivo de cabecera es conio.h.

getch() : Una variación de getche() que no hace eco de los caracteres tecleados. El archivo de cabecera es stdio.h.

getchar() : Otra variación de getche() que almacena la entrada hasta que se presiona la tecla enter. El archivo de cabecera es stdio.h.

// Secuencia ascendente de caracteres

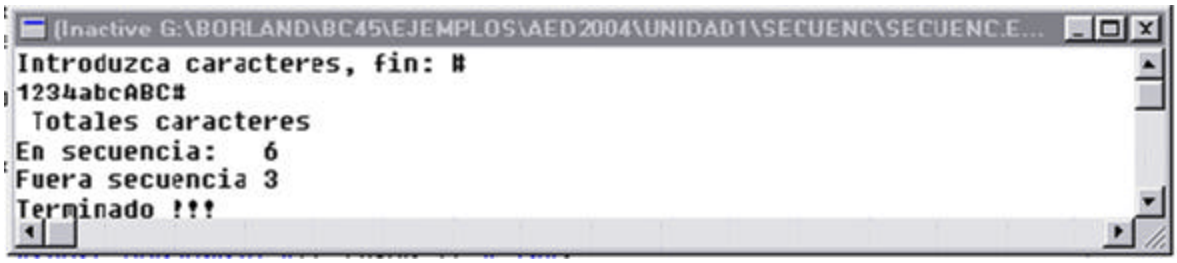
Enunciado: Procesar una secuencia de caracteres. La secuencia termina cuando leemos el carácter '#'. Contar cuantas secuencias y "fuera de secuencia" tenemos.
Lo primero es **definir el concepto**: tenemos una secuencia ascendente cuando el carácter actual es mayor o igual que el anterior. En cambio, si nuestro carácter actual es menor que el anterior, tenemos un "fuera de secuencia". (Usaremos la convención: si el carácter actual está fuera de secuencia, no lo consideraremos como anterior del siguiente)

A B F C B #
Son secuencias ascendentes AB, BF, y nada mas.
Son fuera de secuencia FC, FB.

Que precisamos como atributos del objeto secuencia ? El carácter actual y el anterior. Y dos contadores. Y que comportamiento debemos codificar ? Si tenemos una secuencia, tenemos un ciclo. Si optamos por dejar el ciclo en main() (Decisión arbitraria), nos tenemos que preocupar por la primera lectura, la detección de secuencia, mostrar el objeto ... Una posible solución, y puede haber muchas otras, incluso mejores:

```
# include <iostream.h>
class Secuenc{
private:
        char carAct, carAnt;     // Caracteres actual, anterior
```

```
        int enSec, fuSec;        // Contadores
        int priLect();           //  Es primera lectura ? Necesitamos saberlo,
public:
        Secuenc();               // Constructor
        void leeCar(){carAct=cin.get();} // Inicializa carAct
        void contSec();                  // Controla secuencia
        void toString();         // Visualizando el objeto
        int finDatos();          // Es fin de datos ?
};
Secuenc::Secuenc(){      // Constructor
        enSec=0;fuSec=0; carAnt=' ';
cout << "Introduzca caracteres, fin: # \n";}
void Secuenc::contSec(){
        if(priLect())carAnt=carAct; // Si es primera lectura,  NO controlamos:
        else if (carAct>=carAnt){enSec++;carAnt=carAct;} // En secuencia
                else fuSec++;       // Fuera de secuencia
}
void Secuenc::toString(){
        cout << " Totales caracteres \n";
        cout << "En secuencia:   "<< enSec << " \n";
        cout << "Fuera secuencia "<< fuSec << " \n";
        cout << "Terminado !!!\n";
}
int Secuenc::finDatos(){int finDat=0; if(carAct=='#') finDat=1;
return finDat;}
int Secuenc::priLect(){
        int priLect=0;       // Suponemos que no es primera lectura
        if(carAnt == ' ') priLect=1; // Es la primera ...
        return priLect;
}
void main(){
        Secuenc sec;    // Construimos el objeto sec de la clase Secuencia
        sec.leeCar();            // Leemos primer caracter
        while(!sec.finDatos()){ // Mientras no sea fin de datos
                sec.contSec();           // Controlamos, contabilizamos
                sec.leeCar();    // Leo siguiente caracter
        };
        sec.toString();          // Visualizamos el objeto
        };
```



// Términos en serie aritmética

Enunciado: Procesar una secuencia de números, terminados en 999. Se quiere saber cuantos de sus términos satisfacen una serie aritmética.
Recordemos que en una serie aritmética cada término difiere del anterior en un valor denominado razón. Usando el mismo criterio del ejemplo anterior, si un término de la secuencia no es parte de la serie, no lo consideraremos anterior del siguiente ...

En el siguiente ejemplo, usando una razón 3,

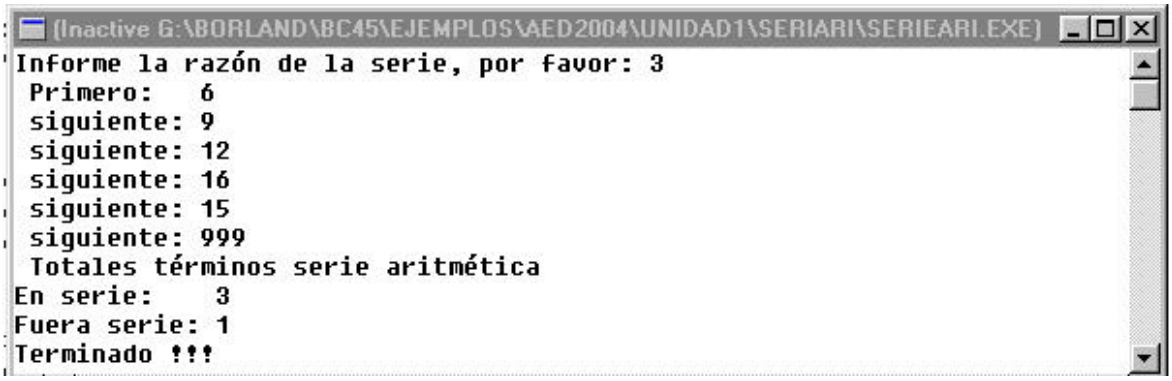
```
2  5  7  8  11  999
```

No contamos el 2, por ser el primer término (Decisión arbitraria, podríamos optar por contarlo). Entonces, términos de la serie son: 5 8 11, nada mas.

Cuales son los atributos del objeto SerieAri ?. La razón, los términos actual y anterior, un par de contadores. Y cual su comportamiento ? Como en el caso anterior, optamos por dejar el ciclo de lectura en el main(). Tendremos un constructor, la detección de términos en serie, la exhibición de resultados. Como el ciclo lo hacemos en el main y los datos los definimos private, necesitaremos comportamiento para leer y detectar fin de datos. Una solución posible :

```
# include <iostream.h>
class SerieAri{
private:
        int razon, serAct, serAnt, enSer, fuSer;  // Datos miembro del objeto
public:
        SerieAri();             // Constructor
        void lectura();
        void procSer();         // Procesa nuevo término
        void toString();        // Visualizando el objeto
        int finDatos();         // Es fin de datos ?
};
SerieAri::SerieAri(){       // Constructor
        enSer=0; fuSer=0;
        cout << "Informe la razón de la serie, por favor: ";
        cin >> razon;
        cout << " Primero:  ";
        cin >> serAnt;          // Primer término
}
void SerieAri::lectura() {cout << " siguiente: "; cin >> serAct;}
void SerieAri::procSer(){
        if(serAct-serAnt==razon){
                enSer++; serAnt=serAct;       // Es término de la serie;
        } else fuSer++;         // No lo es ...
}
void SerieAri::toString(){
        cout << " Totales términos serie aritmética \n";
        cout << "En serie:    "<< enSer << " \n";
        cout << "Fuera serie: "<< fuSer << " \n";
        cout << "Terminado !!!\n";
}
int SerieAri::finDatos(){
        int finDat=0; if(serAct==999) finDat=1;
        return finDat;
}

void main(){
        SerieAri ser;     // Construimos el objeto ser
        ser.lectura();            // Leemos 2do término
        while(!ser.finDatos()){ // Mientras no sea fin de datos
                ser.procSer();          // Procesamos
                ser.lectura();          // Términos siguientes
        }
        ser.toString();           // Visualizamos el objeto
};
```



// **Tratamiento de caracteres** -

Enunciado: Procesar una secuencia de caracteres, terminados en '#'. Se quiere conocer cuantos son letras, de ellas cuales son minúsculas, mayúsculas, vocales, consonantes, etc. Los caracteres pueden ser dígitos decimales, hexadecimales. Tener en cuenta que un carácter puede ser miembro en mas de un grupo simultáneamente.

Cual es el concepto que estamos modelando ?. Las diversas clasificaciones del carácter, ya enunciadas. Que atributos necesitamos ? El carácter, y los contadores de los diversos conjuntos.

Nuevamente tenemos una secuencia, por consiguiente un ciclo, que dejaremos en el main(). En el comportamiento debemos tener funciones miembro capaces de discriminar a que conjunto el carácter pertenece. Lo podemos resolver mediante arrays de caracteres. Una posible solución sigue.

```
# include <iostream.h>
class Caracter{
private:                        // Parte interna de la clase
        char car;               // Nuestro caracter
        int cLetras;    // Contador de letras
        int cLetMin;    // Minusculas
        int cLetMay;    // Mayusculas
        int cVocal;     // Vocales
        int cConso;     // Consonantes
        int cDigDec;    // Dígitos decimales
        int cDigHex;    // Dígitos hexadec.
        int cSigPun;    // Signos de puntuacion
        int esLetra();  // Retorna 1 minúscula, 2 mayúscula, 0 no es letra
        int esVocal();  // Retorna 1 si vocal, 0 no es vocal
        int esConso();  // Retorna 1 si consonante, 0 si no es
        int esLetMay(); // Retorna 1 si es letra mayuscula
        int esLetMin(); // Retorna 1 si es letra minúscula
        int esDigDec(); // Retorna 1 si es dígito decimal
        int esDigHex(); // Retorna 1 si es dígito hexadecimal
        int esSigPun(); // Retorna 1 si es signo puntuación
public:                         // Parte publica, interface con el usuario: main()
        Caracter();             // Constructor
        char leeCar();  // Inicializa car mediante lectura y lo retorna
        void setCar(char car);  //  Inicializa car mediante parámetro
        char getCar();          // Retorna car
        int  finDatos(); // Retorna 1 si se digita #
        void cuenta();          // Contabiliza los distintos tipos
        void toString();        // Exhibe resultados
};

Caracter::Caracter(){  // Constructor
car=' ';
cLetras=cLetMin=cLetMay=cVocal=cConso=cDigDec=cDigHex=cSigPun=0;
```

```
cout << "Introduzca caracteres, fin: # \n";
}

char Caracter::leeCar(){ // Todo lo digitado se almacena en buffer del teclado
        car=cin.get(); // pareciendo en pantalla. Al <Enter> se produce la primera
        return car;    // lectura, y se necesita de un ciclo para las siguientes.
}               // Cada caracter leido es almacenado en car y tambien retornado

void Caracter::setCar(char cara){
        car = cara;          //  Inicializamos el atributo car
}

char Caracter::getCar(){
        return car;          //  Retornamos el atributo car.
}


int Caracter::esLetra(){
        int letra=0;
        if(esLetMay()) letra=1;
        if(esLetMin()) letra=1;
        return letra;              //  Retornemos lo que haya sido
}

int Caracter::esVocal(){
        int vocal=0;             // Inicialmente suponemos que no es vocal
        char voca[10]="aeiouAEIOU";
        for(int i=0;i <= sizeof(voca);i++)
                if(voca[i]==car) vocal=1;  //  Es una vocal
        return vocal;              //  Retornamos lo que sea ...
}

int Caracter::esConso(){
        int conso=0;
        if(esLetra() && !esVocal()) conso=1;
        return conso;
}

int Caracter::esLetMay(){
        int letra=0;     //  Inicialmente suponemos que no es letra minuscula
        char mayu[26]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        for(int i=0;i <= sizeof(mayu);i++)
                if(mayu[i]==car) letra=1;  //  Es una letra mayúscula
        return letra;
}

int Caracter::esLetMin(){
        int letra=0;
        char minu[26]="abcdefghijklmnopqrstuvwxyz";
        for(int i=0;i < sizeof(minu);i++)
                if(minu[i]==car) letra=1;  //  Es una letra minúscula
        return letra;
}

int Caracter::esDigDec(){
        int digDec=0;
        char dig10[10]="1234567890";
        for(int i=0;i <= sizeof(dig10);i++)
                if(dig10[i]==car) digDec=1;
        return digDec;
}
```

```
int Caracter::esDigHex(){
        int digHex=0;
        char dig16[16]="1234567890ABCDEF";
        for(int i=0;i <= sizeof(dig16);i++)
                if(dig16[i]==car) digHex=1;
        return digHex;
}
int Caracter::esSigPun(){
        int sigPun=0;
        char punct[04]=".,;:";
        for(int i=0;i <= sizeof(punct);i++)
                if(punct[i]==car) sigPun=1;
        return sigPun;
}
int Caracter::finDatos(){
        int finDat=0;
        if(car=='#') finDat=1;
        return finDat;
}
void Caracter::cuenta(){
        if(esLetra()) cLetras++;        if(esLetMin())cLetMin++;
        if(esLetMay())cLetMay++;        if(esVocal ())cVocal ++;
        if(esConso ())cConso ++;        if(esDigDec())cDigDec++;
        if(esDigHex())cDigHex++;        if(esSigPun())cSigPun++;
}
void Caracter::toString(){
        cout << " Totales =\n";
        cout << "cLetras: "<< cLetras << " \n";
        cout << "cLetMin: "<< cLetMin << " \n";
        cout << "cLetMay: "<< cLetMay << " \n";
        cout << "cVocal : "<< cVocal  << " \n";
        cout << "cConso : "<< cConso  << " \n";
        cout << "cDigDec: "<< cDigDec << " \n";
        cout << "cDigHex: "<< cDigHex << " \n";
        cout << "cSigPun: "<< cSigPun << " \n";
        cout << "Terminado !!!\n";
}
```


```
void main(){
        Caracter kar; // Definimos el objeto, clase Caracter
        kar.leeCar();     // Leemos primer caracter
        while(!kar.finDatos()){//         Ciclo de lectura y conteo
```

```
                kar.cuenta();       //  Contamos los distintos tipos
                kar.leeCar();       //  leer otro
        };
        kar.toString();         //  Exhibimos el resultado
};
```