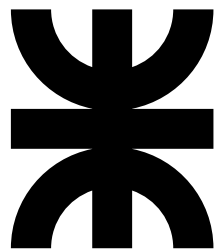
Informática II

Más sobre clases en C++

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

Objetos y funciones miembros `const`

Si se necesita un objeto de la clase **`Hora`** llamado **`mediodia`** cuyo valor no pueda ser modificado

```
const Hora mediodia(12, 0, 0);
```

Objetos y funciones miembros `const`

Si se necesita un objeto de la clase **`Hora`** llamado **`mediodia`** cuyo valor no pueda ser modificado

```
const Hora mediodia(12, 0, 0);
```

- No se pueden llamar a funciones miembros de objetos **`const`** a menos que dicha función se declare también **`const`**
- Las funciones miembros **`const`** no pueden modificar el objeto
- Un objeto **`const`** no puede modificarse por asignación así que es necesario inicializarlo

Objetos y funciones miembros `const`

Si se necesita un objeto de la clase **`Hora`** llamado **`mediodia`** cuyo valor no pueda ser modificado

```
const Hora mediodia(12, 0, 0);
```

- No se pueden llamar a funciones miembros de objetos **`const`** a menos que dicha función se declare también **`const`**
- Las funciones miembros **`const`** no pueden modificar el objeto
- Un objeto **`const`** no puede modificarse por asignación así que es necesario inicializarlo

Una función se debe especificar **`const`** en su prototipo y en su definición

```
int A::obtieneValor() const
{
  return datoMiembroPrivado;
}
```

Objetos y funciones miembros `const`

Si se necesita un objeto de la clase **`Hora`** llamado **`mediodia`** cuyo valor no pueda ser modificado

```
const Hora mediodia(12, 0, 0);
```

- No se pueden llamar a funciones miembros de objetos **`const`** a menos que dicha función se declare también **`const`**
- Las funciones miembros **`const`** no pueden modificar el objeto
- Un objeto **`const`** no puede modificarse por asignación así que es necesario inicializarlo

Una función se debe especificar **`const`** en su prototipo y en su definición

```
int A::obtieneValor() const
{
  return datoMiembroPrivado;
}
```

Es recomendable declarar como **`const`** a todas las funciones miembros que no necesitan modificar el objeto

Dato miembro `const` – inicialización

```
1  class Incremento {
2    public:
3      Incremento(int c = 0, int i = 1);
4      void sumaIncremento() { cuenta += incremento; }
5      void imprime() const;
6
7
8
9
10 };
```

Dato miembro `const` – inicialización

```
1  class Incremento {
2    public:
3      Incremento(int c = 0, int i = 1);
4      void sumaIncremento() { cuenta += incremento; }
5      void imprime() const;
6
7    private:
8      int cuenta;
9      const int incremento; // dato miembro const
10 };
```

Dato miembro `const` – inicialización

```
1  class Incremento {
2    public:
3      Incremento(int c = 0, int i = 1);
4      void sumaIncremento() { cuenta += incremento; }
5      void imprime() const;
6
7    private:
8      int cuenta;
9      const int incremento; // dato miembro const
10 };
```

```
1  // Constructor para la clase Incremento
2  Incremento::Incremento(int c, int i)
3  {
4    cuenta = c;
5    incremento = i;
6  }
```

Dato miembro `const` – inicialización

Error de compilación al intentar modificar un dato miembro constante

```
incremento.cpp: In constructor 'Incremento::Incremento(int, int)':
incremento.cpp:21:1: error: uninitialized const member in
   'const int' [-fpermissive]
 Incremento::Incremento(int c, int i)
 ^
incremento.cpp:17:15: note: 'const int Incremento::incremento'
  should be initialized
                const int incremento;   // dato miembro const
                          ^
incremento.cpp:24:14: error: assignment of read-only member
  'Incremento::incremento'
         incremento = i;
                    ^
```

Dato miembro `const` – inicialización

Error de compilación al intentar modificar un dato miembro constante

```
incremento.cpp: In constructor 'Incremento::Incremento(int, int)':
incremento.cpp:21:1: error: uninitialized const member in
   'const int' [-fpermissive]
 Incremento::Incremento(int c, int i)
 ^
incremento.cpp:17:15: note: 'const int Incremento::incremento'
  should be initialized
               const int incremento;   // dato miembro const
                         ^
incremento.cpp:24:14: error: assignment of read-only member
  'Incremento::incremento'
         incremento = i;
                    ^
```

¿Cómo inicializar un dato miembro `const` si no es posible realizar una asignación en el constructor?

Dato miembro `const` – inicialización

Se tiene que modificar el constructor de **`Incremento`** de la sig. manera:

```
Incremento::Incremento(int c, int i) : incremento(i)
{
  cuenta = c;
}
```

Dato miembro `const` – inicialización

Se tiene que modificar el constructor de `Incremento` de la sig. manera:

```
Incremento::Incremento(int c, int i) : incremento(i)
{
  cuenta = c;
}
```

- La notación `:incremento(i)` inicializa `incremento` al valor de `i`.

Dato miembro `const` – inicialización

Se tiene que modificar el constructor de `Incremento` de la sig. manera:

```
Incremento::Incremento(int c, int i) : incremento(i)
{
  cuenta = c;
}
```

- La notación `:incremento(i)` inicializa `incremento` al valor de `i`.
- Si se necesitan varios inicializadores se tiene que utilizar una lista separada por comas después de los dos puntos

Dato miembro `const` – inicialización

Se tiene que modificar el constructor de **`Incremento`** de la sig. manera:

```
Incremento::Incremento(int c, int i) : incremento(i)
{
  cuenta = c;
}
```

- La notación **`:incremento(i)`** inicializa **`incremento`** al valor de **`i`**.
- Si se necesitan varios inicializadores se tiene que utilizar una lista separada por comas después de los dos puntos
- Todos los datos miembros *pueden* inicializarse utilizando la sintaxis anterior, pero los datos miembros **`const`** *deben* inicializarse de esta manera

Dato miembro `const` – inicialización

Se tiene que modificar el constructor de `Incremento` de la sig. manera:

```
Incremento::Incremento(int c, int i) : incremento(i)
{
  cuenta = c;
}
```

- La notación `:incremento(i)` inicializa `incremento` al valor de `i`.
- Si se necesitan varios inicializadores se tiene que utilizar una lista separada por comas después de los dos puntos
- Todos los datos miembros *pueden* inicializarse utilizando la sintaxis anterior, pero los datos miembros `const` *deben* inicializarse de esta manera

La sintaxis `incremento(i)` puede verse como crear un objeto `incremento` (aún cuando sea un tipo de dato predefinido), pasándole al constructor el valor `i`

Objetos como miembros de objetos – composición de clases

Un objeto de la clase `AlarmaReloj` debe conocer cuando hacer sonar la alarma, por lo que se puede incluir un miembro dato `Hora` a la clase `AlarmaReloj`.

Objetos como miembros de objetos – composición de clases

Un objeto de la clase `AlarmaReloj` debe conocer cuando hacer sonar la alarma, por lo que se puede incluir un miembro dato `Hora` a la clase `AlarmaReloj`.

Composición de clases

Una clase puede tener como miembro objetos de otra clase

Objetos como miembros de objetos – composición de clases

Un objeto de la clase `AlarmaReloj` debe conocer cuando hacer sonar la alarma, por lo que se puede incluir un miembro dato `Hora` a la clase `AlarmaReloj`.

Composición de clases

Una clase puede tener como miembro objetos de otra clase

- Siempre que se crea un objeto se invoca a un constructor. ¿Cómo pasarle argumentos a los constructores de los objetos miembros?
- Los objetos miembros se construyen en el orden que se declara y antes de que se construyan los objetos que los contienen

Objetos como miembros de objetos – composición de clases

```
1  class Fecha {
2    public:
3      Fecha(int = 1, int = 1, int = 1900); // constructor predeterminado
4      void imprime() const; // imprime la fecha en formato mes/día/año
5      ~Fecha(); // proporcionado para confirmar el orden de destrucción
6
7    private:
8      int mes, dia, anio;
9
10     // Función de utilidad para verificar el día adecuado para el mes y el año
11     int verificaDia(int);
12 };
```

Objetos como miembros de objetos – composición de clases

```
1  class Fecha {
2    public:
3      Fecha(int = 1, int = 1, int = 1900); // constructor predeterminado
4      void imprime() const; // imprime la fecha en formato mes/día/año
5      ~Fecha(); // proporcionado para confirmar el orden de destrucción
6
7    private:
8      int mes, dia, anio;
9
10     // Función de utilidad para verificar el día adecuado para el mes y el año
11     int verificaDia(int);
12 };
```

```
1  class Empleado {
2    public:
3      Empleado(char *, char*, int, int, int, int, int, int);
4      void imprime() const;
5
6    private:
7      char nombre[25];
8      char apellido[25];
9      const Fecha fechaNacimiento;
10     const Fecha fechaContratacion;
11 };
```

Objetos como miembros de objetos – composición de clases

Los datos miembros `fechaNacimiento` y `fechaContratacion` son objetos `const` de la clase `Fecha`, la cual contiene miembros privados `dia`, `mes` y `anio`.

Objetos como miembros de objetos – composición de clases

Los datos miembros `fechaNacimiento` y `fechaContratacion` son objetos `const` de la clase `Fecha`, la cual contiene miembros privados `dia`, `mes` y `anio`.

¿Cómo pasarles desde el constructor de `Empleado` los valores al constructor de `Fecha`?

Objetos como miembros de objetos – composición de clases

Los datos miembros **fechaNacimiento** y **fechaContratacion** son objetos **const** de la clase **Fecha**, la cual contiene miembros privados **dia**, **mes** y **anio**.

¿Cómo pasarles desde el constructor de **Empleado** los valores al constructor de **Fecha**?

```
Empleado::Empleado( char *nomb, char *apell,
                    int mesnacim, int dianacim, int anionacim,
                    int mescontrat, int diacontrat, aniocontrat )
```

Objetos como miembros de objetos – composición de clases

Los datos miembros **fechaNacimiento** y **fechaContratacion** son objetos **const** de la clase **Fecha**, la cual contiene miembros privados **dia**, **mes** y **anio**.

¿Cómo pasarles desde el constructor de **Empleado** los valores al constructor de **Fecha**?

```
Empleado::Empleado( char *nomb, char *apell,
                    int mesnacim, int dianacim, int anionacim,
                    int mescontrat, int diacontrat, aniocontrat )
   : fechaNacimiento( mesnacim, dianacim, anionacim ),
```

Objetos como miembros de objetos – composición de clases

Los datos miembros **fechaNacimiento** y **fechaContratacion** son objetos **const** de la clase **Fecha**, la cual contiene miembros privados **dia**, **mes** y **anio**.

¿Cómo pasarles desde el constructor de **Empleado** los valores al constructor de **Fecha**?

```
Empleado::Empleado( char *nomb, char *apell,
                    int mesnacim, int dianacim, int anionacim,
                    int mescontrat, int diacontrat, aniocontrat )
   : fechaNacimiento( mesnacim, dianacim, anionacim ),
     fechaContratacion( mescontrat, diacontrat, aniocontrat )
```

Objetos como miembros de objetos – composición de clases

Los datos miembros **fechaNacimiento** y **fechaContratacion** son objetos **const** de la clase **Fecha**, la cual contiene miembros privados **dia**, **mes** y **anio**.

¿Cómo pasarles desde el constructor de **Empleado** los valores al constructor de **Fecha**?

```
Empleado::Empleado( char *nomb, char *apell,
                    int mesnacim, int dianacim, int anionacim,
                    int mescontrat, int diacontrat, aniocontrat )
   : fechaNacimiento( mesnacim, dianacim, anionacim ),
     fechaContratacion( mescontrat, diacontrat, aniocontrat )
```

Ver código fuente de **fig17_04.cpp**.

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

- La amistad se gana, no se toma. Para que la clase B sea amiga de la clase A, la clase A debe declarar de manera explícita que la clase B es su amiga

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

- La amistad se gana, no se toma. Para que la clase B sea amiga de la clase A, la clase A debe declarar de manera explícita que la clase B es su amiga
- La amistad no es simétrica ni transitiva

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

- La amistad se gana, no se toma. Para que la clase B sea amiga de la clase A, la clase A debe declarar de manera explícita que la clase B es su amiga
- La amistad no es simétrica ni transitiva
- Aún cuando los prototipos de las funciones amigas aparecen en la definición de la clases, las mismas no son funciones miembro

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

- La amistad se gana, no se toma. Para que la clase B sea amiga de la clase A, la clase A debe declarar de manera explícita que la clase B es su amiga
- La amistad no es simétrica ni transitiva
- Aún cuando los prototipos de las funciones amigas aparecen en la definición de la clases, las mismas no son funciones miembro
- Algunos programadores consideran que la *"amistad"* rompe el ocultamiento de información y debilita el valor del método de diseño orientado a objetos

Funciones y clases amigas – `friend`

- Las funciones **`friend`** de una clase se definen fuera del alcance de la clase (no son miembros) pero tiene acceso a los miembros privados (y protegidos) de la clase
- Se puede declarar una función o una clase completa como **`friend`** de otra clase

Se declara una función amiga de una clase antecediendo la palabra reservada **`friend`** al prototipo de la función en la definición de la clase.

Para declarar a la **`ClaseDos`** como amiga de la **`ClaseUno`**, en la definición de **`ClaseUno`** debe agregarse

```
friend class ClaseDos;
```

- La amistad se gana, no se toma. Para que la clase B sea amiga de la clase A, la clase A debe declarar de manera explícita que la clase B es su amiga
- La amistad no es simétrica ni transitiva
- Aún cuando los prototipos de las funciones amigas aparecen en la definición de la clases, las mismas no son funciones miembro
- Algunos programadores consideran que la *"amistad"* rompe el ocultamiento de información y debilita el valor del método de diseño orientado a objetos

Ver código fuente ejemplo **`fig17_05.cpp`** y **`fig17_06.cpp`** (D&D 4° ed.)

Funciones y clases amigas – `friend`

```
1  #include <iostream>
2  using namespace std;
3
4  // Clase modificada Cuenta
5  class Cuenta {
6   friend void estableceX(Cuenta & , int ); // Declaración de la amiga
7
8   public:
9     Cuenta() { x = 0; } // Constructor
10    void imprime() const { cout << x << endl; } // Salida
11
12  private:
13    int x; // dato miembro
14 };
15
16 void estableceX(Cuenta &c, int val) {
17   c.x = val; // legal: estableceX es una amiga de Cuenta
18 }
19
20 void noPuedeEstablecerX(Cuenta &c, int val) {
21   c.x = val; // ERROR: 'Cuenta::x' no es accesible
22 }
23
24 // Sigue función main()
```

Funciones y clases amigas – `friend`

```
1  #include <iostream>
2  using namespace std;
3
4  // Clase modificada Cuenta
5  class Cuenta {
6   friend void estableceX(Cuenta & , int ); // Declaración de la amiga
7
8   public:
9     Cuenta() { x = 0; } // Constructor
10    void imprime() const { cout << x << endl; } // Salida
11
12  private:
13    int x; // dato miembro
14 };
15
16 void estableceX(Cuenta &c, int val) {
17   c.x = val; // legal: estableceX es una amiga de Cuenta
18 }
19
20 void noPuedeEstablecerX(Cuenta &c, int val) {
21   c.x = val; // ERROR: 'Cuenta::x' no es accesible
22 }
23
24 // Sigue función main()
```

En general resulta apropiado definir funciones **`set`** como funciones miembro de clase.

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado **`this`**

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto
- También puede utilizarse de manera explícita

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto
- También puede utilizarse de manera explícita

El tipo de puntero `this` depende del tipo del objeto y de si la función miembro donde se utiliza se declara `const`

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto
- También puede utilizarse de manera explícita

El tipo de puntero `this` depende del tipo del objeto y de si la función miembro donde se utiliza se declara `const`

1. En una función miembro no constante de la clase `Empleado` el puntero `this` es del tipo `Empleado * const`
   (o sea, un puntero constante a un objeto `Empleado`)

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado `this`

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador `sizeof` sobre el objeto
- El puntero `this` se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto
- También puede utilizarse de manera explícita

El tipo de puntero `this` depende del tipo del objeto y de si la función miembro donde se utiliza se declara `const`

1. En una función miembro no constante de la clase `Empleado` el puntero `this` es del tipo `Empleado * const`
   (o sea, un puntero constante a un objeto `Empleado`)
2. E una función miembro constante de la clase `Empleado` el puntero `this` es del tipo `const Empleado * const`
   (o sea, un puntero constante a un objeto `Empleado` constante)

El puntero `this`

Todos los objetos tiene un puntero apuntando a sí mismo llamado **`this`**

- Este puntero no es parte del objeto, esto se aprecia si se utiliza el operador **`sizeof`** sobre el objeto
- El puntero **`this`** se le pasa al objeto como primer argumento implícito de cada función miembro no estática (lo hace el compilador)
- Se utiliza de manera implícita para acceder a los miembros de un objeto
- También puede utilizarse de manera explícita

El tipo de puntero **`this`** depende del tipo del objeto y de si la función miembro donde se utiliza se declara **`const`**

1. En una función miembro no constante de la clase **`Empleado`** el puntero **`this`** es del tipo **`Empleado * const`**
   (o sea, un puntero constante a un objeto **`Empleado`**)
2. E una función miembro constante de la clase **`Empleado`** el puntero **`this`** es del tipo **`const Empleado * const`**
   (o sea, un puntero constante a un objeto **`Empleado`** constante)

Ver código fuente ejemplo **`fig17_07.cpp`**

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

(el operador punto se asocia de izquierda a derecha)

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

(el operador punto se asocia de izquierda a derecha)

¿Qué devuelve **`h.estableceHora(20, 20, 20)`**?

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

  (el operador punto se asocia de izquierda a derecha)

  ¿Qué devuelve **`h.estableceHora(20, 20, 20)`**?

- Otro ejemplo

```
h.estableceHora(18).estableceMinuto(30).estableceSegundo(22);
```

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

  (el operador punto se asocia de izquierda a derecha)

  ¿Qué devuelve **`h.estableceHora(20, 20, 20)`**?

- Otro ejemplo

```
h.estableceHora(18).estableceMinuto(30).estableceSegundo(22);
```

Funciones miembros que retornan el puntero **`this`**

```
1 Hora &Hora::estableceHora(int h, int m, int s)
2 {
3   estableceHora(h);
4   estableceMinuto(m);
5   estableceSegundo(s);
6   return *this; // permite la cascada
7 }
```

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

  (el operador punto se asocia de izquierda a derecha)

  ¿Qué devuelve **`h.estableceHora(20, 20, 20)`**?

- Otro ejemplo

```
h.estableceHora(18).estableceMinuto(30).estableceSegundo(22);
```

Funciones miembros que retornan el puntero **`this`**

```
1 Hora &Hora::estableceHora(int h, int m, int s)
2 {
3   estableceHora(h);
4   estableceMinuto(m);
5   estableceSegundo(s);
6   return *this; // permite la cascada
7 }
```

(**`this`** es un puntero y **`*this`** es un objeto)

El puntero `this`

- El puntero **`this`** se puede utilizar para poder realizar llamadas a funciones miembro en cascada. Supongamos un objeto **`h`** de la clase **`Hora`**

```
h.estableceHora(20, 20, 20).imprimeEstandar();
```

  (el operador punto se asocia de izquierda a derecha)

  ¿Qué devuelve **`h.estableceHora(20, 20, 20)`**?

- Otro ejemplo

```
h.estableceHora(18).estableceMinuto(30).estableceSegundo(22);
```

Funciones miembros que retornan el puntero **`this`**

```
1 Hora &Hora::estableceHora(int h, int m, int s)
2 {
3   estableceHora(h);
4   estableceMinuto(m);
5   estableceSegundo(s);
6   return *this; // permite la cascada
7 }
```

(**`this`** es un puntero y **`*this`** es un objeto)

Ver código fuente ejemplo **`fig17_08.cpp`**

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

- En ANSI C se puede crear un objeto **`nombreTipo`** de forma dinámica con

```
ptrNombreTipo = malloc(sizeof(nombreTipo));
```

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

- En ANSI C se puede crear un objeto **`nombreTipo`** de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función **`malloc`** y se hace uso explícito del operador **`sizeof`**)

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

- En ANSI C se puede crear un objeto **`nombreTipo`** de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función **`malloc`** y se hace uso explícito del operador **`sizeof`**)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo;
  ```

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

- En ANSI C se puede crear un objeto **`nombreTipo`** de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función **`malloc`** y se hace uso explícito del operador **`sizeof`**)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo;
  ```

  (crea un objeto del tamaño apropiado, llama al constructor y devuelve puntero del tipo adecuado).

Asignación dinámica en C++ – operadores `new` y `delete`

C++ dispone de los operadores **`new`** y **`delete`** para realizar asignación dinámica de memoria para cualquier tipo predefinido o definido por el usuario

Es una mejora a las funciones **`malloc`** y **`free`** de la biblioteca estándar de C.

Si se tiene

```
nombreTipo *ptrNombreTipo; // int *ptrInt, o Hora *ptrHora
```

- En ANSI C se puede crear un objeto **`nombreTipo`** de forma dinámica con

  ```
  ptrNombreTipo = malloc(sizeof(nombreTipo));
  ```

  (se llama a la función **`malloc`** y se hace uso explícito del operador **`sizeof`**)

- En C++ resulta

  ```
  ptrNombreTipo = new nombreTipo;
  ```

  (crea un objeto del tamaño apropiado, llama al constructor y devuelve puntero del tipo adecuado). Para destruir el objeto y liberar memoria se hace

  ```
  delete ptrNombreTipo;
  ```

Asignación dinámica en C++ – operadores `new` y `delete`

En C++ se puede utilizar un *inicializador* para un objeto creado con **`new`** tal como

```
double *ptrPi = new double(3.14159);
```

Asignación dinámica en C++ – operadores `new` y `delete`

En C++ se puede utilizar un *inicializador* para un objeto creado con **`new`** tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace

```
int *ptrArreglo = new int [10];
```

Asignación dinámica en C++ – operadores `new` y `delete`

En C++ se puede utilizar un *inicializador* para un objeto creado con **`new`** tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace

```
int *ptrArreglo = new int [10];
```

el cual se borra con

```
delete [] ptrArreglo;
```

Asignación dinámica en C++ – operadores `new` y `delete`

En C++ se puede utilizar un *inicializador* para un objeto creado con **`new`** tal como

```
double *ptrPi = new double(3.14159);
```

Para crear un arreglo de 10 elementos se hace

```
int *ptrArreglo = new int [10];
```

el cual se borra con

```
delete [] ptrArreglo;
```

El uso de **`new`** y **`delete`** ofrece varios beneficios comparados con **`malloc()`** y **`free()`**. Por ejemplo, **`new`** invoca al constructor y **`delete`** invoca al destructor de la clase.

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`
- Si están presente menos de cinco `Marcianos`, cada `Marciano` se acobarda

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`
- Si están presente menos de cinco `Marcianos`, cada `Marciano` se acobarda
- Cada `Marcianos` necesita saber cuantos son (`cuentaMarcianos`)

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`
- Si están presente menos de cinco `Marcianos`, cada `Marciano` se acobarda
- Cada `Marcianos` necesita saber cuantos son (`cuentaMarcianos`)
- Se podría incluir `cuentaMarcianos` en cada instancia de la clase `Marciano` como un dato miembro

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`
- Si están presente menos de cinco `Marcianos`, cada `Marciano` se acobarda
- Cada `Marcianos` necesita saber cuantos son (`cuentaMarcianos`)
- Se podría incluir `cuentaMarcianos` en cada instancia de la clase `Marciano` como un dato miembro
- Entonces, cada `Marciano` tendrá una copia por separado de los miembros datos, y cada vez que se crea un `Marciano` se debe actualizar el dato miembro `cuentaMarcianos` en cada objeto `Marciano`

Miembros de clases `static`

- Cada objeto de una determinada clase tiene su propia copia de los datos miembros
- Un dato miembro `static` contiene información *"intrínseca de la clase"*, o sea, propia de la clase y no de un objeto particular

Ejemplo: juego de video de `Marcianos`

- Cada `Marciano` tiende a ser valiente y está dispuesto a atacar a otras criaturas del espacio cuando se da cuenta de que están presente al menos cinco `Marcianos`
- Si están presente menos de cinco `Marcianos`, cada `Marciano` se acobarda
- Cada `Marcianos` necesita saber cuantos son (`cuentaMarcianos`)
- Se podría incluir `cuentaMarcianos` en cada instancia de la clase `Marciano` como un dato miembro
- Entonces, cada `Marciano` tendrá una copia por separado de los miembros datos, y cada vez que se crea un `Marciano` se debe actualizar el dato miembro `cuentaMarcianos` en cada objeto `Marciano`
- Esto resulta en un desperdicio de espacio con las copias redundante, y un desperdicio de tiempo para actualizar cada copia por separada

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Los datos miembros estáticos de una clase existen incluso si no existen objetos de esa clase. Se tiene acceso al dato miembro `static`

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Los datos miembros estáticos de una clase existen incluso si no existen objetos de esa clase. Se tiene acceso al dato miembro `static`

- Si es un miembro público se utiliza como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::cuentaMarcianos`)

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Los datos miembros estáticos de una clase existen incluso si no existen objetos de esa clase. Se tiene acceso al dato miembro `static`

- Si es un miembro público se utiliza como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::cuentaMarcianos`)
- Si es un miembro privado debe existir una función miembro pública y estática la cual debe invocarse colocando como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::obtieneCuenta()`)

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Los datos miembros estáticos de una clase existen incluso si no existen objetos de esa clase. Se tiene acceso al dato miembro `static`

- Si es un miembro público se utiliza como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::cuentaMarcianos`)
- Si es un miembro privado debe existir una función miembro pública y estática la cual debe invocarse colocando como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::obtieneCuenta()`)

Ver código fuente ejemplo `fig17_09.cpp` (D&D 4° ed.)

Miembros de clases `static`

- En vez de esto, se declara a la variable `cuentaMarcianos` como `static`
- Por lo tanto, `cuentaMarcianos` es un dato intrínseco de la clase
- Cada objeto `Marciano` puede ver a `cuentaMarcianos` como si fuera un dato miembro de `Marciano` pero C++ solo mantiene una copia estática del dato miembro

Los datos miembros estáticos de una clase existen incluso si no existen objetos de esa clase. Se tiene acceso al dato miembro `static`

- Si es un miembro público se utiliza como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::cuentaMarcianos`)
- Si es un miembro privado debe existir una función miembro pública y estática la cual debe invocarse colocando como prefijo el nombre de la clase y el operador de resolución de alcance (`Marciano::obtieneCuenta()`)

Ver código fuente ejemplo `fig17_09.cpp` (D&D 4° ed.)

El dato miembro estático `cuenta` se inicializa a cero (con alcance de archivo) haciendo

```
int Empleado::cuenta = 0;
```