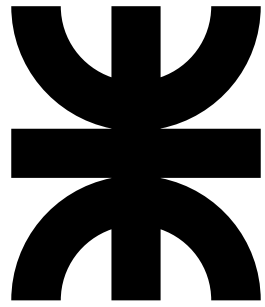
Informática II
Más sobre clases en C++

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2019 –

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 NombreClase::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 NombreClase::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 de inicialización, en este caso `i`

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

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

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

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      int verificaDia(int); // Verifica día para mes y año
10 };
```

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      int verificaDia(int); // Verifica día para mes y año
10 };
```

```
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 };
```

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`.

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`?

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 )
```

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 ),
```

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 )
```

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 tienen 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 tienen 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 tienen 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 tienen 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`

- 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`

- 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`

- 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
  (si la clase A es amiga de la clase B, y la clase B es amiga de la clase C, no se puede inferir que la clase B sea amiga de la clase A (la amistad no es simétrica), que la clase C es amiga de la clase B o que la clase A es amiga de la clase C (la amistad no es transitiva))

Funciones y clases amigas – `friend`

- 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
  (si la clase A es amiga de la clase B, y la clase B es amiga de la clase C, no se puede inferir que la clase B sea amiga de la clase A (la amistad no es simétrica), que la clase C es amiga de la clase B o que la clase A es amiga de la clase C (la amistad no es 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`

- 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
  (si la clase A es amiga de la clase B, y la clase B es amiga de la clase C, no se puede inferir que la clase B sea amiga de la clase A (la amistad no es simétrica), que la clase C es amiga de la clase B o que la clase A es amiga de la clase C (la amistad no es 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`

- 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
  (si la clase A es amiga de la clase B, y la clase B es amiga de la clase C, no se puede inferir que la clase B sea amiga de la clase A (la amistad no es simétrica), que la clase C es amiga de la clase B o que la clase A es amiga de la clase C (la amistad no es 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  class Cuenta { // Clase modificada Cuenta
5   friend void estableceX(Cuenta & , int ); // Declaración de la amiga
6
7   public:
8     Cuenta() { x = 0; } // Constructor
9     void imprime() const { cout << x << endl; } // Salida
10
11  private:
12    int x; // dato miembro
13 };
14
15 void estableceX(Cuenta &c, int val) {
16   c.x = val; // legal: estableceX es una amiga de Cuenta
17 }
18
19 void noPuedeEstablecerX(Cuenta &c, int val) {
20   c.x = val; // ERROR: 'Cuenta::x' no es accesible
21 }
```

Funciones y clases amigas – `friend`

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

En general resulta apropiado definir funciones `set` como funciones miembro de la 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`

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`

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`

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. En 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`

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. En 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++ (`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++ (`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++ (`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++ (`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++ (`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++ (`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; // ptrInt = new int, o ptrHora = new Hora
  ```

Asignación dinámica en C++ (`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; // ptrInt = new int, o ptrHora = new Hora
  ```

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

Asignación dinámica en C++ (`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; // ptrInt = new int, o ptrHora = new Hora
  ```

  (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; // delete ptrInt, o delete ptrHora
  ```

Asignación dinámica en C++ (`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++ (`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++ (`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++ (`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.

Actividad práctica

A partir de la declaración de la clase `Cadena` en el archivo `cadena.h` y del programa `prueba_cadena.cpp`, codificar el archivo `cadena.cpp` donde se implementen las funciones miembros de la clase. Utilizar asignación dinámica de memoria (con `new` y `delete`) para reservar espacio para guardar la cadena.

`cadena.h`

```
1  #ifndef CADENA_H
2  #define CADENA_H
3
4  class Cadena {
5    public:
6      Cadena(int = 20); // Constructor
7      ~Cadena(); // Destructur
8
9      void establecer(const char *);
10     void imprimir() const;
11
12   private:
13     const int longitud;
14     char *cad;
15 };
16 #endif
```

`prueba_cadena.cpp`

```
1  #include <iostream>
2  #include <cstring>
3  #include "cadena.h"
4
5  int main() {
6    Cadena cad1;
7    cad1.establecer("Hola");
8    cad1.imprimir();
9
10   Cadena cad2(27);
11   cad2.establecer(
12     "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
13   cad2.imprimir();
14
15   return 0;
16 }
```