4. Clases II. Uso avanzado

Julio Vega

julio.vega@urjc.es

Contenidos

- Todos sabemos el mecanismo de la llamada a una función *normal*.
  - 1. Al llamarla, la CPU almacena la dirección de la siguiente instrucción.
  - 2. La CPU ejecuta la función, almacena el valor de retorno y *vuelve*.
- Puede suponer un sobrecoste comput. si $t_{ejec\_funcion} < t_{cambio\_contexto}$.
  - En función compleja compensa, ya que $t_{ejec\_funcion} \ggg t_{cambio\_contexto}$.
  - En función simple, puede ocurrir que $t_{ejec\_funcion} < t_{cambio\_contexto}$.
- C++ ofrece el tipo de función `inline` para paliar este problema.
  - Es una solicitud, no una orden; el compilador puede no ponerla en línea.
  - Una función declarada como `inline` no conlleva cambio de contexto.
  - El compilador *copia* el código de una función `inline`.
    - Y lo *pega* en cada sitio desde donde se llame, en t.^{o} de compilación.
  - Pros: $t_{cambio\_contexto} \approx 0$. Contras: tamaño de ejecutable $\uparrow\uparrow$.
  - ¿Cuándo usar explícitamente `inline`? Con funciones muy simples.

- Un método de la clase se puede definir fuera de la definición de clase.
  - Y *enlazarse* a la clase mediante el operador de resolución binario.
- Lo normal es definir una función miembro en el cuerpo de la clase.
  - Porque el compilador intenta poner en línea las llamadas a estas.
  - Si están fuera se pueden poner en línea con la palabra `inline`.
- Otro beneficio de la POO es la simplificación en las llamadas.
  - El número de parámetros de los métodos de clase es reducido.
  - Los atributos y métodos están encapsulados en el objeto.
  - Los métodos tienen el derecho a acceder directamente a los atributos.

Uso redundante e inapropiado de inline dentro de una clase

```
class A {
public:
   inline int methodA () { // redundant use of inline
      // [...] this function is automatically inline
   }
}; // end class A
```

Uso redundante aunque apropiado de inline en la definición del método

```
class B {
public:
   int methodB (); // declare the function
};

inline int B::methodB () { // inline in the definition
  // redundante, pero "sugerencia fuerte" al compilador?
}
```

```
#include <iostream>
using namespace std; // to be analyzed
inline void smallFunction () { // inline function out of class
  cout << "Me gusta usar inline" << endl;
}

class C {
private:
  int x;
public:
  C (int x0) : x (x0) {}
  inline int getX () const { // to be analyzed
    return x; }
}; // end of class C

int main() {
  smallFunction();
  C c(7); cout << c.getX () << endl; return 0;
}
```

Uso generalizado y recomendado de `inline`

- La función *inline* no pertenece a la clase.
  - Por tanto, `inline` en este caso sí que tiene *efecto*.
- La función *inline* es muy simple; tiene poco código.
  - Se va a copiar su contenido en todos lo sitios desde los que se llame.
  - Pero el tamaño resultante del ejecutable no supondrá un problema.
- Las funciones *inline* deben ir directamente en el encabezado (.h).
  - No hay peligro de multi-definición.
    - La función no *inline* es implementada en un encabezado, y este es...
    - incluido en varios lugares $\Longrightarrow$ error compilación - definición múltiple.
  - El símbolo de la función *inline* no aparece explícit. en el enlazado.
    - Pues todas las llamadas a esta función han sido reemplazadas.
  - Al compilar un .h de ISV, la def. de la función pueda ir en su sitio.
    - ISV = *Independent Software Vendor* (o software privativo).

- Un *namespace* (ns) o espacio de nombre es un bloque o ámbito.
- Se usa para limitar el alcance de variables y funciones.
- Surge en C++ ante la limitación de no poder repetir identificadores.
  - En un mismo ámbito no se pueden repetir variables, funciones o clases.
- Problema: conflicto al importar librería con un identif. que ya usamos.
  - E.g. al importar OpenGL, que tiene función `print`, y nosotros también.
  - Solución: poner prefijo al identificador (e.g. `print` $\Longrightarrow$ `gl_print`).
- Para poder usar las variables o funciones que están en un `namespace`.
  - Hay que poner su *prefijo* con el operador de resolución de alcance (`::`).
  - Para no tener que usar el *prefijo*, se resuelve con `using`.
    - E.g. `using namespace std;` para usar la librería estándar.
    - Así, podemos usar `cout` en lugar de `std::cout`.
- Pros: $\uparrow$ legibilidad $\downarrow$ tamaño código. Contras: conflictos con otros *ns*.
  - E.g. uso dos ns *opencv* y *opengl* y uso una función común a ambos...
    - ¿A qué función se llamaría? ¡El resultado sería impredecible!

```
namespace saludo {
  void print () { std::cout << "hola";
}

void print () {
  std::cout << "adios";
}

int main() {
  print ();
  saludo::print ();
  return 0;
}
```

- *Algo* `const` indica al compilador que ha de prevenir su modificación.
  - E.g. si intentamos modificar una variable *const*, el compilador nos avisa.
- Una función miembro declarada como *const* es una *read-only func.*
  - E.g. en las *getters*, que solo consultan un valor, pero no lo modifican.
- Para usar una variable *const* desde otro módulo se le pone `extern`.
  - Tanto en la declaración de esta como en el módulo donde se usa.
    - E.g. (.h): `extern const int i = 2;` y (.cpp): `extern const int i;`
  - Esto no ocurre en C: `const int i = 2;` y `extern const int i;`

```
int main() { // Example A: it shows compile errors
  const int i = 5;
  i = 10; // C3892: assignment of read-only variable 'i'
  i++; // C2105: increment of read-only variable 'i'
}

int main() { // Example B: const value to specify size of array
  const int maxArray = 128;
  char storeChar [maxArray]; // allowed in C++, but not in C
}

int main() { // const objects: only const functions can be called
    const C c{}; // calls default constructor
    //c.x = 5; // (with x public) compiler error: violates const
    //c.setX(5); // compiler error: violates const
    c.getX(); // it's the only function that works
    return 0;
}
```

- Definir como `const` un método que modifica atributos del objeto.
- Definir como `const` un método que llama a un método no `const`.
- Invocar a un método no `const` en un objeto const.
- Declarar un constructor o destructor `const`.

Clase Increment con count (no const) e increment (const)

```
class Increment { // Definition of class Increment
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement() {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment

private:
   int count;
   const int increment; // const data member
}; // end class Increment
```

Inicializador de miembros utilizado para inicializar un atributo const

```
Increment::Increment (int c, int i) // constructor
   : count( c ), // initializer for non-const member
     increment( i ) { // required initializer for const member
   // empty body
} // end constructor Increment

void Increment::print() const { // print count & increment values
   cout << "count = "<<count<<", increment = "<<increment<<endl;
} // end function print
```

- Todos los atributos se pueden inicializar mediante inicializador.
- Los miembros `const` y miembros que sean referencias **deben**.
  - Para así proporcionar al constructor el valor inicial de los atributos.
- Ya veremos que los objetos miembro también **deben**.
  - Y con herencia, los fragmentos heredados por las clases hija también.
- Un objeto `const` no se puede modificar mediante la asignación.
  - Por lo que debe inicializarse.
- *Tip*: declara `const` todas los métodos que no modifiquen el objeto.
  - Aunque no haya intención de:
    - Crear objetos `const` de esa clase.
    - Acceder a objetos de esa clase a través de referencias `const`.
  - Así, si por error el método modifica el objeto $\Longrightarrow$ error de compilación.

- Pensemos en un objeto `Empleado` (de *Julio Veganos e Hijos*).
- Este empleado tendrá un objeto fecha (*date*) de nacimiento.
- ¿Por qué no incluir un objeto `Date` como miembro de `Employee`?
  - De hecho, un empleado tendría dos fechas: f. nac. y f. contrato.
- Esta capacidad se conoce como composición o relación *tiene un*.
  - Es decir, una clase puede tener objetos de otras clases como atributos.
- Una forma común de reutilización de software es la composición.

```
#ifndef DATE_H
#define DATE_H

class Date {
public:
   static const unsigned int monthsPerYear = 12; // in a year
   explicit Date( int = 1, int = 1, int = 1900 ); // def. const.
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   unsigned int month; // 1-12 (January-December)
   unsigned int day; // 1-31 based on month
   unsigned int year; // any year

   // utility funct. to check if day is proper for month and year
   unsigned int checkDay( int ) const;
}; // end class Date

#endif
```

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition

class Employee {
public:
   Employee (const std::string &, const std::string &,
      const Date &, const Date &);
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   std::string firstName; // composition: member object
   std::string lastName; // composition: member object
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
}; // end class Employee

#endif
```

- La clase `Date` y la clase `Employee` incluyen su destructor.
- Los objetos se construyen de dentro a afuera y se destruyen inversa/.
  - Si ponemos trazas a los const./destructores se puede apreciar.
- Un objeto miembro no necesita inicializarse de manera explícita.
  - A través de un inicializador de miembros.
  - Si no existe, se llama implícita/ a su constructor predeterminado.
- Si objeto `A` es atributo `public` de una clase `B`, no se viola...
  - ...encapsulamiento/ocultamiento de los atributos private del objeto `A`.
  - Pero sí se viola ocultamiento de la clase `B` que lo contiene.
  - Corolario: los atributos objeto deben ser `private` como los demás.

- Una función `friend` de una clase es la que se define fuera de esta.
  - Pero que tiene derecho a acceder a todos sus miembros (pub. y priv.).
- Se pueden declarar funciones independientes y clases `friend`.
  - Func. útiles para sobrecargar operadores, y tb. para clases iteradoras.
- Para declarar una función como `friend` de una clase.
  - Se pone `friend` al prototipo de la función en definición de clase.
- Para que todos los métodos de la clase `B` sean `friend` de clase `A`.
  - Declarar `friend class B;` en la definición de la clase `A`.
- Todo miembro puede ser `friend`: `private`, `protected` y `public`.
  - Todos los `friend` van al ppio. de clase sin especificar visibilidad.
- La relación `friend` no es simétrica ni transitiva.
  - Si A `friend` de B & B `friend` de C:
    - $\not\Rightarrow$ B `friend` de A (la amistad no es simétrica).
    - $\not\Rightarrow$ C `friend` de B (la amistad no es simétrica).
    - $\not\Rightarrow$ A `friend` de C (la amistad no es transitiva).
- `friend` puede corromper ocultamiento y debilitar POO. Ejemplos...

```
class A {
   friend void setX (A &, int); // friend declaration

public:
   A () : x (0) {};
   void print() const { cout << x << endl; }

private:
   int x;
};

void setX (A &a, int value) {
   a.x = value; // it is possible due to setX is A's friend
}
```

- Lo ideal sería definir la función `setX` como miembro de `A`.
- Archivo de encabezado (`A.h`) y archivo de implementación (`A.cpp`).
- Y en otro archivo separado, el programa de prueba (`main.cpp`).

- Hemos visto que los métodos pueden manipular los atributos de clase.
  - ¿Cómo saben los métodos qué atributos del objeto pueden manipular?
- ¿Recuerdas lo de clase = plano para construir un objeto de esa clase?
  - ¿Al ejecutar código de clase para objeto concreto, cómo sabe dónde?
- Cada objeto tiene acceso a su propia direc. mediante apuntador `this`.
  - Este apuntador se pasa implícitamente como un argumento más.

```
class A {
public:
  void print() const { // includes argument "this" (hidden)
    cout << x << endl; // implicit use
    cout << this->x << endl; // explicit use
    cout << (*this).x << endl; // explicit unreferenced use
// () is mandatory, otherwise: *this.x = *(this.x) => comp. error
  }

private:
  int x;
};
```

- C++ permite controlar la (des)asignación de memoria del programa.
- Uso: e.g. en la práctica, ¿cómo almacenar el nombre de un empleado?
  - Uso fijo: aprovecha mal la memoria (¿y si nombre más largo/corto?).
  - Uso dinámico: usar exactamente lo necesario según la necesidad.
- Toda colección dinámica creada se almacena en el *heap* o cúmulo.
  - Es una de las regiones de la memoria (RAM) asignada al programa.
  - Se accede a la colección mediante un apuntador al primer elemento.

```
Date *datePtr;
datePtr = new Date; // "new" calls the default constructor
delete datePtr; // "delete" calls the destructor & frees memory
// -------------------------
int *grades = new int [10]; // dynamic array with a static size
delete [] *grades; // "delete []" calls every element's destr.
                   // "delete" for a collection is a logic error!
```

- En general, un objeto tiene copia de todos los miembros de la clase.
- En ciertos casos, solo se necesita de algunos miembros; no de todos.
- Un miembro `static` es compartido por todas las instancias de clase.
  - Es decir, no es una propiedad de un objeto específico de la clase.
    - Es como una variable global pero con acceso a miembros de clase.
  - E.g. atributo `count` para llevar cuenta de instancias de la clase.
    - Si atributo es no `static`, cada objeto llevaría su propia cuenta.
    - Si atributo es `static`, este llevaría la cuenta *global* de la clase.
- Corolario: se usa `static` cuando solo se necesite una copia.

```
class A { // --------- A.h -----------------
public:
  A ();
  static int getCounter (); // return no. objects
private:
  static int counter; // no. objects
};
//--------------- A.cpp --------------------
int A::counter = 0; // can't include static keyword
int A::getCounter () { return counter; }
A::A () { counter++; }
//-------------- main.cpp -----------------
A *a = new A (); // constructor is called
cout << A::getCounter() << endl; // shows 1 (obj.)
delete a;
a = 0;
```

- En llamadas a func. miembro `static` se antepone nombre de clase.
- Usar `this` en func. miembro `static` es un error de compilación.
  - Una función miembro `static` no tiene apuntador `this`.
  - Un miembro `static` es independiente de cualquier obj. de la clase.
- Declarar `const` una func. miembro `static` es error de compilación.
  - `const` indica que la func. no puede modificar el contenido del objeto.
  - Pero func. miembro `static` opera independiente/ de los obj. de clase.
- Tras eliminar la memoria asignada de forma dinámica (`delete a;`).
  - Este espacio podría seguir teniendo información de lo que sea.
    - Y el programa podría acceder, obteniendo errores lógicos.
  - Una buena costumbre es establecer el puntero en 0 (`a = 0;`).
    - Así se desconecta el puntero del espacio que le había sido asignado.
    - Apuntando a un espacio al cual el programa no tiene acceso.

4. Clases II. Uso avanzado

Julio Vega

julio.vega@urjc.es