Programación Orientada a Objetos

Matilde Fernández Azuela

Madrid, septiembre de 2008

Clases y objetos

Objeto (Booch,1994)

- Algo visible o tangible
- Un concepto intelectualmente descrito
- Una entidad hacia la que se dirige el procesamiento o la acción

Partiendo del enunciado de un problema todos los nombres del enunciado son objetos a tener en cuenta.

Ejs.: coche, número complejo, empleado, empresa, curso de formación, pedido, etc.

Características de un objeto

- identidad: lo distingue de otros objetos
- estado: características o atributos con unos valores determinados
- comportamiento: operaciones que puede realizar

Clase

Tipo de datos definido por el programador que describe de un grupo de objetos con análogos atributos y comportamiento.

Las clases en C++ tienen un nombre o identificador, un conjunto miembros (miembros dato y métodos o funciones miembro) y unos niveles de acceso, especificados mediante `public:`, `protected:` o `private:`, que protegen ciertas partes. Por defecto, en una clase, todo es privado, las estructuras (`struct`) son análogas a las clases pero en ellas, por defecto, todo es público.

Estructura de una clase

```
class <nombre de la clase>
{
   [public:
   [ <miembros_dato> ]
   [ <funciones_miembro> ] ]
   [protected:
   [ <miembros_dato> ]
   [ <funciones_miembro> ] ]
   [private:
   [ <miembros_dato> ]
   [ <funciones_miembro> ] ]
};
```

Ejemplo de clase

```
class fecha
{
   short dia, mes, anno;
  public:
   void setdia(short d=(short)1){dia = d;}
   void setmes(short m=(short)1){mes = m;}
   void setanno(short a=(short)1900){anno = a;}
   void mostrar()
   { cout << (dia < 10 ? "0" : "") << dia << "/";
     cout << (mes <10? "0" : "") << mes << "/";
     cout << anno << endl;
   }
};
```

Otra forma de declaración

```
class fecha
{
   short dia, mes, anno;
  public:
   void setdia(short =(short)1);
   void setmes(short =(short)1);
   void setanno(short =(short)1900);
   void mostrar();
};

int main()
{
    // ....
    system("pause");
    return EXIT_SUCCESS;
}
void fecha::setdia(short d)
{
     dia=d;
}
//…
void fecha::mostrar()
{
  cout << (dia < 10 ? "0" : "") << dia << "/";
  cout << (mes <10? "0" : "") << mes << "/";
  cout << anno << endl;
}
```

Programación modular

En un módulo se definirá una única clase o clases muy relacionadas.

Diferentes clases relacionadas se empaquetan en bibliotecas (o librerías).

Las partes de un módulo en C++ son:

- Archivo de encabezado (declaración, cabecera o Interfaz) de extensión `.h` o `.hpp`
- Archivo de implementación (cuerpo del módulo) de extensión `.cpp`

Archivos de encabezado

Los archivos de cabecera contienen las declaraciones de constantes, variables y funciones de las que consta el módulo, así como llamadas a otros archivos de encabezado necesarios.

Las directivas

```
#ifndef _Nombre_
#define _Nombre_
…
#endif
```

se utilizan para impedir que se produzcan errores de compilación cuando un fichero de definición es importado varias veces por el mismo módulo

Archivos de implementación

Los archivos con extensión `.cpp` implementan las declaraciones efectuadas en el archivo `.h` e inicializan las variables de clase.

Deben importar el fichero de declaraciones mediante la directiva `#include`.

Si una función es `inline`, su implementación debe efectuarse en el archivo `.h`

Operaciones fundamentales: Instanciación

```
class fecha
{
  //…
}f1;
int main()
{
    fecha f2;
    fecha* f3 =new fecha();
    //…
}
```

Operaciones fundamentales: Paso de mensajes

El mensaje debe contener:

- El nombre del objeto seguido de `.` o el nombre del puntero al objeto seguido de `->`
- el nombre de la operación o método
- Uno o una lista de parámetros.

Ejs.:

```
f1.setdia();
f2.setanno(2005);
f1.setanno(2008);
f1.mostrar();
f3->mostrar();
```

Operaciones fundamentales: Destrucción de objetos

El destructor es llamado automáticamente cuando un objeto termina su vida, que puede ser:

- Cuando se termina de ejecutar el bloque en el que un objeto ha sido definido (objetos locales)
- Al final del programa (objetos globales, `static` o creados con `new` y no eliminados por el programador)
- Para objetos creados con `new`, cuando el programador lo establezca mediante `delete` (los objetos creados dinámicamente con `new` deben ser eliminados por el programador cuando ya no se necesiten utilizando `delete`), ej. `delete f3;`

Clasificación de los miembros por su accesibilidad

- **Públicos, privados, protegidos**
- Flexibilización de las restricciones mediante **relaciones de amistad**. La relación de amistad no es transitiva.

  Los miembros de una clase A, declarada amiga de otra clase B, pueden acceder a todos los miembros de B, ya sean públicos, privados o protegidos

```
class B
{
  // miembros de B
  friend class A;
}
```

  También es posible definir una función como amiga de una clase, lo que le permite acceder a todos los miembros de la clase

Clasificación de los miembros por su vinculación con la clase

- **Estáticos** (ligados a la clase)
  - Los miembros dato `static` actúan como datos globales asociados a una clase y, si son variables, se tienen que inicializar en el ámbito global (por ejemplo, inmediatamente antes de la función main)
  - Los métodos `static` acceden a los miembros dato estáticos.
- **Ligados a los objetos.**

Definición de funciones miembro.

- **Cabecera**:
  - tipo de valor devuelto (puede no haber)
  - nombre
  - lista de parámetros formales (de 0 a n)
- **Cuerpo**:
  - datos:
    locales
    de instancia (datos miembro de la clase).
  - secuencia de instrucciones
  - sentencia de regreso: `return`. (puede no haber)

Miembros

```
class X
{ public:
    X (int v)
    { valor = v;
      numObjetos++;}
    int  getValor() const { return valor; }
    static int cuantos() { return numObjetos; }
    static void Reinicializar() { numObjetos = 0; }
  private:
    int valor;
    static const float PI=3.141592F;
    static int numObjetos;};

int X::numObjetos = 0; // inicialización de variables static

int main()
{
  X a(3);  X* b = new X(4);
  cout << "Número de Objetos: " << X::cuantos() << endl;
  X::Reinicializar();
  //…
}
```

Funciones miembro `inline`

- Sugerencia al compilador para que sustituya la llamada a la función por el código, ahorrando así el tiempo de llamada y de copia de argumentos pero aumentando la longitud del programa.
- Se pueden declarar de dos formas
  - Definiendo la función en la declaración de la clase
  - Anteponiendo la palabra `inline` a la declaración de la función. La implementación debe ponerse en mismo archivo donde se declara la clase, aunque se haga fuera de ella.

Las funciones miembro y el calificador `const`

- En la lista de parámetros implica que el parámetro no es modificable en la función.
- Después de la lista de parámetros tanto en la declaración como en la definición de una función implica declarar la función como constante. Las funciones constantes no pueden modificar los datos miembros. Se utilizan para operar sobre objetos `const`

Ejecución de funciones miembro

Una función miembro no declarada `static` es un algoritmo que se ejecuta cuando se la llama con el punto o la flecha mediante un objeto. Puede acceder a todos los miembros de la clase y a sus argumentos.

Un método (función miembro) `static` puede llamarse de la forma habitual pero es más típico llamar a estas funciones sin especificar ningún objeto, utilizando el nombre de la clase y el operador de resolución de rango. Una función miembro `static` no puede ser declarada virtual, ni acceder a los datos miembro ordinarios, sólo a los datos miembro `static`.

Sobrecarga de métodos

- **Sobrecarga**.

  En una misma clase pueden definirse varios métodos (funciones miembro) con el mismo nombre siempre y cuando tengan diferente el número, orden o tipo de los parámetros.

- **Invocación de métodos sobrecargados**

  Cuando se llama a un método sobrecargado el compilador determina qué versión se ejecuta mediante el análisis de los parámetros que intervienen en la invocación

Métodos o funciones miembro. Clasificación por su objetivo I

- **Constructores**

  Método especial que se utiliza para inicializar un objeto (inicialización de variables miembro o ejecución de código inicial) cuando se crea, ya sea por declaración ó dinámicamente con el operador `new`.

- **Características de los constructores en C++**

  Deben ser públicos, se pueden sobrecargar, tienen el mismo nombre que la clase, no retornan ningún valor y no pueden ser heredados ni declararse virtuales.

  Si no se declara constructor para una clase, el compilador crea un constructor *de oficio* sin argumentos.

  Los constructores de objetos constantes no requieren el modificador `const`

Clasificación de las funciones miembro por su objetivo II

- **Destructores**

  Los destructores se encargan de funciones de limpieza y liberación de memoria cuando se destruye un objeto (su ámbito acaba ó se libera explícitamente con el operador `delete`)

- **Características de los destructores en C++**

  Un destructor es un método que debe ser público, tiene el mismo nombre de la clase precedido por el símbolo ~, carece de tipo devuelto y no tiene argumentos. Es conveniente declarar los destructores virtuales en las clases abstractas.

  Solo puede existir un destructor por clase. Si no se declara el compilador crea un destructor por defecto.

Constructores y destructores definidos por el programador in-line y off-line

```
class fecha
{
  //…
  public:
  /* constructor por
     defecto in-line */
  fecha()
    {
       //…
    }
  //destructor in-line
  ~fecha()
  {
    //…
  }
};
```

```
class fecha
{
  //…
  public:
  //constructor por defecto
  fecha();
  // destructor
  ~fecha();
  //…
};
// definiciones off-line
fecha::fecha()
{
  //…
}
fecha::~fecha()
{
  //…
}
```

Clasificación de las funciones miembro por su objetivo III

- **Selectores**.

  Devuelven valores de miembros dato.

- **Modificadores**.

  Cambian contenidos de miembros dato.

- **Operadores**.

  Definen operaciones estándar para objetos de la clase.

- **Iteradores**.

  Procesan colecciones de objetos.

Declaración de una clase C++

```
class Complejo
{  private:
     double parteReal;
     double parteImag;
   public:
     Complejo();
     Complejo(const Complejo& c);      //constructores
     Complejo(double x, double y);
     ~Complejo();                      //destructor
     void SetReal(double x);
     void SetImag(double y);
     double GetReal();
     double GetImag();
     Complejo operator+(const Complejo&);
     Complejo operator-(const Complejo&);
     ...
};
```

Implementación de funciones miembro en C++

```
Complejo::Complejo()
{
    parteReal=parteImag=0;
}
Complejo::Complejo(double x, double y)
{
    parteReal=x;
    parteImag=y;
}
Complejo::Complejo(const Complejo& c)
{
    parteReal=c.parteReal;
    parteImag=c.parteImag;
}
void Complejo::SetImag(double y)
{
    parteImag=y;
}
double Complejo::GetReal()
{
    return parteReal;
}
```

Constructor por defecto

Es un constructor sin parámetros que resulta necesario para:

efectuar declaraciones del tipo `Profesor profeMatematicas;`

o crear un vector de objetos `Alumno alumnos[100];`

Puede ser

- **de oficio**, establecido por el compilador cuando el programador no define **ningún constructor**.
- **definido por el programador,** la definición puede hacerse

  ```
  Complejo::Complejo(){pReal=pImag=0;}
  ```

  Usando inicializadores (de interés para inicializar datos miembro constantes):

  ```
  Complejo::Complejo(): pReal(0),pImag(0){}
  ```

Constructor de copia

Clase especial de constructor con argumentos que tiene un único argumento que es una referencia constante a un objeto de su misma clase. Se utiliza cuando se crea un objeto a partir de otro de su misma clase y se llama implícitamente cuando a una función se le pasan objetos como argumentos por valor y cuando una función tiene un objeto como valor de retorno.

Llamada: `Complejo c2(c1);`

Puede ser:

- **de oficio**, establecido por el compilador cuando el programador no define **ningún constructor de copia**.
- **definido por el programador**. Si entre los datos de una clase hay punteros o variables estáticas es necesario que el programador defina el constructor de copia.

this

variable predefinida en todas las funciones miembro de una clase que contiene la dirección del objeto al que se está aplicando una función.

```
// Ejemplo uso de this
#include <iostream>
#include <string>
using namespace std;

class empleado
{ string nombre;
  float salario;
  public:
  empleado (string nombre, float salario)
  {    this->nombre  = nombre;
       this->salario = salario; }
  empleado incrementarSalario()
  {    salario = float(salario+0.05*salario);
       return *this;  }
  void mostrarSalario()
  {    cout << nombre << ": " << salario <<endl;  }
};

int main()
{  empleado e("Pedro", 1675);
   e.incrementarSalario().mostrarSalario();
   system ("PAUSE");
   return 0;}
```

¿Preguntas?