Curso C++ - Clase 2

Juan Antonio Zubimendi
azubimendi@lifia.info.unlp.edu.ar

LIFIA

17 de noviembre de 2010

Arreglos

Los arreglos se definen de la siguiente manera:

```
<tipo> nombre_arreglo[<tamaño>];
```

Se puede inicializar el arreglo si se desea. El indice del primer elemento es 0. El *tamaño* puede ser ser un valor constante, o una expresión entera no negativa.

```
int n = 5;
char caracteres[n];
```

typedef

Define un alias de tipo. No se define un tipo de datos nuevo, sino otro nombre para el mismo tipo de datos.

```
typedef tipoExistente tipoNuevo;
```

enum

Define un tipo de datos enumerado.

```
enum <nombre> {
    PRIMERO,
    SEGUNDO = 1,
    TERCERO,
    CUARTO
};
```

El tipo será un alias de *int*, los valores seran autoincrementales. Si no se asigna ningún valor inicial, el primer valor será 0.

struct

Definen tipos de datos propios. Definen un conjunto de campos y/o funciones propias.

```
struct Cuadrado {
  int lado;
  int area() { return lado * lado; }
};
```

Puedo acceder a los miembros de una estructura:

```
Cuadrado c;
c.lado = 5;
```

Clases

Para definir una clase usamos:

```
class Punto {
  private:
    int x, y;
   public:
    int getX() { return x;}
    int getY() { return y;}
};
```

Visibilidad

Cada miembro de una clase puede tener una de estas tres visibilidades:

- Public: El acceso a este miembro de la estructura es irrestricto.
- Private: Solo se puede acceder dentro de la misma clase.
- Protected: La clase y sus subclases pueden acceder a este miembro.

Dentro de la declaración de la clase se puede definir la visibilidad escribiendo la misma seguida por dos puntos para indicar que de ahí en adelante todo lo que se declare tendrá esa visibilidad. Si no se indica nada la visibilidad será *Private*.

struct y *class* son totalmente equivalentes, la única diferencia es la visibilidad que ofrecen por omisión.

Métodos

Los métodos se pueden definir en la misma declaración de la clase. Pero es preferible definir una función fuera de la declaración de la clase.

```
class Cuadrado {
    Cuadrado(int lado);
  private:
    int _lado;
  public:
    int area();
};
```

Métodos

Una vez definida la clase, hacemos:

```
int Cuadrado::Cuadrado(int lado) {
     _lado = lado;
}

int Cuadrado::area() {
    return lado * lado;
}
```

E

s conveniente separar la declaración y la definición en archivos diferentes.

Constructores

Cada clase posee por lo menos un constructor.
Si no indicamos uno especificamente, se creara uno sin parámetros.
Los constructores son funciones de una clase con el mismo nombre de la clase.
Podemos tener más de un constructor, es decir podemos sobrecargarlo.
Creamos instancias de un objeto invocando a uno de los constructores de la clase.
En el constructor podemos reservar e inicializar los aspectos del objeto que sean necesarios.

```
class Figura {
  public:
    Figura(int lados);
```

Constructores importantes

Destructores

Cuando un objeto va a ser destruido se invoca al destructor de la clase.
El destructor es uno solo, a diferencia de los constructores.
El destructor no toma parámetros ni devuelve parámetros.
El nombre del método será un seguido del nombre de la clase.

A

cordarse de liberar recursos utilizados por la clase.

Variables y Métodos de Clase

Los métodos y variables de clase, se definen de la misma manera que los de instancia, la única diferencia es que se antepone la palabra clave *static* antes de la declaración del método o variable.

```
class Figura {
  public:
      Figura();
      static int figurasTotales();

  private:
    static int _totalFiguras;

};

Figura::_totalFiguras = 0;
int figurasTotales() {
    return _totalFiguras;
}
```

Uso de const

const afecta de diferentes maneras a las expresiones donde aparece.

- Como modificador - antes - de una variable, es una constante.
- Como modificador - antes - de un párametro, que ese parámetro no se puede modificar.
- Al final la declaración de un metodo, que dicho método no modifica el estado del objeto. Esto le permite al compilador realizar optimizaciones.

Referencias

Una referencia es un alias de una variable.
Definimos una referencia de la siguiente manera:

```
int entero;

int & referencia = entero;
```

Una vez asociada una referencia con una variable, no se puede cambiar. En el momento de definir la referencia hay que asociarla a la variable a la que hacemos referencia.
Dentro de una función o método podemos retornar una referencia a una variable.

O

JO! Con las referencias a las variables locales

Referencias

Hasta ahora habiamos visto que todas las pasajes de parámetros a funciones o métodos eran por valor, ahora vemos que también podemos pasar parámetros por referencia:

```
Matriz multiplicar(Matriz &m, Matriz &n);
```

Si no pasaramos por referencia, los objetos

Uso de static

El modificador *static* se puede utilizar

- Como modificador - antes - de una variable global o función, esta función o variable no se podrá utilizar fuera del archivo donde este definido.
- Como modificador - antes - de una variable local a una función o método, esta variable no se destruirá cuando la función termine, persistiendo su valor a través de las diferentes llamadas a la misma.

Trabajando con multiples archivos

Es conveniente

- Separar cada objeto en un archivo distinto, para
- Separar la implementación de la interfaces de los objetos.

Separar implementación de interfaz

Sabiendo que C++ separa claramente la definición de la declaración de tipos de datos y de funciones, podemos usar eso para realizar la separación.

- En archivos de encabezados "*headers*": Pondremos las definiciones. Estos archivos suelen llevar la extensión *.h*.
- En archivos de implementación escribiremos las definiciones, pero incluyendo previamente las declaraciones necesarias.
- Si existe una dependencia de una clase con otra, podremos incluir el respectivo archivo de encabezado para que el compilador pueda acceder correctamente a todas las clases.

Archivo de encabezados

- Estos archivos suelen tener la extensión *.h* y llevan el mismo nombre que el archivo de implementación.
- Aqui escribiremos todas las definiciones que necesitemos usar del archivo de implementación.
- Los archivos de encabezado se incluyen con la directiva de preprocesador *#incldue*.

Guardas en los archivos de encabezados

a.h:

```
#include "b.h"

struct A
{
  struct B *b;
} A;
```

b.h:

```
#include "a.h"

typedef struct B
{
  struct A *a;
} B;
```

Guardas en los archivos de encabezados

Para evitar esto, usamos guardas sobre los archivos de encabezados.

```
#ifndef __A_H__
#define __A_H__

#include "b.h"

struct A
{
  struct B *b;
} A;

#endif
```

b.h:

Archivo de implementación

- Estos archivos suelen tener la extensión *.cpp*, o *.cxx*
- Este archivo incluye (mediante *#include*) el archivo de encabezado propio y los demas archivos de encabezados que necesite.

Google Test

Vamos a ver como utilizar una librería de Test de Unidad de C++.
¿Por qué Google Test?

- Porque está basada en la arquitectura xUnit (sUnit, jUnit...)
- Porque es fácil de usar
- Porque es software libre

La página web del sitio está en
http://code.google.com/p/googletest/

Instalación de Google Test

Los pasos son:

- Descargar la librería
- Descomprimirla, ejecutando: *tar xjf gtest-1.5.2.tar.bz2*
- Compilarla, ejecutando: *make*
- Instalarla, ejecutando: *make install*

Instalación de Google Test

En el curso le proveemos un ejemplo completo basado en el tutorial.
La Wiki del proyecto posee una buena introducción:
http://code.google.com/p/googletest/wiki/Primer