Curso C++ - Clase 3

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

LIFIA

22 de noviembre de 2010

Memoria dinámica

Existen dos tipos de memoria de dinámica disponibles en C++:

- *Stack*: Esta memoria se utiliza para las variables locales a las funciones y métodos. Este memoria será devuelta cuando la función finalice.
- *Heap*: Con esto podemos obtener memoria para variables que necesitemos más allá del alcance de una función.

Heap

Para obtener y liberar memoria dinámica (de la Heap) tenemos dos instrucciones:

- *new*: Con esta instrucción pedimos memoria dinámica al sistema.
- *delete*: Con esta instrucción devolvemos al sistema la memoria asignada a una variable

Podemos obtener memoria para:

- Para crear objetos
- Para tipos básicos u otro tipo de datos.

IMPORTANTE

La administración de la memoria pedida con los operadores *new* y *delete* es total responsabilidad del programador.

new

Con el operador *new* podemos pedir memoria para un objeto o tipo de datos.
El operador nos devuelve un puntero a la entidad recién construida.
De ser necesario se pueden pasar los parámetros necesarios para el constructor.

```
Casa *pCasa = new Casa;
```

delete

Con el operador *delete* liberamos la memoria apuntada por el puntero que recibe como parámetro. El destructor correspondiente a ese objeto es llamado y la memoria liberada.

Punteros

Un puntero nos permite un acceso indirecto a un objeto al cual esta apuntando. Dependiendo al tipo de objeto que apunte, será el tipo de puntero. Para definir un puntero usamos $*$ antes del nombre de la variable:

```
int * pEntero = new int;
```

¿Cómo accedemos al valor "apuntado"?
El acceso al valor apuntado se llama desreferenciar y se puede hacer con el operador $*$:

```
*pEntero = 5;
```

Punteros - continuación

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

Punto * p = new Punto(10,10);
```

Punteros - continuación

Como puedo invocar al metodo *getX* de *p* ?

```
std::cout "Pos X: " << (*p).getX() << std::endl;
```

Notar el parentesis, para que el operador "*" se aplique antes que el operador ".".
Como el acceso a miembros de una estructura o clase a partir de un puntero es una operación común, tenemos una manera abreviada de hacerlo:

```
std::cout "Pos X: " << p->getX() << std::endl;
```

Punteros - continuación

Existe un valor especial que se le puede asignar a un puntero, de manera tal que cual intento de desreferenciarlo nos produzca un error.
Este valor es *0*, a este valor se lo suele llamar puntero nulo (NULL POINTER). En nuestro programa podemos usar la constante NULL como equivalente del puntero nulo.

Cuando terminamos de usar un puntero o cuando todavía no tiene un valor definido, es buena costumbre asignarle el valor 0 a un puntero. De esta manera podemos saber si tiene un valor valido o no.

Problemas con Punteros

Los punteros suelen ser la fuente de los problemas más comunes de programación en C++:

- Acceder a un puntero no inicializado o no valido (memoria ya devuelta al sistema).
- Perder o cambiarle el valor a un puntero que ya tiene un valor asignado.

El acceso a un puntero no valido puede provocar un comportamiento indifinido muy díficil de encontrar, inicializando correctamente los punteros y cargando el valor de puntero nulo si no tiene un valor actual es buena practica.
El perder el acceso a una variable dinámica nos produce una perdida de memoria (leak).

new [] y delete []

El operador emphnew[] nos permite pedirle al sistema opertivo más de un elemento a la vez. Es muy importante utilizar el operador *delete[]*, en lugar de *delete* para liberar la memoria.

```
int *enteros = new int[10];
for (int i = 0; i < 10; i++)
   enteros[i] = i;
  ...
delete [] enteros;
```

¿Alguien nota algo raro?

En el ejemplo anterior utilizamos al puntero como si fuera un arreglo. En C++, es común utilizar este tipo de notación. Esta es la manera alternativa de representarlo:

```
int *enteros = new int[10];
for (int i = 0; i < 10; i++)
   *(enteros + i) = i;
  ...
delete [] enteros;
```

Leaks

Las perdidas de memoria se producen cuando no liberamos toda la memoria que hemos pedido al sistema operativo. Las causas mas comunes son:

- No hacer *delete* de todos los objetos alocados.
- Utilizar *delete* en lugar de *delete[]*.
- Pedir memoria dentro de una función o método y olvidarnos de liberar la memoria.
- Olvidarnos de liberar toda la memoria pedida en un destructor de una clase.

operador this

Este operador es valido solamente dentro de cualquier método de instancia de cualquier clase.
Apunta a la instancia (es un puntero) específica del objeto que esta ejecutando el método.
Puede ser usado como cualquier otra variable de instancia.

Subclases

Para definir una subclase lo hacemos de la siguiente manera:

```
class Cuadrado : public Figura {
  public:
       Cuadrado();
};
```

De esta manera *Cuadrado* se define como subclase de *Figura*, teniendo acceso a metodos y variables de instancia que hereda de la siguiente manera:

- Los campos *protected* de la clase base serán accesibles a los metodos de la clase
- Los campos *public* de la clase base serán accesibles desde los metodos y a los usuarios de los objetos.

Herencia multiple

C++ nos ofrece herencia multiple para los objetos. Si bien no es muy común y en la mayoría de los casos recomendamos no usarla.

```
class Avion : public Transporte,
              public BuscarMejorEjemplo {
  public:
     Avion();
};
```

Punteros y Polimorfismo

C++ nos deja aprovechar el carácter polimorfico de los objetos a través de los punteros. A un puntero de una clase base puede apuntar a un objeto de esa clase o de cualquier subclase.

Punteros y Polimorfismo - Ejemplo

```
#include <iostream>
class A {
public:
    void imprime() {
        std::cout << "hola, soy A" << std::endl;
    }
};

class B : public A {
public:
   void imprime() {
      std::cout << "hola, soy B" << std::endl;
   }
}
```

Punteros y Polimorfismo - Ejemplo

```
void imprimir(A *objeto) {
    std::cout << "objeto metodo imprime()" << std::endl;
    objeto->imprime();
}
int main(int argc, char **argv) {
    A *objetoA = new A;
    B *objetoB = new B;
    imprimir(objetoA);
    imprimir(objetoB);

    delete objetoA; delete objetoB;
    return 0;
}
```

Métodos virtuales

¿Qué pasó?
Por cuestiones de eficiencia, las busquedas de los métodos polimorficos en C++ no se buscan en las subclases. Para indicarle al compilador de C++ que un método es polimorfico debemos anteponer al mismo la palabra clave “virtual”. En C++ llamamos a estos métodos virtuales.

Punteros y Polimorfismo - Ejemplo Corregido

```
#include <iostream>
class A {
public:
    virtual void imprime() {
        std::cout << "hola, soy A" << std::endl;
    }
};

class B : public A {
public:
   void imprime() {
      std::cout << "hola, soy B" << std::endl;
   }
}
```

Clases abstractas

Cuando diseñamos clases abstractas, podemos querer dejar la implementación de una función a las subclases. Podemos hacerlo igualando a cero el método que queremos marcar como abstracto.

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

     virtual int area() = 0;
  };
```

Obviamente, no podremos instanciar un objeto de una clase que contenga por lo menos un método abstracto.

Interfaces

Si bien C++ no nos ofrece interfaces como Java, podemos lograr un resultado similar haciendo:

- Definimos la interface como una clase abstracta con todos los mensajes que queremos que tenga la interfaz. Cada uno de estos métodos los indicamos como abstractos.
- A la clase que queremos que tenga esa interfaz, la hacemos heredar a partir de la clase creada en el paso anterior.