Constructores, destructores y parámetros por referencia en C++

Elaborado por: Ukranio Coronilla

Debido a que es muy común inicializar variables dentro del constructor, se dispone de otra forma que ocupa menos líneas de código y que mostraremos a continuación. Suponga que se tiene el siguiente constructor:

```
Fecha::Fecha(int dd, int mm, int aaaa)
{
   mes = mm;
   dia = dd;
   anio = aaaa;
}
```

La siguiente es una forma equivalente:

```
Fecha::Fecha(int dd, int mm, int aaaa) : dia(dd), mes(mm), anio(aaaa)
{/*Vacio intencional*/}
```

Es recomendable hacer en los constructores una validación para verificar que los datos pasados en el argumento son correctos. Por ejemplo:

```
Fecha::Fecha(int dd, int mm, int aaaa) : dia(dd), mes(mm), anio(aaaa)
{
   if((mes < 1) || (mes > 12)){
      cout << "Valor ilegal para el mes!\n";
      exit(1);
   }
   …
}
```

Ejercicio 1: Pruebe la nueva forma de declarar un constructor en el código del programa 2-2 y añada el código para verificar que los datos pasados en el argumento sean correctos (sólo valide 1< dia 31, y 0 < año < actual) . Pruebe que funciona la validación.

Un parámetro pasado por referencia es más eficiente que un parámetro por valor, sobre todo cuando se trata de una clase. Por ello se recomienda pasar objetos por referencia aun cuando la función no modifique el valor del objeto.

En C++ el paso de parámetros por referencia se logra añadiendo el símbolo & entre el tipo de dato y la variable; tanto en la declaración de la función como en el encabezado de la definición de la función. Al compilador no le importa la posición del &, podría estar junto a la variable o junto al tipo de dato o entre ambas.

Ejercicio 2: Con ayuda del **comando** `time` (véase con man) y el programa 2-1, elabore un programa para probar que los datos pasados por referencia tienen mejor desempeño que los que

se pasan por valor. Para esto elabore las funciones con parámetros por valor y por referencia cuyas declaraciones son las siguientes:

```
int masVieja(Fecha fecha1, Fecha fecha2);
int masVieja(Fecha &fecha1, Fecha& fecha2);
```

La función devuelve 1 si fecha1 > fecha2, 0 si son iguales y -1 si fecha1 < fecha2. La función debe llamarse n veces en un ciclo, y los valores miembro deben asignarse con números aleatorios antes de llamar a la función (véase la función rand()). Recomendación: Para hacer notoria la diferencia además de mandar llamar muchas veces a la función, es importante que el objeto `Fecha` tenga un tamaño suficientemente grande. La idea es que la diferencia de tiempo entre el paso por valor y el paso por referencia se encuentre en el orden de los segundos.

Ejercicio 3: Realice la misma actividad que en el ejercicio 2 pero utilizando el paso por referencia al estilo del lenguaje C, con la declaración de función siguiente:

```
int masVieja(Fecha *fecha1, Fecha *fecha2);
```

Las variables que hacen reserva de memoria dinámica mediante las palabras reservadas `malloc` en C o `new` en C++ tienen el inconveniente de no liberar el espacio, aún si dicha variable se reservó dentro de una función y esta ha terminado. Para liberar la memoria dinámica es necesario llamar a la función `delete`.

Liberar memoria es muy importante en las clases, porque un programador que utilice dicha clase no tiene conocimiento ni acceso a las variables dinámicas dentro de la clase, y puede terminar por consumir la memoria de la computadora al crear n objetos de dicha clase.

Para estos casos C++ incluye una función miembro especial conocida como destructor el cual debe tener el mismo nombre que el constructor y además debe ser antecedido con una tilde.

Para probar lo anterior compile y ejecute el siguiente código:

```
#include <unistd.h>
#include <iostream>
using namespace std;

class NumerosRand
{
private:
        int *arreglo;
        unsigned int numeroElementos;
public:
        NumerosRand(unsigned int num);
        void inicializaNumerosRand(void);
};

NumerosRand::NumerosRand(unsigned int num)
{
        numeroElementos = num;
        arreglo = new int[numeroElementos];
```

```
}

void NumerosRand::inicializaNumerosRand()
{
        unsigned int i;

        for(i = 0; i < numeroElementos; i++)
                arreglo[i] = rand();
        return;
}

void pruebaClase(){
        unsigned int capacidad;

        cout << "Numero de enteros aleatorios en el arreglo : ";
        cin >> capacidad;
        NumerosRand tmp(capacidad);
        tmp.inicializaNumerosRand();
        sleep(7);
}

int main()
{
        char res;

        do
        {
                pruebaClase( );
                cout << "Probar de nuevo? (s/n) ";
                cin >> res;
        } while ((res == 's') || (res == 'S'));
}
```

Observe que básicamente este programa manda a llamar a la función `pruebaClase()` , la cual crea un objeto de la clase `NumerosRand` que contiene un arreglo dinámico de enteros, cuyo tamaño es especificado por el usuario. Dicho arreglo se llena con números aleatorios, espera siete segundos y después termina para preguntar al usuario si desea volver a llamar a la función `pruebaClase()`.

Abra su monitor de sistema en Linux y ponga atención en la pestaña “Recursos” de la cantidad de memoria RAM ocupada hasta el momento. Ejecute el programa y reserve un arreglo con cien millones de enteros .

¿Qué sucede con la cantidad de memoria ocupada en el monitor de sistema?

¿Qué pasa si no se llama al método que inicializa el arreglo de enteros? ¿Por qué?

¿Qué sucede con la memoria si se llama dos o más veces a ejecutar la función `pruebaClase()`?

¿Qué sucede con la memoria si en el programa primero requiero un arreglo de cien millones de elementos y después solo necesito un arreglo con cien elementos?

Para resolver los problemas ocasionados con la reserva dinámica de memoria debemos incluir el método destructor siguiente:

```
NumerosRand::~NumerosRand(){
	delete[] arreglo;
}
```

Y en la interfaz:

```
~NumerosRand(); // Destructor
```

Pruebe que funciona el destructor de manera adecuada.