Metodologías y Técnicas de Programación II

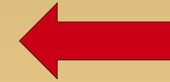
Programación Orientada a Objeto (POO) C++

Inicialización y Limpieza II

Estado del Programa

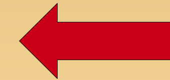
Introducción a la POO

- Historia de la Programación
- Conceptos de POO
- C++
- Mi primera Clase



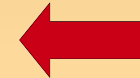
Repaso de Conceptos

- Estándares de Programación
- Punteros y Memoria
- E/S
- Control y Operadores



Clases y Objetos en C++

- Uso y aplicación
- Constructores
- Funciones Amigas
- Constantes e “inline”



Sobrecarga

- De Operadores
- De Funciones

Herencia.

- Tipos de Visibilidad
- Herencia Múltiple

Polimorfismo

- Funciones Virtuales
- Polimorfismo y Sobrecarga.

Plantillas

- Contenedores
- Iteradores

Inicialización y Limpieza de Objetos

El constructor es una función sin tipo de retorno y con el mismo nombre que la estructura. **Alumno()**

El destructor tiene la misma forma, salvo que el nombre va precedido el operador "~". **~Alumno()**

No devuelven nada. Es diferente a devolver **void**.

En general deben ser públicos.

Podemos tener varios constructores (usando sobrecarga de funciones)

Si sólo tenemos un constructor estilo: Alumno (int nota), estamos obligados a pasar un entero en la creación de cada objeto.

7.1 Repaso de Sesiones Anteriores

Constructores

Modo simplificado de inicialización:

int i(0);   Alumno   a("Pepe");

Inicializadores:

```
pareja::pareja(int a, int b)
{
   a_ = a;
   b_ = b;
}
```

```
// Es más seguro y eficiente así:
pareja::pareja(int a,int b) : a_(a), b_(b) {}
```

```
class Pareja
{
...
  // Constructores
  Pareja(int a, int b);
  Pareja() : a_(0), b_(0){};
....
```

```
class Pareja
{
...
// Argumentos por defecto
Pareja(int a=0,int b=0) :a_(0),b_(0){};

....
```

7.2 Constructores

Constructor Copia

Crea un objeto a partir de otro objeto existente.

Estos constructores sólo tienen un argumento, que es una referencia a un objeto de su misma clase.

```
// Constructor Copia para Alumno
Alumno::Alumno(const Alumno &obj_alumno);
```

```
class Pareja
{
...
// Constructor
pareja(int a2=0, int b2=0) : a(a2), b(b2) {}
// Constructor copia:
pareja(const pareja &p);

....
```

```
pareja::pareja(const pareja &p) : a(p.a), b(p.b) {}
pareja par1(12, 32)
// Uso del constructor copia:
// par2 = par1
pareja par2(par1);
int x, y;
par2.Lee(x, y);
cout <<"Valor par2.a: " << x << endl;
cout <<"Valor par2.b: " << y << endl;
```

7.2 Constructores

Inicialización de tipos agregados

Arrays, estructuras y clases: tipos agregados. El array es de un sólo un tipo

```
// Si ponemos más valores error de compilación
int a[5] = { 1, 2, 3, 4, 5 };
// Todos los elementos a cero.
int a[5] = {0};
// Contexto automático – No explicitamos los límites
int c[] = { 1, 2, 3, 4 };
class X
{
   int i_;
   char c_;
public:
    X(int i,char c): i_(i), c_(c) {}
};
X x1[] = { X(1,'a'), X(2,'b'}; // Tres objetos y tres llamadas
```

7.2 Constructores

Constructores por defecto

Es un constructor que puede ser invocado sin argumentos

```
class X
{
   int i_;
   char c_;
public:
    X(int i,char c): i_(i), c_(c) {}
};
X x1[10]; // El Compilador se va a quejar!!
X x3;
```

El constructor por defecto es tan importante que si (y sólo si) una clase no tiene constructor el compilador crea uno automáticamente (no hace nada).

```
class V {
  int i; // private
}; // No constructor

  V v, v2[10];          // Esto funciona!!
```

7.2 Destructores

Destructores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase, liberando la memoria utilizada por dicho objeto (o para otras cosas como cerrar un fichero abierto.

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido.

Si el objeto se creó con **new** se llama al constructor al invocar **delete**.

¿Cuándo definimos un destructor?

En general cuando nuestra clase tenga datos miembro de tipo puntero.

El destructor no puede sobrecargarse, por la sencilla razón de que **no admite argumentos.**

7.2 Destructores

Uso de Destructores

```
class Cadena
{
   public:
    cadena();         //  Constructor por defecto
    cadena(char *c); //  Constructor desde cadena c
    cadena(int n);    //  Constructor de cadena de n caracteres
    cadena(const cadena  &);    // Constructor copia
    ~cadena();        //  Destructor
    void Asignar(char *dest);
    char *Leer(char *c);
   private:
    char *cad_;        // Puntero a char: cadena de caracteres
};
```

7.2 Destructores

Uso de Destructores

```
Cadena::cadena() : cad_(NULL) {}
Cadena::cadena(char *c)
{
   cad_ = new char[strlen(c)+1];// Reserva memoria para cadena
   strcpy(cad_, c);              // Guardamos la cadena
}
Cadena::cadena(int n)
{
   cad_ = new char[n+1]; // Reserva memoria para n caracteres
   cad_[0] = 0;          // Cadena vacía!!
}
Cadena::cadena(const cadena &cad)
{
   // Reservamos memoria para la nueva y la almacenamos
   cad_ = new char[strlen(cad.cad_)+1];
   // Reserva memoria para cadena
   strcpy(cad_, cad.cad_);                // Guardamos la cadena
}
```

7.2 Destructores

En acción:

```
Cadena::~cadena()          // Destructor
{
   delete[] cad_;          // Libera la memoria reservada a cad_
}
void Cadena::asignar(char* dest)
{
   // Eliminamos la cadena actual:   ¿Por qué?
   delete[] cad_;
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(dest)+1];
   // Reserva memoria para la cadena
   strcpy(cad, dest);              // Guardamos la cadena
}
char* Cadena::leer(char* c)
{
   strcpy(c, cad_);
   return c;
}
```

7.2 Destructores

En Acción:

```
{
  Cadena  cadena1("Cadena de prueba");
  Cadena  cadena2(Cadena1); // Cadena2 es copia de Cadena1
  Cadena* cadena3;          // Cadena3 es un puntero
  char c[256];
  // Modificamos cadena1:
  cadena1.asignar("Otra cadena diferente");
  // Creamos cadena3:
  cadena3 = new Cadena("Cadena de prueba nº 3");
  // Ver resultados
  cout << "cadena 1: " << cadena1.leer(c) << endl;
  cout << "cadena 2: " << cadena2.leer(c) << endl;
  cout << "cadena 3: " << cadena3->leer(c) << endl;
  delete cadena3; // Destruir Cadena3.
  // Cadena1 y Cadena2 se destruyen automáticamente
  return 0;
}
```

Uso de Destructores

```
Cadena::cadena(const cadena &cad)  // Constructor Copia MAL!!
{
   cad_ = cad.cad_;
}
Cadena::cadena(const cadena &cad) // Constructor Copia BIEN!! (Defecto)
{
   cad_ = new char[strlen(cad.cad_)+1];
   strcpy(cad_, cad.cad_);
}
```

```
char* Cadena::leer(char* c)
{
   strcpy(c, cad_); // Devolvemos una copia no el puntero
                    // para que no sea accesible desde fuera.
   return c;
}
```

7.2 Constructores y Referencias

Repasando Referencias

1. Cuando una referencia se crea, se ha de inicializar. (Los punteros pueden inicializarse en cualquier momento.)

2. Una vez una referencia se inicializa ligándola a un objeto, no se puede ligar a otro objeto. (Los punteros se pueden apuntar a otro objeto en cualquier momento.)

3. No se pueden tener referencias con valor nulo. Siempre ha de suponer que una referencia está conectada a una trozo de memoria ya asignada.

```
int y;
int& r = y;          // La asignamos a algo que ya existe
const int& q = 12; // También se puede hacer esto.
int x = 0;
int& a = x;          // x y a son lo mismo
int main()
{
   cout << "x = " << x << ", a = " << a << endl;
   a++;
   cout << "x = " << x << ", a = " << a << endl;
}
```

7.2 Constructores y Referencias

Referencias en las funciones

Cualquier cambio realizado en lareferencia dentro de la función se realizará también en el argumento fuera de la función.

```
// -------------------------
int* f(int* x)
{
  (*x)++;
  // Ok, x viene de fuera
  return x;
}
// -------------------------
int& g(int& x)
{
  // igual que f()
  x++;
  // Ok, x vienen de fuera
  return x;
}
```

```
int& h()
{
  int q;
//! return q; // Devuelve algo que se
              // saldrá de ámbito
  static int x;
  return x;   // Ok, sigue vivo después
}
int main()
{
  int a = 0;
  f(&a); // Ok, pero feo ¿no?
  g(a);  // Con referencias es más limpio
}
```

Referencias Constantes

Si ponemos un argumento como constante en una función:

- Para tipos predefiniddos no modificará el argumento.
- Para tipos definidos por el usuario:
  - Llamará sólo a métodos constantes.
  - No modificará ningún atributo público.

La referencias constantes se usan mucho porque uann función puedee recibir un objeto temporal, que puede haber sido creado como valor de retorno de otra función. Los Objetos temporales son siempre constantes.

```
void f(int&) {}
void g(const int&) {}
int main()
{
//! f(1); // Error: La memoria debe ser constante.
          //No tiene sentido cambiarlo.
  g(1);
}
```

7.2 Constructores y Referencias

Referencias a Puntero

A veces queremos modificar el puntero, no a lo que apunta.

void f(int** p);

```
#include <iostream>
using namespace std;
void increment(int*& i)
{
    i++; // ¿Qué estamos incrementando?
}
int main()
{
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
}
```

Paso de Argumentos Optimizado

Lo ideal sería pasarlo como una referencia constante (si no lo vamos a modificar).

Se requiere un constructor copia para pasar un objeto por valor, y esto no siempre es posible.

Pasar un argumento por valor necesita una llamada a un constructor y otra a un destructor, pero si no se va a modificar el argumento, el hecho de pasarlo como una referencia constante sólo necesita poner una dirección en la pila.

De hecho, prácticamente la unica situación en la que no es preferible pasar la dirección, es cuando se va a producir en el objeto tal daño que la única forma segura de que no ocurra es pasándolo por valor.

Más sobre el constructor Copia

A veces surgen problemas debido a que el compilador hace una suposición sobre cómo crear un objeto nuevo.

Cuando se pasa un objeto por valor, se crea un nuevo objeto, que estará dentro del ámbito de la función, del objeto original ya existente fuera del ámbito de la función.

```
Elemento e2 = f(h);
```

e2 es un objeto que no estaba creado anteriormente y se crea a partir del valor que retorna f(), y otra vez un nuevo objeto se crea de otro ya existente.

El compilador supone que la creacion ha de hacerse con una copia bit a bit

Cuando la clase contiene punteros pues,

¿a qué deben apuntar?

¿debería copiar solo los punteros o debería asignar memoria y que apuntaran a ella?

Más sobre el constructor Copia

Para evitar que el compilador haga una copia bit a bit, construimos nuestro función de copia.

Como estamos creando un objeto esta función tiene que ser un cosntructor, y el único argumento tiene que ver con el objeto del que partimos para crear el nuevo.

```
X(X&)
```

Si creamos un constructor copia el compilador no realizará una copia bit a bit cuando cree un nuevo objeto de otro existente.

Siempre se llamará al constructor copia que hemos creado.

Cuantos.cpp

```
class Cuantos
{
   string nombre_; // Identificador
   static int contador_;
public:
  Cuantos(const string& id = "") :
     nombre_(id)
  {
     ++contador_;
     print("Cuantos()");
  }
  ~Cuantos()
  {
     --contador_;
     print("~Cuantos()");
  }
```

```
 // Nuestro constructor Copia
  Cuantos(const Cuantos& h) :
    nombre_(h.nombre_)
  {
     nombre_ += " copiado";
     ++contador_;
     print("Cuantos(const Cuan&)");
  }
  void print(const string& msg = "")
    const
  {
     if(msg.size() != 0)
       out << msg << endl;
     out << '\t' << nombre_ << ": "
         << "contador_ = "
         << contador_ << endl;
   }
};
```

7.2 Constructores y Referencias

```
int Cuantos::contador_ = 0;
// Retorno y Argumento pasado por valor
Cuantos f(Cuantos x)
{
   x.print("Arguento x dentro de f()");
   out << "Saliendo de f()" << endl;
   return x;
}
// ----------------------------------------------------------------------
int main()
{
   Cuantos h("h");
   out << "Antes de f()" << endl;
   Cuantos h2 = f(h);
   h2.print("h2 despues de llamada a f()");
   out << "Antes de Llamada a f() sin valor de retorno" << endl;
   f(h);
   out << "Despues de Llamada to f()" << endl;
}
```