

Décimoprimer Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

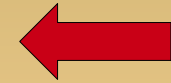
Control de Nombres

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

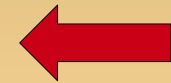
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

Creación Dinámica de Objetos

Herencia.

Tipos de Visibilidad

Herencia Múltiple

Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

11.1 Repaso de Sesiones Anteriores

const

El motivo unicial para usar *const* fue para eliminar el uso de la directiva de compilación `#define` para sustitución de valores. Tiene ventajas sobre `#define`.

```
#define NOTA_ESPERADA 8
```

```
const float nota_esperada = 8;
```

```
const int cantidad = 19;
```

```
Alumnos curso[cantidad];
```

Punteros constantes:

```
// "p es un puntero que apunta //  
a: un entero constante.
```

```
const int* p;
```

```
const int *p;
```

```
// "w es un puntero constante  
// que apunta a: un entero.
```

```
int i=3;
```

```
int* const w = &i;
```

```
*w = 2; // ¿Está bien o está mal?
```

En C++ lo mejor es utilizar una referencia constante en lugar de paso por valor.

Un método que no modifica ningún atributo se debería escribir como constante y así se podría usar desde objetos constantes.

11.1 Repaso de Sesiones Anteriores

funciones inline

Sustituye la utilización de macros que es muy común en la programación en C.

Se pensó para dotar al lenguaje C++ de mayor eficiencia.

Es una sugerencia para el compilador.

Hay dos maneras de definir un método de una clase como “inline”. Es preferible la que se utiliza en la implementación y no en la declaración de la clase.

11.2 Control de Nombres

Conceptos previos

La creación de nombres es una actividad fundamental en la programación y, cuando un proyecto empieza a tomar grandes dimensiones, el número de nombres puede fácilmente llegar a ser inmanejable.

En C++ podemos controlar:

- Creación y Visibilidad de nombres.

- Lugar dónde se almacenan.

- Enlazado de nombres.

Vamos a ver:

- “**static**” y cómo controla el almacenamiento y la visibilidad.

- “**namespace**” para controlar los nombres.

11.2 Control de Nombres

Conceptos previos

La creación de nombres es una actividad fundamental en la programación y, cuando un proyecto empieza a tomar grandes dimensiones, el número de nombres puede fácilmente llegar a ser inmanejable.

En C++ podemos controlar:

- Creación y Visibilidad de nombres.

- Lugar dónde se almacenan.

- Enlazado de nombres.

Vamos a ver:

- “**static**” y cómo controla el almacenamiento y la visibilidad.

- “**namespace**” para controlar los nombres.

11.2 Control de Nombres

static

Dos usos básicos:

1.- Se almacena la variable o el objeto una sola vez en una dirección de memoria fija. El objeto o la variable se crea en un área de datos estática especial en lugar de la pila.

2.- Variable local para un ámbito en particular. Con `static` controlamos la visibilidad de un nombre que no puede ser visto fuera del ámbito o de una clase.

11.2 Control de Nombres

Variables estáticas dentro de funciones

Si tenemos una variable local dentro de una función, cada vez que se llama a la función:

- Se reserva espacio para esa variable en la pila.

- Si hay inicializador:

 - Se realiza la inicialización.

Si utilizamos “**static**”:

- El almacenamiento no se realiza en la pila sino en el área de datos estáticos.

- La inicialización sólo se realiza la primera vez.

11.2 Control de Nombres

Variables estáticas dentro de funciones

```
char un_caracter(void)
{
    static const char*
        s="ABCDEFGHIIJK";
    if(*s == '\0')
        return 0;
    return *s++;
}

int mayor(int i)
{
    static int mayor=0;
    if(i > mayor)
    {
        mayor = i;
    }
    return mayor;
}
```

```
int main()
{
    char c;
    while((c = un_caracter()) != 0)
        cout << c << endl;

    cout <<"mayor(5)"<< mayor(5)<<endl;
    cout <<"mayor(7)" << mayor(7)<<endl;
    cout <<"mayor(5)"<< mayor(5)<<endl;
    cout <<"mayor(9)"<< mayor(5)<<endl;
}
```

11.2 Control de Nombres

Objetos estáticos dentro de funciones

Hemos visto lo que ocurre con variables (en realidad objetos de alguno de los tipos o clases básicas del lenguaje)

Las reglas para objetos estáticos:

Son las mismas.

El objeto requiere ser inicializado (a través de sus constructores).

Si no especificamos argumentos en los constructores cuando definimos un objeto estático, la clase deberá tener un constructor por defecto.

```
void funcion(void)
{
    static Alumno a("pepe",5);
    static Alumno b; // Constructor por defecto!!
    .....
}
```

y ¿cuándo se llama a los destructores?

11.2 Control de Nombres

Objetos estáticos: Destrucción

```
class Obj
{
    char c; // Identificador
public:
    Obj(char cc) : c(cc) { cout << "Constructor de " << c << endl; }
    ~Obj() { cout << "Destructor de " << c << endl; }
};
Obj a('a'); // Global
void f()
{
    static Obj b('b');
}
int main()
{
    f();
    cout << "leaving main()" << endl;
}
```

```
Constructor de a
Constructor de b
Saliendo de main()
Destructor de b
Destructor de a
```

11.2 Control de Nombres

Espacios de Nombres

Los nombres pueden estar anidados dentro de clases.

Son diferentes los métodos imprimir() de las clases:

Alumno Coche Coordenada Perro

Pero los nombres de:

Funciones globales.

Variables globales.

Clases.

No se pueden repetir....

Están dentro de que llamamos el espacio de nombres global.

11.2 Control de Nombres

Espacios de Nombres

En un proyecto grande la falta de control sobre quién ha utilizado ya los nombres en el espacio de nombres global puede traernos problemas.

Una solución común:

Nombres de funciones muy largos intentando evitar repeticiones.

```
_METODOS_2_2007_funcion_imprimir_alumnos(Lista* p);
```

Pero no es muy elegante.

Utilizaremos “**namespace**” para dividir el espacio de nombres globales en partes más manejables.

Con “**namespace**” creamos un nuevo espacio de nombres.

11.2 Control de Nombres

Crear un espacio de nombres

Muy similar a una clase:

Una definición namespace sólo puede aparecer en un rango global de visibilidad o dentro de otro “**namespace**”.

Podemos utilizar otro nombre que nos venga mejor a un espacio cuyo nombre ha puesto alguien de forma enrevesada (acordaos del ejemplo de antes).

```
namespace mi_libreria
{
    // Declaraciones
}
```

```
namespace _METODOS_II_2007_NOMBRES_
{
    class Alumno { /* ... */ };
    class AlumnoM2 { /* ... */ };
    // ...
}
// Demasiado lago. Quiero ahorrar escritura...
namespace M2 = _METODOS_II_2007_NOMBRES_ ;
int main() { M2.Alumno a; }
```

11.2 Control de Nombres

Usando los espacios de nombres

```
namespace X
```

```
{  
    class Y  
    {  
        static int i;  
    public:  
        void f();  
    };  
    class Z;  
    void func();  
}  
int X::Y::i = 9;  
class X::Z  
{  
    int u, v, w;  
public:  
    Z(int i);  
    int g();  
};
```

```
X::Z::Z(int i)  
{  
    u = v = w = i;  
}  
int X::Z::g()  
{  
    return u = v = w = 0;  
}  
void X::func()  
{  
    X::Z a(1);  
    a.g();  
}  
void funcion()  
{  
    using namespace X;  
    Z b(1);  
}
```

11.2 Control de Nombres

Espacios de Nombres

Con la directiva `using` es como si estuviéramos dentro de la definición del espacio de nombres.

Con `using` no nos hace falta utilizar en el ámbito donde nos situemos el nombre del espacio de nombres al que nos referimos.

```
#ifndef USINGDECLARATION_H
#define USINGDECLARATION_H
namespace U {
    inline void f() {}
    inline void g() {}
}
namespace V {
    inline void f() {}
    inline void g() {}
}
#endif // USINGDECLARATION_H
```

```
#include "UsingDeclaration.h"
namespace Q {
    using U::f;
    using V::g;
    // ...
}
void m() {
    using namespace Q;
    f(); // Llamada a U::f();
    g(); // Llamada a V::g();
}
int main() {}
```

11.2 Control de Nombres

Espacios de Nombres

```
#include <iostream>
namespace uno {
int x;
}
namespace dos {
int x;
}
using namespace uno;
int main() {
    x = 10;
    dos::x = 30;
    std::cout << x << ", " << dos::x << std::endl;
    std::cin.get();
    return 0;
}
```

11.2 Control de Nombres

Espacios de Nombres anónimos

```
namespace Nombre
{
    int f();
    char s;
    void g(int);
}
namesmace
{
    int x = 10;
}
// x sólo se puede desde este punto hasta el final del fichero
// Resulta inaccesible desde cualquier otro punto o fichero
namespace Nombre {
    int f() {
        return x;
    }
}
```

11.3 Resumen

Static

Podemos controlar el almacenamiento de nuestros objetos y variables.

Podemos controlar el ámbito en el que se ven.

Espacios de Nombres

Utilizamos la palabra reservada ***“namespace”***.

Nos ayudan a dividir el espacio de nombres global de forma que podamos repetir nombres de funciones, clases y variables.

Es una declaración y se suele utilizar por tanto en los ficheros de cabecera.