

Tercera Sesión

Metodologías y Técnicas de Programación II Programación Orientada a Objeto (POO)

C++

Profesor:

José Luis Marina
jlmarina@nebrija.es

Laboratorios:

José Luis Marina
jlmarina@nebrija.es
Borland Builder 6.0

Puntuación:

Prácticas Laboratorio:	20%
Diarias	20%
Trabajo Laboratorio	80%

Exámen Parcial	15%
----------------	------------

Exámen Final	65%
--------------	------------

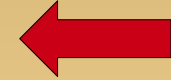
Exámen Final Extraordinario	70%
------------------------------------	------------

3.0 Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

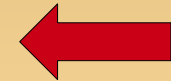
C++
Mi primera Clase



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

3.1 Repaso de clases anteriores.

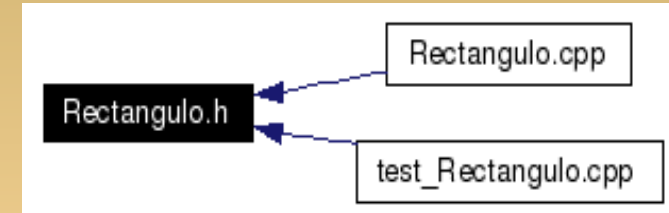
Estándares

Nombres de Ficheros:

Rectángulo.h (#ifndef NOMBRE_CLASE_H ...)

Rectangulo.cpp

test_Rectangulo.cpp (main)



Documentación de Cabeceras de Ficheros.

```
/**
 * @file test_Rectangulo.cpp
 * @brief Pruebas sobre la clase Rectangulo. Ejemplo Metodos II
 *
 * @author Jose Luis Marina <jlmarina@nebrija.es>
 * @date 19-FEB-2007
 */
```

Documentación Cabeceras de Clases.

Nombres de Clases, Objetos, atributos y métodos – Indexado y corchetes

Ayudan a: Reutilizar Código a Documentar (Código más comprensible)

3.1 Repaso de clases anteriores.

Programación Estructurada

Funciones

Encapsulan la complejidad

Declaración (.h) :

```
int translate(float x, float y, float z);  ó void f(void)
```

Implementación:

```
int translate(float x, float y, float z)    // Correcto
{
    x = y = z;
    return 33;
}
```

Control del Flujo

```
if (expresion)          while(guess != secret)          for(i = 0; i<128;i++)
{
    sentencias          {
                        cout << "Numero: ";
    }                  cin >> apuesta;
else                  }
{
    sentencias          //do { } while ()
}

                        swicth (c)
                        {
                            case 'a': 4..
```

3.1 Repaso de clases anteriores.

Programación Estructurada

Operadores

Matemáticos	(+ , - , * , / , %)	Relacionales	(> , >= , < , <= , == , !=)
Lógicos	(&& ,)	Bits	(& , , ^ , >> , <<)
Unarios	(- , + , ++ , --)	Direcciones	(& , * , ->)
Ternario	a = --b ? b : (b = -99);		

Casting o Moldeado

Hay que utilizarlo con mucho cuidado y en pocas ocasiones

```
i = 1; // Posible perdida de digitos
i = f; // Posible perdida info,
i = static_cast<int>(1); // no warning
i = static_cast<int>(f); // no warning
char c = static_cast<char>(i);
```

Operador sizeof

Número de bytes utilizado por un tipo de datos o por una variable.

3.1 Repaso de clases anteriores.

Programación Estructurada

Tipos de Datos

Básicos:

Predefinidos : char, int, float, double)

Se combinan con : long y short signed y unsigned.

Sus límites están en ***limits.h*** y ***float.h***

```
# define SHRT_MIN      (-32768)
# define SHRT_MAX      32767
```

Algunas combinaciones no valen (long float -- unsigned float)

Creados por el usuario (programador)

El compilador aprende a usarlos.

Utiliza las declaraciones (por ejemplo de la clases)

3.4 Programación Estructurada II

Punteros

Cuando se ejecuta un programa, primero se carga en memoria.

Se cargan cada uno de sus elementos:

main() funciones variables objetos

Operador & : Nos permite saber la dirección de memoria de un elemento.

```
int dog, cat;
void f(int pet)
{
    cout << "pet id number: " << pet << endl;
}
int main()
{
    int i, j;
    cout << "f(): " << (long)&f << endl;
    cout << "dog: " << (long)&dog << endl;
    cout << "i: " << (long)&i << endl;
    cout << "j: " << (long)&j << endl;
}
```

f(): 4198736
dog: 4323632
i: 6684160
j: 6684156

3.4 Programación Estructurada

Punteros

Las direcciones de memoria se pueden guardar dentro de otras variables para su uso posterior: Los Punteros.

Un puntero se define con el operador “*” y el tipo de variable a la que apunta.

```
int i;  
int *pi;
```

```
pi = &i; // Puntero a la dirección de i.
```

```
i = 33; // Valor de i.
```

```
*pi= 55; // Contenido de lo que apunta pi
```

```
// Atentos con:
```

```
int a, b, c; // Ok
```

```
int* pa, pb, pc ; // No OK. Solo pa es puntero.
```

```
int* p1;
```

```
int* p2; // Ok.
```

6684160

33

i

DIR pi

6684160

pi

3.4 Programación Estructurada

Punteros: Modificando argumentos.

En el **paso-por-valor** se realiza una copia del argumento.

```
#include <iostream>
using namespace std;
void f(int a)
{
    cout << "a = " << a << endl;
    a = 5;
    cout << "a = " << a << endl;
}
int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(x);
    cout << "x = " << x << endl;
}
```

```
#include <iostream>
using namespace std;
void f(int* a)
{
    cout << "a = " << a << endl;
    cout << "*a = " << *a << endl;
    *a = 5;
}
int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(&x);
    cout << "x = " << x << endl;
}
```

En el segundo caso el valor de **“a”** es el valor de la dirección de **“x”**.
Un puntero es casi otro nombre de la misma variable.

3.4 Programación Estructurada

Referencias de C++.

Es un modo adicional de pasar una dirección a una función.

Es más limpio sintácticamente que con punteros.

```
#include <iostream>
using namespace std;
void f(int& a)
{
    cout << "a = " << a << endl;
    cout << "&a = " << &a << endl;
    a = 5;
}
int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(x); // Parece paso-por-valor
    cout << "x = " << x << endl;
}
```

```
#include <iostream>
using namespace std;
void f(int* a)
{
    cout << "a = " << a << endl;
    cout << "*a = " << *a << endl;
    *a = 5;
}
int main() {
    int x = 47;
    cout << "x = " << x << endl;
    f(&x);
    cout << "x = " << x << endl;
}
```

Conseguimos lo mismo que con los punteros.

3.4 Programación Estructurada

Punteros y Referencias como Modificadores de Tipos de Dato

Todas las combinaciones de tipos de dato básicos, especificadores, punteros y referencias:

```
void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
         unsigned short int usi, unsigned long int uli);
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
         long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
         unsigned short int* usip,
         unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
         long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
         unsigned short int& usir,
         unsigned long int& ulir);
```

3.4 Programación Estructurada

Punteros y el tipo **“void”**

```
void* p;
```

Puedo guardar cualquier tipo de dirección en ese puntero.

Al asignar a void se pierde la información del tipo.

```
void* vp;  
char c;  
int i;  
float f;  
double d;  
// Asigno a cualquier tipo  
vp = &c;  
vp = &i;  
vp = &f;  
vp = &d;
```

```
.....
```

```
int i = 99;  
void* vp = &i;  
*vp = 3; // Error al compilar  
// Lo moldeamos a int antes.  
*((int*)vp) = 3;
```

```
.....
```

En general deberíamos evitar los punteros a void.

3.4 Programación Estructurada

Ámbito o alcance (Scope)

El ámbito nos dice cuándo es válida una variable o un objeto.

Se extiende desde el punto en que es definida hasta la primera llave que se empareja con la llave de apertura anterior a que la variable fuera definida.

Lo define su juego de llaves más cercanas.

```
int main()
{
    int x;
    {
        int y;
        {
            int z;
            // Válidas: x, y, z
        } // Adiós a z.
        // Válidas: XXXXXXXXX
    }
    // Válidas: XXXXXXXXXXXXXXXXXXXX
}
```

3.4 Programación Estructurada

Definición de Variables “al vuelo”

El lenguaje C y otros lenguajes procedimentales obligan a declarar todas las variables al principio de “un bloque”.

C++ permite definir variables en cualquier sitio dentro de un ámbito

Se puede definir una variable justo antes de usarla e inicializarla

```
{
  int q = 0;
  for(int i = 0; i < 100; i++)
  {
    q++; // q viene de ámbito mayor
    int p = 12; // ¡Cuánto dura p?
  }
  int p = 1; // Es una “p” diferente
}
cout << "Intro. caracter:" << endl;
while(char c = cin.get() != 'q')
{
  cout << c << "No es 'q' " << endl;
  if(char x = c == 'a' || c == 'b')
```

```
    cout << "a o b" << endl;
}
cout << "Type A, B, or C" << endl;
switch(int i = cin.get())
{
  case 'A':cout << "A" <<endl;break;
  case 'B':cout << "B" <<endl;break;
  case 'C':cout << "C" <<endl;break;
  default:cout << "Mal!" << endl;
}
```

3.4 Programación Estructurada

Memoria y Espacio de Almacenamiento

Variables Globales

Se definen fuera de la llaves de las funciones incluida la función main(). Están siempre disponibles.

extern: Si están en un fichero que no es en el que trabajamos ahora

```
// fichero_uno.cpp
#include <iostream.h>
int global;
void func();
int main()
{
    globe = 12;
    cout << globe << endl;
    func(); // Modifies globe
    cout << globe << endl;
}
```

```
// fichero_dos.cpp
//Accedemos a variable global externa
extern int global;
// Al enlazar se resuelve la
// referencia

void func()
{
    global = 47;
}
```

3.4 Programación Estructurada

Memoria y Espacio de Almacenamiento

Variables Locales

Son locales a una función (o bloque {.....}) y se crean en el área de almacenamiento automático.

Variables de Registro.

Es una variable local que se almacena en uno de los registros de la CPU. Utilizamos la palabra **register** para sugerirle al compilador que coloque nuestra variable en un registro y así el acceso sea más rápido.

Variables de Estáticas

Cada vez que entramos en una función se genera el espacio de memoria para las variables y se reinician. Con **static** se le da a una variable una dirección fija constante (Como a una global pero ¿en qué se diferencian?)

```
void func()
{
    static int i = 0;
    cout << "i = " << ++i << endl;
}
```

```
int main()
{
    for(int x = 0; x < 10; x++)
        func();
}
// Si quitamos static ¿Que pasa?
```

3.4 Programación Estructurada

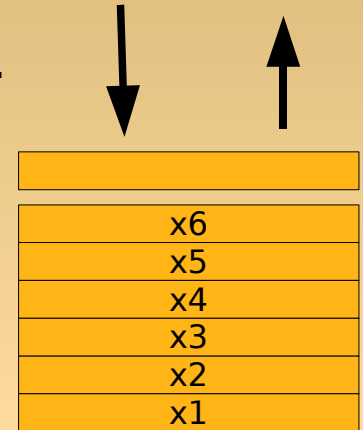
Memoria y Espacio de Almacenamiento

Identificador

Espacio de memoria que aloja variable o función compilada.

Pila

Área de almacenamiento para variables automáticas.
Variables locales a las funciones.



Memoria Global

Para variables globales y cuando hacemos **new()**.

3.4 Programación Estructurada

Valores Constantes

Por defecto se asume que están en decimal. 10 es “diez” y no “dos”

0 al principio	Octal	045
0x al principio	Hexadecimal	0x1fe

Punto flotante

Se convierte de forma implícita, pero podemos ayudarnos utilizando el punto decimal.

Tipo char

Entre comillas simples ***char c = 'p';***

Caracteres especiales:

Usamos la “barra invertida”: '\n' (nueva línea), '\t' (tabulación), '\\' (barra invertida), '\"' (comillas dobles), '\'' (comilla simple), etc.

3.4 Programación Estructurada

Registros o Estructuras

Nos permiten agrupar un conjunto de variables en una estructura común.

```
struct Structure1
{
    char c;
    int i;
    float f;
    double d;
};
int main()
{
    struct Structure1 s1, s2;
    s1.c = 'a'; //
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
    s2.d = 0.00093;
}
```

```
typedef struct
{
    char c;
    int i;
    float f;
    double d;
} Structure2 ;
int main()
{
    Structure2 s1, s2;
    s1.c = 'a'; //
    s1.i = 1;
    s1.f = 3.14;
    s1.d = 0.00093;
    s2.d = 0.00093;
}
```

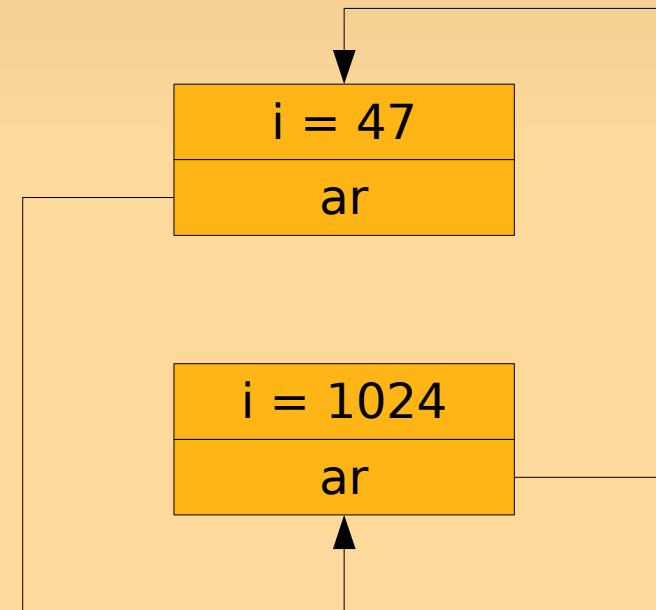
3.4 Programación Estructurada

Registros o Estructuras

Las estructuras agrupan información.

Las clases agrupan información (atributos) y comportamiento (métodos).

```
typedef struct AutoReferencia
{
    int i;
    AutoReferencia* ar;
} AutoReferencia;
int main()
{
    AutoReferencia ar1, ar2;
    ar1.ar = &sr2;
    ar2.ar = &sr1;
    ar1.i = 47;
    ar2.i = 1024
    cout << ar1.ar->i; // ¿Qué sale?
}
```



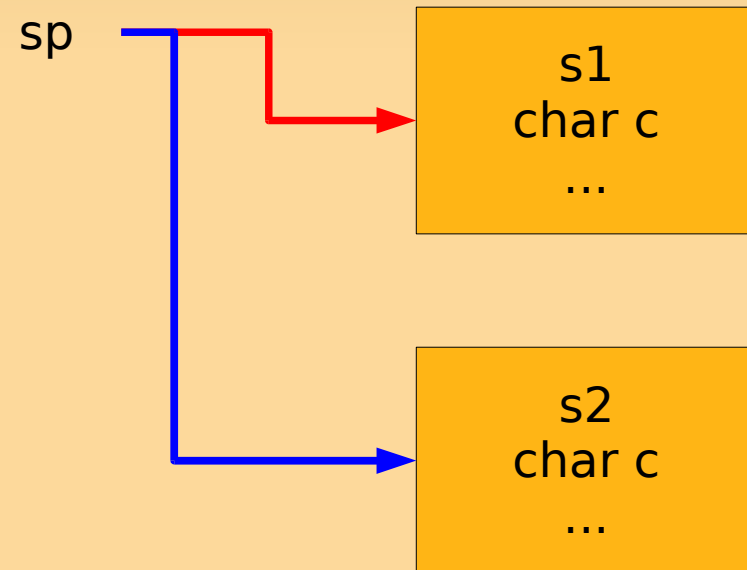
3.4 Programación Estructurada

Estructuras y Punteros

Para seleccionar un elemento de una estructura utilizamos '.'

Si utilizamos un puntero: '->'

```
typedef struct Structure3
{
    char c;
    int i;
    float f;
} Structure3;
int main()
{
    Structure3 s1, s2;
    Structure3* sp = &s1;
    sp->c = 'a'; // Igual que s1.c
    sp->i = 1;
    sp->f = 3.14;
    sp = &s2; // Cambiamos
    sp->c = 'a';
}
```



3.4 Programación Estructurada

Arrays o Vectores

Permiten agrupar variables una a continuación de la otra. Todas del mismo tipo.

```
int main()
{
    int a[10];
    for(int i = 0; i < 10; i++)
    {
        a[i] = i * 10;
        cout<<"a["<<i<<" ]="<<a[i]<< endl;
    }
}
```

```
typedef struct
{
    int i, j, k;
} ThreeDpoint;

ThreeDpoint p[10];
for(int i = 0; i < 10; i++)
{
    p[i].i = i + 1; // Dos 'i' dif.
    p[i].j = i + 2;
    p[i].k = i + 3;
}
```

Accesos muy rápidos.

Si indexamos más allá de los límites: Problemas.

No se comprueban los límites en compilación.

Hay que definir el tamaño en tiempo de compilación.

3.4 Programación Estructurada

Punteros y Arrays

El nombre de un array sin los corchetes ([x]) nos da la dirección del 1er elemento. No podemos asignarle algo al nombre del array. `a[10] a=33;`

```
int main()
{
    int a[10];
    cout << "a = " <<a<< endl;
    cout << "&a[0] ="<<&a[0] << endl;
}
```

```
int main(int argc, char* argv[])
{
    cout << "argc = " << argc << endl;
    for(int i = 0; i < argc; i++)
        cout << "argv[" << i << "] = "
            << argv[i] << endl;
}
```

```
void func2(int* a, int size)
{
    for(int i = 0; i < size; i++)
        a[i] = i * i + i;
}

int array1[5], array2[9];
func2(array1,5);
func2(array2,9);
```

3.4 Programación Estructurada

Aritmética de Punteros

Lo más común es incrementar un puntero en "n" o en "1" (++).

No tiene mucho sentido sumar dos punteros. **¿Y restarlos?**

```
int main()
{
    int i[10];
    double d[10];
    int* ip = i;
    double* dp = d;
    cout << "ip = "
    ip++;
    cout << "ip = "
    cout << "dp = "
    dp++;
    cout << "dp = "
}
```

ip = 6684124

ip = 6684128

dp = 6684044

dp = 6684052

3.5 Resumen

Programación Estructurada

Estándares

Reutilizando

Funciones

argumentos

Control de Flujo

if,while,for

Ámbito de Validez

Globales, locales
static

Operadores

+,--,&&, >>

Casting

i = (int)i;

Tipos de Datos

char, int, short int

Punteros

int* , void*

Constantes

const int i=0;

Almacenamiento

Heap, Global

Estructuras

struct

Arrays

int a[10];