Tutorial de C++

o el diario de Peter Class

Peter Class

Pello Xabier Altadill Izura

Tutorial de C++: o el diario de Peter Class
por Peter Class y Pello Xabier Altadill Izura

Historial de revisiones
Revisión 1.0 19-11-2004 Revisado por: px
Documento inicial
Revisión 1.1 23-11-2004 Revisado por: px
Revision inicial, correcciones menores e imagenes

Tabla de contenidos

1. Intro 1
2. Hola nena 3
3. Funciones 5
4. Tipos de datos 9
5. Operadores 13
6. Parametros, ambito, sobrecarga 19
7. Clases 25
8. Iteraciones 31
9. Punteros 37
10. Referencias 43
11. Funciones avanzadas 51
12. Arrays 59
13. Herencia 63
14. Herencia multiple 69
15. Miembros estaticos 77
16. Clases y sus amigas 81
17. Entrada/Salida 89
18. Preprocesador 95
19. Principios de POO 99
20. Templates 101
21. Excepciones 107
22. Librerias estandar 113
23. Notas, autoria, licencia, referencias 117

Capítulo 1. Intro

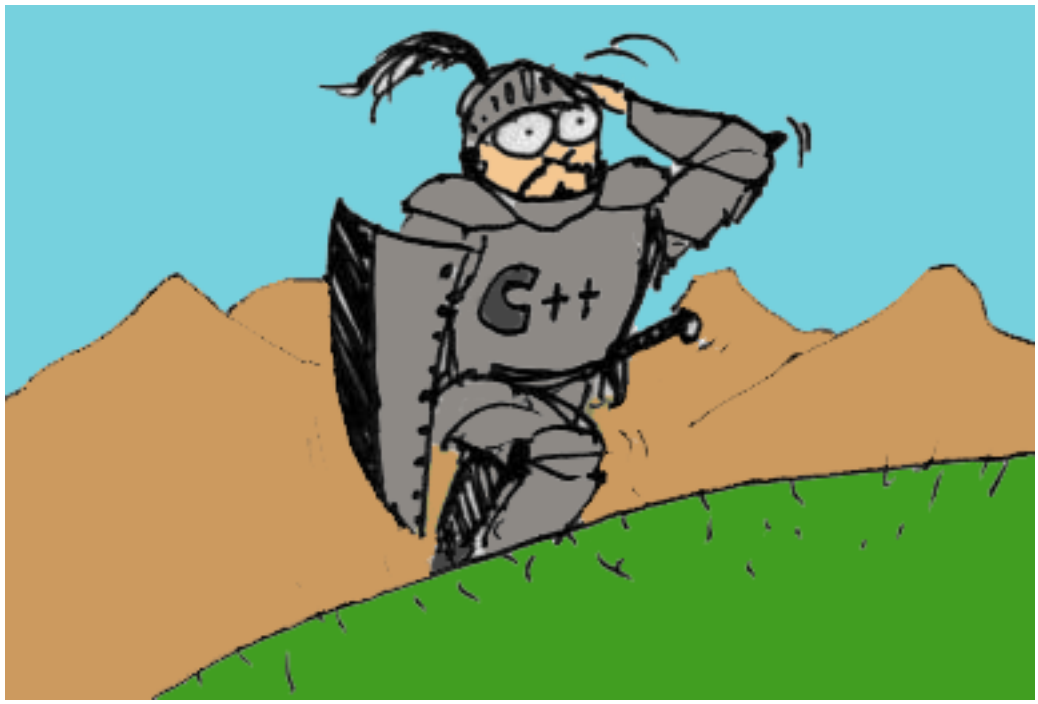


Figura: el caballero de Peter Class.

Este es el diario de Peter Class sobre sus dias aprendizaje de una disciplina de caballeros: c++ sobre linux. No pretende ser una vision exaustiva del lenguaje c++, simplemente muestra su uso a traves de los ejemplos. Un lenguaje orientado a objetos como c++ precisa de cierta explicacion previa antes de meterse en desarrollos serios, y para aprenderlo que mejor que programar ejemplos.

Peter era un campesino que se empeño en convertirse en paladin de c++, para desfacer entuertos y para llevar la virtud a los lugares mas sacrilegos de la programacion. No fue facil, y sus experiencias se cuentan aqui. Este diario es por tanto un conjunto de ejemplos de codigo glosados por el ahora caballero Peter Class.

Atencion: este tutorial no contiene ni una sola linea de codigo util. Simplemente es un conjunto de ejemplos ultrasimplones que tratan de mostrar la sintaxis cd c++. Puede ser util como referencia rapida, sobre todo si da pereza mirar en los tipicos libros de c++ de 900 paginas. Si, esos mismos libros que en la pagina 200 todavia estan con las estructuras de control; pero no dire nombres <-- estoy pensando en Deitel, pero openjade ocultara esto jeje --> Doh!

Capítulo 2. Hola nena

Bueno, vamos a ver si en 21 dias se va creando codigo c++ y se puede aprender este lenguaje de forma ordenada. Se esta probando este codigo con gcc. Hoy es el dia uno e incluyo el código mas simple posible.

```
/**
* HolaNena.cpp
* Codigo iniciatico que simplemente muestra el mensaje estandar HolaNena de nano
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ HolaNena.cpp -o HolaNena
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida estandar un mensaje
 cout << "HolaNena!\n";

 return 0;
}
```

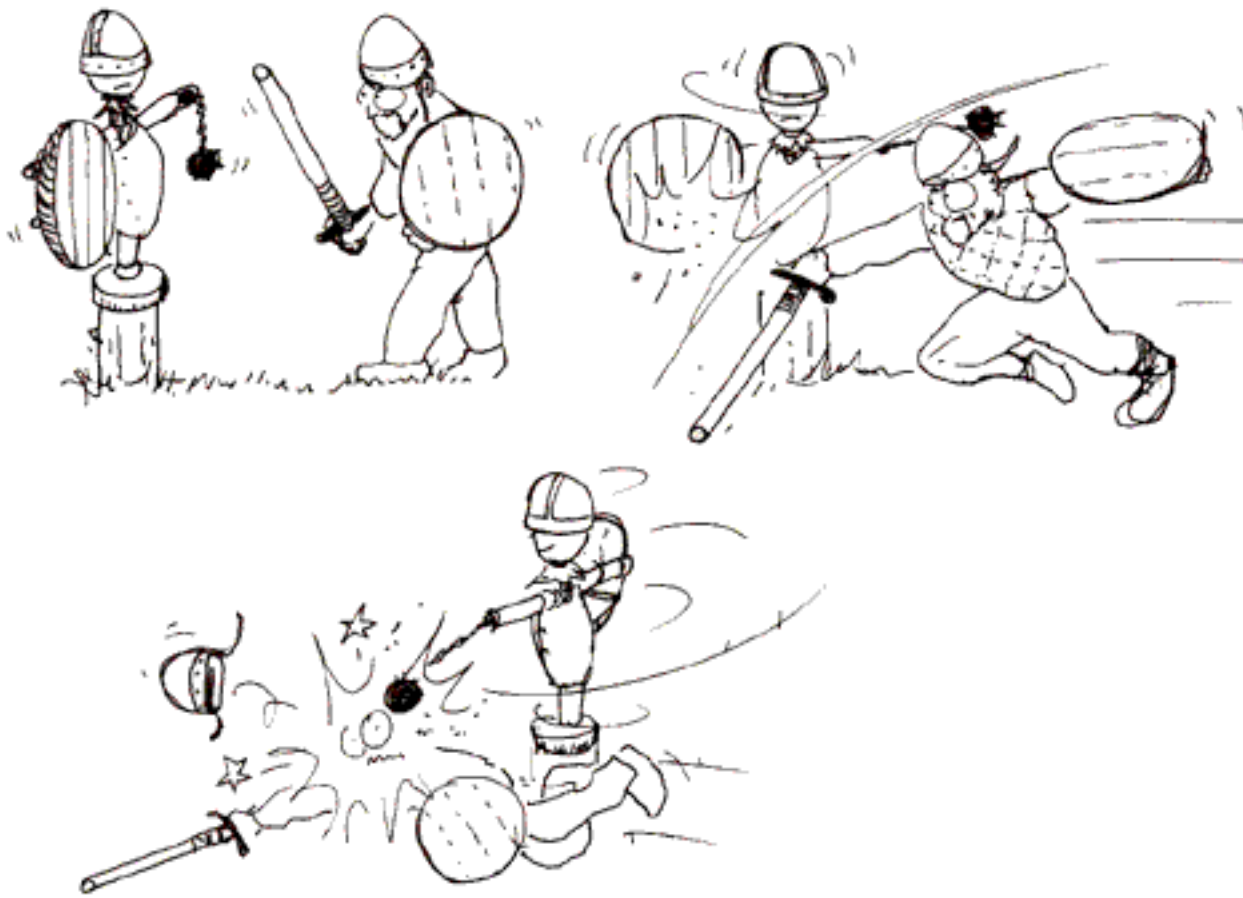


Figura: Entrenando. Hay que ir poco a poco

Aqui otro codigo de ejemplo.

```
/**
* Test.cpp
* Programa del ejemplo simple, para saber que es lo que hace ejecutalo
*
*
* Compilado: g++ Test.cpp -o Test
*/

using namespace std;
#include <iostream>

int main() {
 int x = 5;
 int y = 7;

 cout << "\n";
 cout << x + y << " " << x * y;
 cout << "\n";

 return 0;
}
```

Capítulo 3. Funciones

Vamos a ver como se declaran las funciones en c++. No tiene ningun misterio, es igual que en c. Siempre hay que especificar el tipo de retorno.

```
/**
* Funcion.cpp
* Programa con llamada a una funcion
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Funcion.cpp -o Funcion
*/

using namespace std;
#include <iostream>

// Funcion: llamada
int llamada (int x, int y) {
 cout << "Estamos en la funcion!!" << endl;
 return (x+y);
}


int main() {
 // Estos comentarios son propios de C++
 cout << "Vamos a llamar a la funcion.." << endl;

 // Llamamos a la funcion
 // Llamamos a una funcion y asignamos
 int z = llamada(5,7);
 cout << "Resultado:" << z << endl;

 // Llamada desde el output
 cout << "Resultado:" << llamada(6,7) << endl;
 cout << "Programa terminado \n" << endl;

 return 0;
}
```

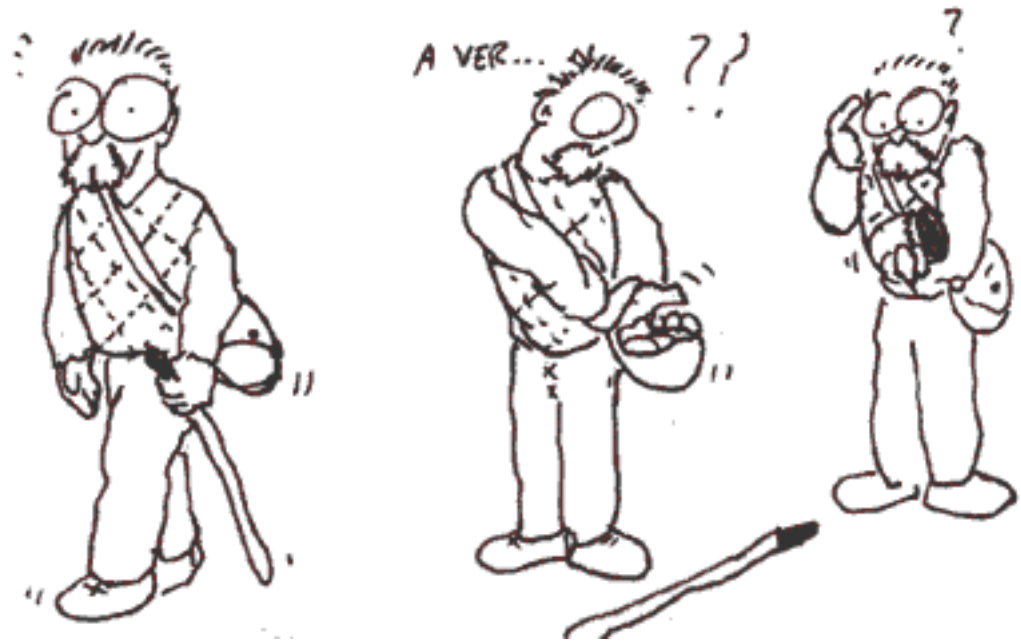


Figura: Quiza llevemos encima un tarro de esencia de Fibonaci

Atencion, en este caso veremos como recoger datos de stdin o entrada estandar.

```
/**
* Funcion3.cpp
* Programa con llamada a una funcion
* El programa principal recoge datos de la entrada estandar
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Funcion3.cpp -o Funcion3
*/

using namespace std;
#include <iostream>

// Funcion: llamada
int llamada (int x, int y) {

 cout << "Estamos en la funcion!!" << endl;
 return (x+y);
}

int main() {

 // Estos comentarios son propios de C++
 cout << "Vamos a llamar a la funcion.." << endl;

 // Llamamos a la funcion
 // Llamamos a una funcion y asignamos
 int z = 0, x = 0, y = 0;

 // Recogemos los parametros
 cout << "Dame el primer parametro:";
 cin >> x;
 cout << "\nOK!\nDame el segundo parametro:";
 cin >> y;
 cout << "\nOK vamos a calcular.";
```

```
 // Efectuamos la funcion.
 z = llamada(x,y);

 // Mostramos el resultado
 cout << "Resultado:" << z << endl;

 // Llamada desde el output
 cout << "Resultado:" << llamada(6,7) << endl;
 cout << "Programa terminado \n" << endl;

 return 0;
}
```

Facil no?

Capítulo 4. Tipos de datos

Figura: al principio puede hacerse un poco complejo

Los tipos de datos de c++ no varian mucho respecto a c y son bastante evidentes, tal y como se puede apreciar en este codigo.

```
/**
* Tipos.cpp
* Programa para sacar el tamaÃ±o de cada tipo de datos
* Pello Xabier Altadill Izura
*
* Compilado: g++ Tipos.cpp -o Tipos
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos el tamaÃ±o de cada tipo
 cout << "El tamaÃ±o del int es:\t\t" << sizeof(int) << " bytes.\n";
 cout << "El tamaÃ±o del short es:\t" << sizeof(short) << " bytes.\n";
 cout << "El tamaÃ±o del long es:\t" << sizeof(long) << " bytes.\n";
 cout << "El tamaÃ±o del char es:\t\t" << sizeof(char) << " bytes.\n";
 cout << "El tamaÃ±o del float es:\t\t" << sizeof(float) << " bytes.\n";
 cout << "El tamaÃ±o del double es:\t" << sizeof(double) << " bytes.\n";

 // Sacamos por salida standar un mensaje
 cout << "Termino el programa\n";

 return 0;

}
```

Tambien se pueden definir constantes:

```
/**
* Constante.cpp
* Programa en el que definimos un valor constante
* Pello Xabier Altadill Izura
*
* Compilado: g++ Constante.cpp -o Constante
*/

using namespace std;
#include <iostream>

#define MEMOLA 25

int main () {

 int y = 0;

 // Definimos el valor constante
 const float PI = 3.1416;

 cout << "Ten fe en el caos: " << PI << endl;

 // Sacamos por salida standar un mensaje
 cout << "\nTermino el programa : " << MEMOLA << endl;

 return 0;
}
```

Tabla 4-1. Tipos basicos de datos

Tipo	Descripcion	Modificadores
void	Vacio	
char	Caracter (8 bits)	signed char(8 bits), unsigned char(8 bits)
int	Entero simple (16 bits)	signed int(16 bits), unsigned int(16 bits), long int (32 bits), unsigned long int(32 bits), signed long int(32 bits), short int(16 bits), unsigned short int(16 bits), signed short int(16 bit)
float	Coma flotante (32 bits)	
double	Coma flotante mas grande (64 bits)	long double (80 bits)
bool	Valor booleano: true o false	
wchar_t	Caracteres anchos, para determinado juegos de caracteres	

Sin animo de extenderse mucho mas, en c++ tambien disponemos de struct, union y

enum. Veamos unos ejemplos:

```
struct ficha {
 char nombre[50];
 int edad;
 char dni[9];

} ficha1, ficha2;

strcpy(ficha1.nombre,"Marujita Diaz");
ficha1.edad =
Segmentation fault - value out of range! please use double type
core dumped
```

Las union son parecidos a los structs con la gran diferencia de que sus campos comparten el mismo espacio de memoria. Podemos meter elementos de distintos tipo y la union tendra el tamaño del elemento mas grande.

```
// cuando guardemos un valor en alguna de los campos, tambien se guardara
// en los demas. Podremos tratar el mismo dato de distintas formas.
union valor {
 int numero;
 double numerazo;
 char caracter[2];
} mi_valor;
```

Y mas adelante saldra algun ejemplo de enumeracion...

Capítulo 5. Operadores

Bueno, conociendo los tipos de datos ya podemos empezar a operar con ellos. Dentro de c++ tenemos los tipicos operadores matematicos + - * / y tambien los operadores unarios (++ --) En este primer ejemplo vemos operadores unarios y la asignacion multiple.

```
/**
* Operadores.cpp
* Programa para probar algunos operadores
* Pello Xabier Altadill Izura
*
* Compilado: g++ Operadores.cpp -o Operadores
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 unsigned int a = 0, b = 0, c;

 // Sacamos el valor por pantalla de test
 cout << "Valor de test: " << test << endl;

 // Sacamos el valor por pantalla de test++
 cout << "Valor de test++: " << (test++) << endl;

 // Sacamos el valor por pantalla de ++test
 cout << "Valor de ++test: " << (++test) << endl;

 cout << "Valor de test actual: " << test << endl;

 // asignacion multiple
 c = b = a = test;

 // Veamos el resto de valores
 cout << "Y los demas: " << c << " " << b << " " << a << endl;

 return 0;

}

/**
* Operadores.cpp
* Programa para probar algunos operadores
* Pello Xabier Altadill Izura
*
* Compilado: g++ Operadores.cpp -o Operadores
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 unsigned int a = 0, b = 0, c;
```

```
 // Sacamos el valor por pantalla de test
 cout << "Valor de test: " << test << endl;

 // Sacamos el valor por pantalla de test++
 cout << "Valor de test++: " << (test++) << endl;

 // Sacamos el valor por pantalla de ++test
 cout << "Valor de ++test: " << (++test) << endl;

 cout << "Valor de test actual: " << test << endl;
 // asignacion multiple
 c = b = a = test;

 // Veamos el resto de valores
 cout << "Y los demas: " << c << " " << b << " " << a << endl;

 return 0;

}
```

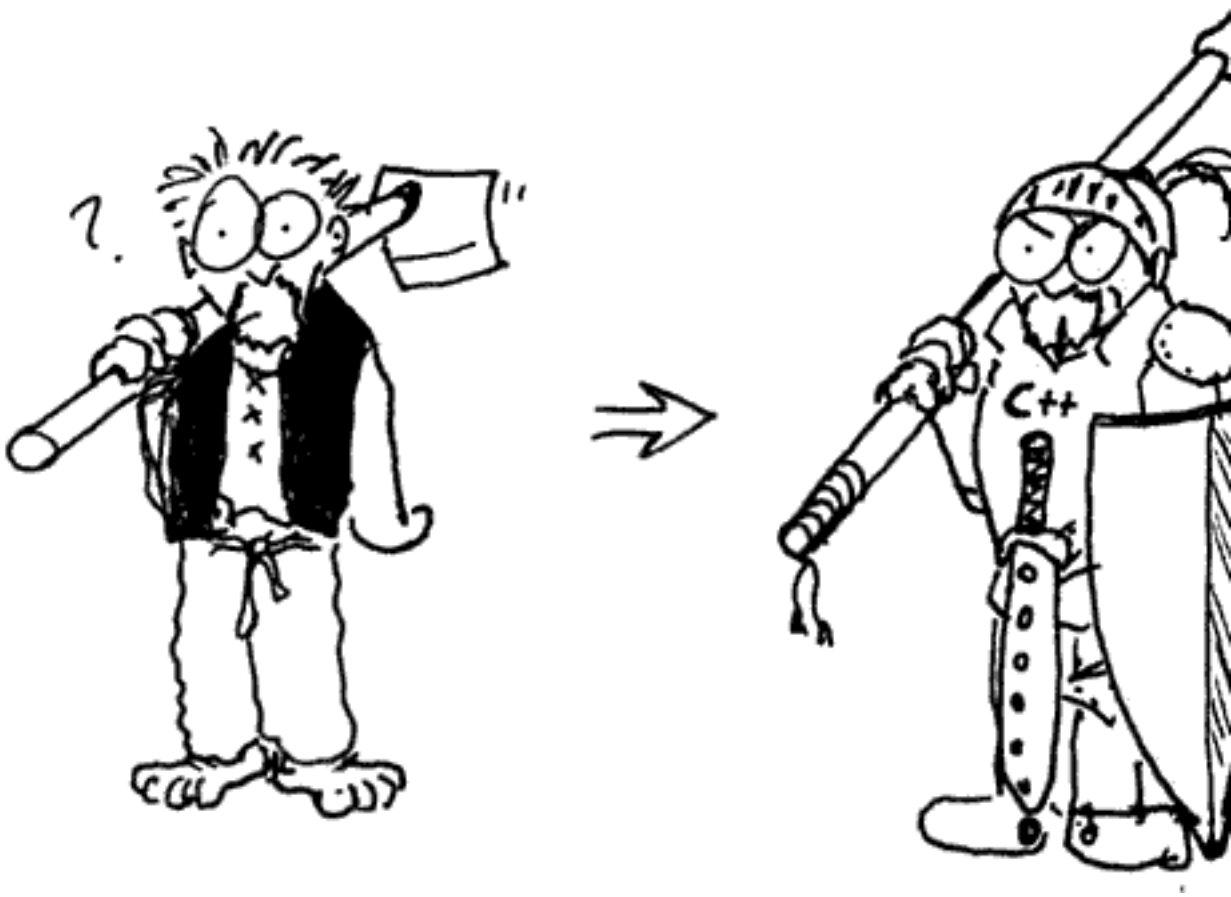


Figura: aprendiendo c++ puedes escalar socialmente. Aunque si lo que quieres es ganar dinero, quedate en el campo.

En el siguiente codigo vamos un poco mas alla y se muestra algunas operaciones abreviadas y algunas comparaciones.

```
/**
* Operadores2.cpp
* Programa para probar algunos operadores segunda parte
* Pello Xabier Altadill Izura
*
* Compilado: g++ Operadores2.cpp -o Operadores2
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 unsigned int a = 0, b = 0, c;

 // asignacion multiple
 c = b = a = ++test;

 b += 3;
 c++;
 a -= 1;

 // Veamos el resto de valores
 cout << "Son estos: c=" << c << " b=" << b << " a=" << a << endl;

 a += b + c;

 cout << "Y ahora son estos: c=" << c << " b=" << b << " a=" << a << endl;

 // Tomamos el valor a
 cout << "Dame valores. \na=";
 cin >> a;

 // Tomamos el valor b
 cout << "b=";
 cin >> b;

 // Tomamos el valor c
 cout << "c=";
 cin >> c;

 cout << "Y ahora son estos: c=" << c << " b=" << b << " a=" << a << endl;

 //Probamos el if
 if (a > b) {
  cout << "A es mayor que B" << endl;
 }

 //Probamos el if
 if (a >= b) {
  cout << "A es mayor o igual que B" << endl;
 }

 //Probamos el if
 if (a <= b) {
  cout << "A es menor o igual que B" << endl;
 }

 return 0;
```

```
}
```

Operadores logicos. A continuacion vemos algunos ejemplos de operadores logicos (comparaciones) y la combinacion de estos.

```
/**
* Logicos.cpp
* Programa para probar operadores Logicos
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Logicos.cpp -o Logicos
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 int a = 23, b = 21, c = 34;

 // Veamos el resto de valores
 cout << "Valores: " << c << " " << b << " " << a << endl;

 // Tomamos el valor a
 cout << "Dame valores. \na=";
 cin >> a;

 // Tomamos el valor b
 cout << "b=";
 cin >> b;

 // Tomamos el valor c
 cout << "c=";
 cin >> c;

 cout << "Y ahora son estos: c=" << c << " b=" << b << " a=" << a << endl;

 // Veamos una sentencia if-else sencilla
 if (!(a == b))
  cout << "a y b no son iguales" << endl;
 else
  cout << "a y b son iguales" << endl;

 // Veamos otra sentencia if-else sencilla
 if ((a == b) || (b == c))
  cout << "A y B son iguales o B y C son iguales" << endl;
 else
  cout << "ni A y B son iguales ni B y C son iguales" << endl;

 // Nota. Ley de De Morgan
 // !(A && B) == !A || !B
 // !(A || B) == !A && !B

 return 0;

}
```

Mas operadores logicos. Tambien introducimos el operador ()?: que simplifica las expresiones pero las hace un poco ilegibles.

```
/**
* Logicos2.cpp
* Programa para probar operadores Logicos 2
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Logicos2.cpp -o Logicos2
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 int a = 23, b = 21, c = 34;

 // Veamos el resto de valores
 cout << "Valores: " << c << " " << b << " " << a << endl;

 // Tomamos el valor a
 cout << "Dame valores. \na=";
 cin >> a;

 // Tomamos el valor b
 cout << "b=";
 cin >> b;

 // Tomamos el valor c
 cout << "c=";
 cin >> c;

 cout << "Y ahora son estos: c=" << c << " b=" << b << " a=" << a << endl;

 // Veamos una sentencia if-else sencilla
 if (!a)
  cout << "A es false (igual 0)" << endl;
 else
  cout << "A es true (distinto de 0)" << endl;

 // Veamos una sentencia if-else sencilla
 if (!b)
  cout << "B es false (igual 0)" << endl;
 else
  cout << "B es true (distinto de 0)" << endl;

 // Veamos una sentencia if-else sencilla
 if (!c)
  cout << "C es false (igual 0)" << endl;
 else
  cout << "C es true (distinto de 0)" << endl;

 // Sentencia con operador logico o TERNARIO ()?:
 c = (a == b)?0:1;

 cout << "C es : " << c << endl;

 return 0;

}
```

If-else Introduciendo esta simple estructura de control:

```
/**
* IfElse.cpp
* Programa para probar If Else anindados
* En c++ no existe la estructura if-elsif-else
* Pello Xabier Altadill Izura
*
* Compilado: g++ IfElse.cpp -o IfElse
*/

using namespace std;
#include <iostream>

int main () {

 // Sacamos por salida standar un mensaje
 cout << "Vamos a probar los operadores\n";

 unsigned int test = 0;
 int a = 23, b = 21, c = 34;

 // Veamos el resto de valores
 cout << "Valores: " << c << " " << b << " " << a << endl;

 // Veamos una sentencia if-else sencilla
 if (a >= b)
  cout << "a mayor o igual que b" << endl;
 else
  cout << "a menor que b" << endl;

 // Veamos una sentencia if-else compleja
 // nota: si dentro de un if o un else metemos mas de una sentencia, hay que meter LLAV
 // y tambien conviene meter las llaves para hacer un codigo menos confuso
 if (a >= b) {
  cout << "a mayor o igual que b" << endl;
  if (a == 23) {
   cout << "a igual que 23" << endl;
   cout << "terminamos." << endl;
  }
 } else {
  cout << "a menor que b" << endl;
 }

return 0;

}
```

Capítulo 6. Parametros, ambito, sobrecarga

El camnio de c++ es largo, pero se sigue avanzando. Veamos las funciones inline, un recurso interesante para mejorar el rendimiento.

```
/**
* Inline.cpp
* Programa para probar funciones Inline
* Las funciones Inline no se compilan como funciones aparte,
* lo que se hace al compilar es a&ntilde;adir el contenido de la funcion haya
* donde se se invoca. Con lo que es mucho mas rapido de ejecutar
* y ademas nos da la limpieza de separar el codigo.
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Inline.cpp -o Inline
*/

using namespace std;
#include <iostream>

// las funciones en CPP las debemos declarar antes de invocar
// aqui tenemos el prototipo. Si no se pone tendremos ERROR de compilador
// Declaramos la funcion como inline
inline double Calcula (double a, double b);

// Log : saca un mensaje por pantalla
void Log(char *mensaje);

// Variables globales
long variable = 666;
char *PROGRAMA = "Globales> ";

int main () {

 // Sacamos por salida standar un mensaje
 Log("Vamos a probar los operadores");

 unsigned int test = 0;
 double a = 23, b = 21, c = 34;

 // Tomamos el valor a
 Log("Dame valores. \na=");
 cin >> a;

 // Tomamos el valor b
 cout << "b=";
 cin >> b;

 cout << "Y ahora son estos: b=" << b << " a=" << a << " global:" << variable << "Y el

 // Probamos la funcion
 Log("Venga va vamos");

 return 0;

}


/**
* Calcula
* parametros: double a, double b
* devuelve double
* En la implementacion no hace falta volver a poner INLINE
*/
```

```
double Calcula (double a, double b) {

 a *= 35462;
 b *=32546 + a;

 return (a / b) * variable;
}


/**
* Log
* parametros: char *mensaje
* devuelve void
*/
void Log (char *mensaje) {
 cout << PROGRAMA << mensaje << endl;
}
```

Paso de parametros Vamos a ver formas de pasar parametros.

```
/**
* Parametros.cpp
* Programa para probar los parametros de las funciones y
* la forma de aplicar valores por defecto
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Parametros.cpp -o Parametros
*/

using namespace std;
#include <iostream>

// las funciones en CPP las debemos declarar antes de invocar
// aqui tenemos el prototipo. Si no se pone tendremos ERROR de compilador
double Calcula (double a, double b);

// Log : saca un mensaje por pantalla
void Log(char *mensaje = "Sin valor prefijado");

// suma: suma dos valores
int Suma(int a = 0, int b = 0, int c = 0);

// Variables globales
long variable = 666;
char *PROGRAMA = "Globales> ";

int main () {
 // Sacamos por salida standar un mensaje
 Log("Vamos a probar los operadores");

 // Llamada sin parametros
 Log();

 unsigned int test = 0;
 int a = 23, b = 21, c = 34, d = 0;

 // Llamanda sin parametros
 d = Suma();

 cout << "Y el resultado de la funcion Suma sin parametros :" << d << endl;

 // Llamada con parametros
```

```
 d = Suma(a,b,c);

 cout << "Y el resultado de la funcion Suma :" << d << endl;

 // Probamos la funcion
 Log("Venga va vamos");

 return 0;

}

/**
* Calcula
* parametros: double a, double b
* devuelve: double
*/
double Calcula (double a, double b) {
 return (a / b) * variable;
}


/**
* Log
* parametros: char *mensaje
* devuelve: void
* NOTA: no hace falta volver a poner el valor prefijado
*/
void Log (char *mensaje) {
 cout << PROGRAMA << mensaje << endl;
}


/**
* Suma
* parametros: int a, int b, int c
* devuelve: int
*/
int Suma (int a = 0, int b = 0, int c = 0) {

 Log("Vamos a ver. Estamos en suma. ");
 // Devolvemos un valor
 return (a + b + c);

}
```

Sobrecarga de funciones La sobrecarga es otro concepto básico en la POO. Aqui se muestra un boton.

```
/**
* Sobrecarga.cpp
* Programa para probar la sobrecarga de funciones
* La sobrecarga es una misma funcion con distintos parametros
* Con la sobrecarga logramos el POLIMORFISMO de clases
y funciones
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Sobrecarga.cpp -o Sobrecarga
*/

using namespace std;
#include <iostream>

// las funciones en CPP las debemos declarar antes de invocar
```

```
// aqui tenemos el prototipo. Si no se pone tendremos ERROR de compilador
double Calcula (double a, double b);

int Calcula (int a, int b);

float Calcula (float a, float b);

// Log : saca un mensaje por pantalla
// Esto provocaria error de compilador por ambiguedad de sobrecarga
//void Log();
// Log : saca un mensaje por pantalla
// NOTA: el valor por defecto solo se pone en la DECLARACION
void Log(char *mensaje = "Sin valor prefijado");

// suma: suma dos valores
int Suma(int a = 0, int b = 0, int c = 0);

// Variables globales
long variable = 666;
char *PROGRAMA = "Globales> ";


int main () {

 // Sacamos por salida standar un mensaje
 Log("Vamos a probar los operadores");

 // Llamada sin parametros
 Log();

 unsigned int test = 0;
 int a = 23, b = 21, c = 34, d = 0;

 // Llamanda sin parametros
 d = Suma();

 cout << "Y el resultado de la funcion Suma sin parametros :" << d << endl;

 // Llamada con parametros
 d = Suma(a,b,c);

 cout << "Y el resultado de la funcion Suma :" << d << endl;

 // Probamos la funcion
 Log("Venga va vamos");

 return 0;

}


/**
* Calcula
* parametros: double a, double b
* devuelve: double
*/
double Calcula (double a, double b) {

 return (a / b) * variable;

}


/**
* Calcula
* parametros: float a, float b
```

```
* devuelve: float
*/
float Calcula (float a, float b) {

 return (a / b) * variable;

}


/**
* Calcula
* parametros: long a, long b
* devuelve: long
*/
long Calcula (long a, long b) {

 return (a / b) * variable;

}


/**
* Log
* parametros: char *mensaje
* devuelve: void
*/
void Log (char *mensaje) {

 cout << PROGRAMA << mensaje << endl;

}


/**
* Suma
* parametros: int a, int b, int c
* devuelve: int
*/
int Suma (int a = 0, int b = 0, int c = 0) {

 Log("Vamos a ver. Estamos en suma. ");

 // Devolvemos un valor
 return (a + b + c);

}
```

El ambito Hasta donde se identifica una variable? Para saltarnos todas las vallas podemos usar variables globales. No conviene abusar de este tipo de variables.

```
/**
* Globales.cpp
* Programa para probar variables y su scope
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Globales.cpp -o Globales
*/

using namespace std;
#include <iostream>

// las funciones en CPP las debemos declarar antes de invocar
// aqui tenemos el prototipo. Si no se pone tendremos ERROR de compilador
double Calcula (double a, double b);
```

```
// Log : saca un mensaje por pantalla
void Log(char *mensaje);

// Variables globales
long variable = 666;
char *PROGRAMA = "Globales> ";


int main () {

 // Sacamos por salida standar un mensaje
 Log("Vamos a probar los operadores");

 unsigned int test = 0;
 double a = 23, b = 21, c = 34;

 // Tomamos el valor a
 Log("Dame valores. \na=");
 cin >> a;

 // Tomamos el valor b
 cout << "b=";
 cin >> b;

 cout << "Y ahora son estos: b=" << b << " a=" << a << " global:" << variable <<< "Y el

 // Probamos la funcion
 Log("Venga va vamos");

 return 0;

}


/**
* Calcula
* parametros: double a, double b
* devuelve double
*/
double Calcula (double a, double b) {

 return (a / b) * variable;

}


/**
* Log
* parametros: char *mensaje
* devuelve void
*/
void Log (char *mensaje) {

 cout << PROGRAMA << mensaje << endl;

}
```

Capítulo 7. Clases

Tu primera clase c++ No hay que perder de vista el hecho de que c++ es un lenguaje orientado a objetos. Sin animos de volver a explicar que es la POO, los beneficios que constituye vamos a limitarnos a resumir. Una clase c++ es la representacion de un objeto. Un objeto es una entidad formada por sus atributos y sus metodos. Con el afan de hacer las cosas ordenadamente, siempre se separa la definicion de la clase en un fichero de cabedeceras (extension .hpp, similar al .h de lenguaje c) y la implementacion se especifica en un fichero cpp. Generalmente las clases c++ tienen el mismo aspecto: se definen unos atributos y unos metodos. Entre los metodos se pueden incluir metodos constructores y la destructora. Ademas de eso se puede definir si los atributos y clases son publicas, protegidas y privadas, dependiendo del nivel de encapsulacion que le queramos dar a la clase. Veamos la representacion del objeto coche en una clase c++:

```
/**
* Coche.hpp
* Clase cabecera que define el objeto Coche
*
* Pello Xabier Altadill Izura
*
* No se compila.
*/

using namespace std;
#include <iostream>

class Coche {

public:

 Coche();

 Coche(char *m,int cil,int cab);

 ~Coche();

 void arranca();

 void detiene();

 void acelera();

private:

 char *marca;

 int cilindrada;

 int caballos;

};
```

Y este seria el fichero de implementacion (se puede tener todo en un unico fichero)

```
/**
* Coche.cpp
* Fichero que implementa la cabecera de la clase Coche.
* NO HACE NADA CONCRETO solo es una muestra
*
* Pello Xabier Altadill Izura
*
```

```
* Compilar usando: g++ -c Coche.cpp
*/

// Hay que incluir el fichero de cabecera
#include "Coche.hpp"

// Implementacion de constructor
Coche::Coche() {

 cout << "Coche creado." << endl;

}


// Implementacion de constructor (con SOBRECARGA)
Coche::Coche (char *m,int cil,int cab) {}


// Implementacion de destructor. Util para liberar memoria.
Coche::~Coche() {

 cout << "Coche destruido." << endl;

}


// implementaciones de metodos...
void Coche::arranca() {}

void Coche::detiene() {}

void Coche::acelera() {}

/**
* Podemos usar una clase main para hacer testeos con la clase
* NOTA IMPORTANTE
* Atencion : al usar esta clase en otra que ya tiene funcion
* main, no se puede tener otra main.
*/
//int main () {
//cout << "Lo hise!!\n" << endl;
//return 1;
//}
```

Podemos usar clases dentro de otras clases? si claro. Veamos la definicion de un Garaje.

```
/**
* Garaje.hpp
* Cabecera del objeto Garaje
*
* En este caso invocamos otro objeto: Coche
*
* Pello Xabier Altadill Izura
*
* La cabecera como tal no se compila
*/

using namespace std;
#include <iostream>
#include "Coche.hpp"

/*
* Definicion de clase Garaje
```

```
*/
class Garaje {

private:

 int maxCoches;

public:

 Garaje();

 Garaje(int maxCoches);

 ~Garaje();

 int entra(Coche coche);

 int sale(Coche coche);

 bool estaLleno();

};
```

Y esta seria la implementacion:

```
/**
* Garaje.cpp
* Implementacion de Clase Garaje
*
* Pello Xabier Altadill Izura
* Atencion: necesitamos el archivo objeto de la clase Coche!!!
* Compilar con: g++ -c Coche.cpp
* g++ -Wall Garaje.cpp Coche.o -o Garaje
*/

#include "Garaje.hpp"

/*
* Implementacion de clase Garaje
*/

/**
* Constructor por defecto
*/
Garaje::Garaje(){

 cout << "Garaje." << endl;

 maxCoches = 3;

}


/**
* Constructor parametrizado
*/
Garaje::Garaje(int mx){

  maxCoches = mx;

}


/**
```

```
* Destructor
*/
Garaje::~Garaje(){}


/**
* entra: un coche entra en el garaje
*/
int Garaje::entra(Coche coche) {

 cout << " Entra un coche." << endl;
 return 0;

}


/**
* sale: un objeto coche sale del garaje
*/
int Garaje::sale(Coche coche) {

 cout << " Sale un coche." << endl;
 return 0;

}

/**
* estaLleno?: devuelve booleano con la respuesta
*/
bool Garaje::estaLleno() {

 return false;

}


/**
* y aqui la funcion main para hacer nuestras pruebillas
*/
int main () {

 cout << " Creamos un garaje. " << endl;
 Garaje garaje = Garaje();

 // Creamos un par de Coches
 Coche cocheAzul = Coche();
 Coche cocheRojo = Coche();

 // Metemos y sacamos los coches
 garaje.entra(cocheAzul);
 garaje.entra(cocheRojo);
 garaje.sale(cocheRojo);

}
```

Funciones o metodos Setter/Getter Por mania o por costumbre o porque asi lo establecen los puristas mas talibanes de la POO casi nunca se deja acceder directamente a los atributos de una clase (se definen como private) y para acceder a ellos se implementan funciones set/get. Las herramientas de desarrollo suelen incluir la opcion de generar ese codigo de forma automatizada.

Figura: nunca tomes a broma a un desarrollador OO

Sin la menor intencion de alimentar la ya tradicional Yihad entre desarrolladores, mostremos un ejemplo y digamos de paso que no esta demas definir estas funciones como inline; cumplimos como profesionales pero no perdemos eficacia. El objeto PERRO

```
/**
* Perro.hpp
* Cabecera de la clase Perro con sus funciones get/set para el atributo edad
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Perro {

public:

 Perro (int initialAge);

 ~Perro();

 int GetAge() { return itsAge;} // inline?

 void SetAge (int age) { itsAge = age;} // inline?

 void Ladra() { cout << "Guau Guau arrr...\n";} // inline?

private:

 int itsAge;

};
```

Y su implementacion

```
/**
* Perro.cpp
* Clase que implementa la clase Perro
*
```

```
* Pello Xabier Altadill Izura
*
* Compilado: g++ Perro.cpp -o Perro
*/

#include "Perro.hpp"

Perro::Perro(int initialAge) //constructor
{

 itsAge = initialAge;

}


Perro::~Perro() //destructor
{
 cout << " objeto destruido." << endl;
}


/**
* La funcion principal, crea un perro y le hace ladrar
*/
int main() {

 bool test = false;

 Perro Canelo(5);

 Canelo.Ladra();

 cout << "Canelo es un perro cuya edad es: " ;
 cout << Canelo.GetAge() << " a&ntilde;os\n";

 Canelo.Ladra();

 Canelo.SetAge(7);

 cout << "Ahora Canelo es " ;
 cout << Canelo.GetAge() << " a&ntilde;os\n";

 return 0;

}
```

Capítulo 8. Iteraciones

Iteracion con etiquetas Es la manera primigenia de implementar iteraciones pero lo cierto es que el uso de etiquetas no se recomienda ya que es dificil de entender y mantener un programa con etiquetas. A ser posible hay que evitarlas.

```
/**
* Loop.cpp
*
* Programa que muestra como usar iteraciones
* Pello Xabier Altadill Izura
* Compilar: g++ Loop.cpp -o Loop
*/

using namespace std;
#include <iostream>

// Programa principal
int main () {

 cout << " Hola, vamos a mostrar un loop " << endl;

 //Inicializamos variables
 int i = 0;
 int max = 0;

 // Le pedimos al usuario que meta el total de iteraciones
 cout << " Cuantas vueltas damos? ";
 cin >> max;

 // Vamos a implementar una iteracion con etiquetas
 // en general no es recomendable usar etiquetas
 bucle: i++;
 cout << "Contador: " << i << endl;

 // si no superamos el tama&ntilde;o maximo, volvemos a la etiqueta
 if (i < max)
  goto bucle;

 // fin de programa

 return 0;

}
```

Bucles Bueno, ahora en forma de clase vamos a ver una serie de iteraciones. No tienen ningun misterio se implementan como en c, php, perl, java, ...

```
/**
* Bucles.hpp
*
* Clase que muestra distintos tipos de iteraciones
* Pello Xabier Altadill Izura
*/

using namespace std;
#include <iostream>

class Bucles {

private:

 int max;
```

```
public:

 // Constructor
 Bucles();

 // Destructor
 ~Bucles();

 // Contructor parametrizado
 Bucles(int maximo);

 // Bucle tipo while
 void bucleWhile(int maximo);

 // Bucle tipo for
 void bucleFor(int maximo);

 // Bucle tipo do while
 void bucleDoWhile(int maximo);

};
```

Y aqui la implementacion

```
/**
* Bucles.cpp
*
* Clase que muestra distintos tipos de iteraciones
* Pello Xabier Altadill Izura
* Compilar: g++ Bucles.cpp -o Bucles
*/

#include "Bucles.hpp"

// Constructor
Bucles::Bucles(){}


// Destructor
Bucles::~Bucles(){}


// Contructor parametrizado
Bucles::Bucles(int maximo){}


// Bucle tipo while
void Bucles::bucleWhile(int maximo){

 int temp = 0;
 cout << " iniciamos el bucle WHILE: " << temp << " y
 max " << maximo << endl;

 while (temp < maximo) {
  cout << temp << " es menor que " << maximo << endl;
  temp++;
 }

}


// Bucle tipo for
void Bucles::bucleFor(int maximo){
```

```
 int temp = 0;
 cout << " iniciamos el bucle FOR: " << temp << " y max " << maximo << endl;

 for (temp=0; temp < maximo; temp++) {

  cout << temp << " es menor que " << maximo << endl;

 }

}


// Bucle tipo do while
void Bucles::bucleDoWhile(int maximo){

 int temp = 0;
 cout << " iniciamos e bucle: " << temp << " y max " << maximo << endl;

 do {

  cout << temp << " es menor que " << maximo << endl;
  temp++;

 } while (temp < maximo);

}


int main () {

 // Creamos dos instancias de la clase Bucles
 Bucles ciclador = Bucles();
 Bucles cicladorparam = Bucles(34);

 // Invocamos los metodos
 ciclador.bucleWhile(23);

 cicladorparam.bucleFor(10);

 ciclador.bucleDoWhile(5);

 return 0;

}
```

Switch/case Por supuesto tenemos el clasico switch-case en c++ En este ejemplo creamos una clase para mostrar el funcionamiento de un menu de seleccion.

```
/**
* Menu.hpp
*
* Clase que especifica un menu de seleccion de opciones
* que implementaremos con un case
* Pello Xabier Altadill Izura
*/

using namespace std;
#include <iostream>

class Menu {

private:

  int resultado;
```

```
public:

 // Constructor
 Menu();

 // Destructor
 ~Menu();

 // Menu tipo case
 int menu();

};
```

Y su implementacion

```
/**
* Menu.cpp
*
* Clase que implementa Menu.hpp
* Pello Xabier Altadill Izura
* Compilar: g++ Menu.cpp -o Menu
*/

#include "Menu.hpp"

// Constructor
Menu::Menu(){}


// Destructor
Menu::~Menu(){}


// Menu tipo case
int Menu::menu(){

 int temp = 0;

 // Iniciamos un bucle que no para hasta que se seleccione
 // algo distinto de 0.

 do {
  cout << " MENU Seleccion." << endl;
  cout << " 1. Ensalada" << endl;
  cout << " 2. Cordero " << endl;
  cout << " 3. Merluza " << endl;
  cout << " 4. Pato " << endl;
  cout << " Elije algo: ";
  cin >> temp;

  // Segun lo elegido sacamos algo.

  switch (temp) {
   case 0 :
    cout << " Nos vamos " << endl;
    break;

   case 1 :
    cout << " Estas a dieta? " << endl;
    break;

   case 2 :
    cout << " Vaya digestion... " << endl;
    break;
```

```
   case 3 :
    cout << " Que sano eres " << endl;
    break;

   case 4 :
    cout << " Vaya finolis esta hecho " << endl;
    break;

   default :
    cout << " Chico, decidete." << endl;
    temp = 0;
  } //end switch

 } while(!temp);

 return temp;

}


int main () {

 // Aqui guardaremos el resultado
 int resultado = 0;

 cout << " Vamos a sacar el menu." << endl;

 // Creamos dos instancias de la clase Menu
 Menu menutero = Menu();

 // Invocamos los metodos
 resultado = menutero.menu();

 cout << " El resultado es: " << resultado << endl;

 return 0;

}
```

Largo es el camino. Bueno, aun queda un huevo por recorrer...

Capítulo 9. Punteros

Los punteros Acaso creiais que en c++ no habia punteros? eso solo ocurre en Java. Los punteros no contienen datos, contienen direcciones de memoria. Para cada tipo de dato hay que definir un puntero.

```
/**
* Puntero.cpp
*
* Clase que muestra las direcciones de variables
* Pello Xabier Altadill Izura
* Compilar: g++ Puntero.cpp -o Puntero
*/

using namespace std;
#include <iostream>

int main () {

 // Creamos varias variables:
 int pruebaInt = 99, prueba2Int;

 short pruebaShort = 34;
 char carac = 'a';
 int *puntero = 0;
 int *punteroNuevo;

 // Ahora las mostramos por pantalla:
 cout << "Variable pruebaInt: " << pruebaInt << endl;
 cout << "Direccion pruebaInt: " << &pruebaInt << endl << endl;

 cout << "Variable prueba2Int: " << prueba2Int << endl;
 cout << "Direccion prueba2Int: " << &prueba2Int << endl << endl;

 cout << "Variable pruebaShort: " << pruebaShort << endl;
 cout << "Direccion pruebaShort: " << &pruebaShort << endl << endl;

 cout << "Variable carac: " << carac << endl;
 cout << "Direccion carac: " << &carac << endl << endl;

 cout << "Variable puntero: " << puntero << endl;

 // ATENCION, si el puntero no tiene valor dara
 // SEGMENTATION FAULT y la CAGAREMOS de gordo
 //cout << "Variable puntero: " << *puntero << endl;

 cout << "Direccion puntero: " << &puntero << endl << endl;

 puntero = &pruebaInt;
 cout << "Variable puntero: " << puntero << endl;
 cout << "Variable puntero: " << *puntero << endl;
 cout << "Direccion puntero: " << &puntero << endl << endl;

 return 0;

}
```

Veamos otro ejemplo...

```
/**
* Puntero2.cpp
*
* Clase que muestra mas usos de los punteros
* Pello Xabier Altadill Izura
```

```
* Compilar: g++ Puntero2.cpp -o Puntero2
*/

using namespace std;
#include <iostream>

// prototipo de funciones que implementamos luego
int devuelve(int *punteroInt, int entero);


int main () {

 // Creamos varias variables:
 int pruebaInt = 99, prueba2Int;
 short pruebaShort = 34;
 char carac = 'a';
 int *puntero = 0;
 int *punteroNuevo;

 // Ahora las mostramos por pantalla:
 cout << "Variable pruebaInt: " << pruebaInt << endl;
 cout << "Direccion pruebaInt: " << &pruebaInt << endl << endl;

 cout << "Variable prueba2Int: " << prueba2Int << endl;
 cout << "Direccion prueba2Int: " << &prueba2Int << endl << endl;

 cout << "Variable pruebaShort: " << pruebaShort << endl;
 cout << "Direccion pruebaShort: " << &pruebaShort << endl << endl;

 cout << "Variable carac: " << carac << endl;
 cout << "Direccion carac: " << &carac << endl << endl;

 cout << "Variable puntero: " << puntero << endl;

 // ATENCION, si el puntero no tiene valor dara
 // SEGMENTATION FAULT y la CAGAREMOS
 //cout << "Variable puntero: " << *puntero << endl;
 cout << "Direccion puntero: " << &puntero << endl << endl;
 puntero = &pruebaInt;
 cout << "Variable puntero: " << puntero << endl;
 cout << "Variable puntero: " << *puntero << endl;
 cout << "Direccion puntero: " << &puntero << endl << endl;

 *puntero = 345;

 cout << "Variable puntero: " << puntero << endl;
 cout << "Variable puntero: " << *puntero << endl;
 cout << "Direccion puntero: " << &puntero << endl << endl;

 // Ahora las mostramos por pantalla:
 cout << "Variable pruebaInt: " << pruebaInt << endl;
 cout << "Direccion pruebaInt: " << &pruebaInt << endl << endl;

 *punteroNuevo = devuelve(puntero,34);

 cout << " Tras llamada: " << endl;
 cout << "Variable puntero: " << punteroNuevo << endl;
 cout << "Variable puntero: " << *punteroNuevo << endl;
 cout << "Direccion puntero: " << &punteroNuevo << endl << endl;

 return 0;

}


int devuelve (int *punteroInt, int entero) {
```

```
 cout << "Variable param. puntero: " << punteroInt << endl;
 cout << "Variable param. puntero: " << *punteroInt << endl;
 cout << "Direccion param. puntero: " << &punteroInt << endl << endl;

 return (*punteroInt) + entero;

}
```

new y delete Con las instrucciones new y delete podemos reservar y liberar espacio libre de memoria. Se utilizan con los punteros (ademas de los objetos) y es muy necesario liberar siempre la memoria con la instruccion delete para evitar memory leaks: espacio de memoria marcados como okupados pero que ya no se usan porque el puntero que les correspondia ahora apunta a otro lado.

```
/**
* Puntero.cpp
*
* Clase que muestra la okupacion/liberacion de memoria
con new y delete
* Pello Xabier Altadill Izura
* Compilar: g++ Puntero.cpp -o Puntero
*/

using namespace std;
#include <iostream>

int main () {

 // Creamos varias variables:
 int *pruebaInt = new int;
 short *pruebaShort = new short;
 pruebaInt = 777;
 pruebaShort = 23;

 // Ahora las mostramos por pantalla:
 cout << "Variable pruebaInt: " << pruebaInt << endl;
 cout << "Direccion pruebaInt: " << &pruebaInt << endl << endl;
 cout << "Variable pruebaShort: " << pruebaShort << endl;
 cout << "Direccion pruebaShort: " << &pruebaShort << endl << endl;

 // Liberamos la memoria
 delete pruebaInt;
 delete pruebaShort;

 // Contra la especulacion del sistema (operativo)
 // volvemos a oKupar un espacio de memoria
 int *pruebaInt = new int;
 short *pruebaShort = new short;
 pruebaInt = 666;
 pruebaShort = 21;

 // quiza tengamos un error, pero se puede comprobar:
 if ( pruebaInt == NULL || pruebaShort == NULL ) {

  cout << "Error al reservar memoria" << endl;
  return 0;

 }

 // Ahora las mostramos por pantalla:
 cout << "Variable pruebaInt: " << pruebaInt << endl;
 cout << "Direccion pruebaInt: " << &pruebaInt << endl << endl;
 cout << "Variable pruebaShort: " << pruebaShort << endl;
 cout << "Direccion pruebaShort: " << &pruebaShort << endl << endl;
```

```
 return 0;

}
```

Objetos y punteros Se pueden crear punteros a objetos y atributos que son punteros. Veamos este ejemplo de una clase llamada Objeto:

```
/**
* Objeto.hpp
*
* Clase que muestra distintos tipos de punteros
* que se usan con los objetos
*
* Pello Xabier Altadill Izura
*/

using namespace std;
#include <iostream>

// Inicio de la clase
class Objeto {

private:

 int *privado;

public:

 int atributo;

 // Constructor
 Objeto();

 // Constructor
 Objeto(int atributo);

 // Destructor
 ~Objeto();

 // Menu tipo case
 int devuelveAlgo();

};
```

Y su implementacion:

```
/**
* Objeto.cpp
*
* Clase que muestra distintos tipos de punteros
* que se usan con los objetos
* Pello Xabier Altadill Izura
* Compilar: g++ Objeto.cpp -o Objeto
*/

#include "Objeto.hpp"

// Constructor
Objeto::Objeto(){

 atributo = 666;

}
```

```
// Constructor
Objeto::Objeto(int atributo){

 this->atributo = atributo;

}


// Destructor
Objeto::~Objeto(){}


// Menu tipo case
int Objeto::devuelveAlgo(){

 int temp = 0;

 return temp;

}


int main () {

 // Aqui guardaremos el resultado
 int resultado = 0;

 cout << " Vamos a jugar con los objetos." << endl;

 // Creamos la instancia del objeto puntero
 Objeto objeto = Objeto();

 //Creamos un puntero a ese objeto,
 // pero cuidado, no asignarle un constructor directamente
 Objeto *objetopuntero;

 // esto si...
 objetopuntero = &objeto;

 // Invocamos los metodos
 resultado = objeto.devuelveAlgo();

 // Observese la diferencia al acceder al atributo publico:
 cout << " El valor de atributo con Objeto es: " << objeto.atributo << endl;
 cout << " El valor de atributo con Objeto es: " << objetopuntero->atributo << endl;

 return 0;

}
```


I SUSPECT THAT PETER HAS A PROBLEM
HA!! LIKE ANY OF US
KORN
SSSH... IT'S A SERIOUS ONE
OH REALLY?
I FOUND HIM HIDDEN IN A DATACENTER CORNER...
KORN
(C) 2003 - Pello Altadill - http://coders.pello.info

Capítulo 10. Referencias

Las referencias Una referencia es otra forma de acceder a un dato, una especie de alias. Cualquier operacion sobre una referencia afectara a ese dato al que hace referencia.

Figura: sin duda los punteros y las referencias fueron obra de los sarracenos.

Veamos un ejemplo simple:

```
/**
* Referencias.cpp
* Programa que muestra el uso de referencias
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ Referencias.cpp -o Referencias
*/

using namespace std;
#include <iostream>

int main() {

 // Definimos un dato y su referencia
 int numero;
 int &referenciaNumero = numero; // Ahi se crea la referencia

 cout << "Vamos a ver que pasa si le asignamos un dato: " << endl;
```

```
 numero = 31337;

 // Los dos mostraran el mismo valor
 cout << "Valor de numero: " << numero << endl;
 cout << "Valor de referenciaNumero: " << referenciaNumero << endl;

 // y a donde apuntan? AL MISMO SITIO
 cout << "Posicion de numero: " << &numero << endl;
 cout << "Posicion de referenciaNumero: " << &referenciaNumero << endl;
 cout << "Programa terminado \n" << endl;

 return 0;

}
```

Con los objetos se pueden hacer referencias igualmente:

```
Objeto miObjeto;
Objeto &refObjeto = miObjeto;
```

Referencias y funciones Vamos a ver distintas formas de pasar referencias a una funcion. Como en c, podemos pasar parametros por referencia y hacer que esos parametros contengan resultados de una funcion.

```
/**
* ReferenciaFunciones.cpp
* Programa que muestra el uso de referencias en las funciones
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ ReferenciaFunciones.cpp -o ReferenciaFunciones
*/

using namespace std;
#include <iostream>

// 1Âº funcion que intercambia dos valores
void exchange (int *refa, int *refb);

// 2Âº funcion -sobrecargada- que intercambia dos valores
void exchange (int &refa, int &refb);


int main() {

 // Definimos un dato y su referencia
 int a, b;

 cout << "Asignamos valores: " << endl;
 a = 45;
 b = 21;

 cout << "Valores: a=" << a << " b=" << b << endl;
 cout << "Hacemos intercambio con exchange(int *refa, int *refb): " << endl;

 exchange(&a, &b); // Con esta llamada invocamos la primera funcion!!

 cout << "Valores: a=" << a << " b=" << b << endl;
 cout << "Hacemos intercambio con exchange(int &refa, int &refb): " << endl;

 xchange(a, b); // Con esta llamada invocamos la segunda funcion!!

 out << "Valores: a=" << a << " b=" << b << endl;
```

```
 out << "Programa terminado \n" << endl;

 return 0;

}

// 1Âº funcion que intercambia dos valores
void exchange (int *refa, int *refb) {

 int tmp;

 tmp = *refa;
 *refa = *refb;
 *refa = tmp;

}


// 2Âº funcion -sobrecargada- que intercambia dos valores
void exchange (int &refa, int &refb) {

 int tmp;

 tmp = refa;
 refa = refb;
 refa = tmp;

}
```

Pasando clases por referencia

```
/**
* Gremlin.hpp
*
* Clase que representa el objeto Gremlin.
* Observese el 3Âº metodo constructor
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Gremlin {

public:

 Gremlin();

 Gremlin(char *nmb,int ed, int p);

 Gremlin(Gremlin&); // atencion a este constructor

 ~Gremlin();

 void correr();

 void dormir();

 void morder();

 int peso;

private:
```

```
 char *nombre;

 int edad;

};
```

Y su implementacion:

```
/**
* Gremlin.cpp
*
* Clase que implementa el objeto Gremlin.
* Pello Xabier Altadill Izura
*
*/

#include "Gremlin.hpp"

Gremlin::Gremlin() {

 peso = 1;

 cout << "Gremlin creado." << endl;

}


Gremlin::Gremlin (char *nmb,int ed, int p) {

 nombre = nmb;
 edad = ed;
 peso = p;
}


Gremlin::~Gremlin() {

 cout << "Aaaargh!\nGremlin destruido." << endl;

}



// El gremlin corre
void correr() {

 cout << "Jaja grrrr!! jajaja!" << endl;

}


// El gremlin duerme
void dormir() {

 cout << "zzzZZZZzzzzz" << endl;

}


// El gremlin muerde
void morder() {

 cout << "roaar &ntilde;am &ntilde;am" << endl;

}
```

```
// Definimos esta funcion aparte de la clase
// Con ella el gremlin come y aumenta su atributo peso.
void comer (Gremlin *g) {

 // Invocamos la mordedura para que coma
 g->morder();

 // Le aumentamos 3 unidades por comer
 g->peso += 3;

}


// Funcion main
int main () {

 cout << "Iniciando programa. " << endl;

 // Definimos un gremlin
 Gremlin tbautista;

 // y lo movemos por la ciudad
 tbautista.correr();
 tbautista.morder();

 // Mostramos su peso
 cout << "El gremlin pesa: " << tbautista.peso << endl;

 // Le hacemos comer:
 comer(&tbautista);

 // Mostramos su peso otra vez
 cout << "El gremlin pesa ahora: " << tbautista.peso << endl;
 cout << "Finalizando programa\n " << endl;

 return 0;

}
```

La ventaja que logramos al pasar parametros por referencia es que ahorramos espacio en memoria ya que sino en cada llamada a una funcion se hacen copias de los parametros. Esto tambien tiene una desventaja: si le pasamos a una funcion el ORIGINAL de un objeto (con una referencia) en lugar de una copia corremos el riesgo de que la funciona haga trizas nuestro objeto y perder el "original" (supongamos que la funcion esta hecha por terceros y no sabemos lo que hace). Que se puede hacer para salvaguardar nuestros objetos? Punteros constantes Esta es la solucion: pasar punteros constantes. Eso hara que la funcion solo tenga permiso para invocar los metodos constantes de la clase. SE cambia un poco la clase gremlin para mostrar esto.

```
/**
* Gremlin2.hpp
*
* Clase que representa el objeto Gremlin.
* Con un metodo definido como const!!
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>
```

```
class Gremlin {

public:

 Gremlin();

 Gremlin(char *nmb,int ed, int p);

 Gremlin(Gremlin&); // atencion a este constructor

 ~Gremlin();

 void correr();

 void dormir();

 void morder();

 // Definimos una funcion constante
 char * getNombre() const;

 int peso;

private:

 char *nombre;

 int edad;

};
```

Y vemos la implementacion en la que simplemente se puede observar como se protege el objeto en la funcion comer() gracias al uso de punteros constantes.

```
/**
* Gremlin2.cpp
*
* Clase que implementa el objeto Gremlin.
* Pello Xabier Altadill Izura
*
*/

#include "Gremlin2.hpp"

Gremlin::Gremlin() {

 peso = 1;

 cout << "Gremlin creado." << endl;

}


Gremlin::Gremlin (char *nmb,int ed, int p) {

  nombre = nmb;
  edad = ed;
  peso = p;

}


Gremlin::~Gremlin() {

 cout << "Aaaargh!\nGremlin destruido." << endl;
```

```
}


// El gremlin corre
void correr() {

 cout << "Jaja grrrr!! jajaja!" << endl;

}


// El gremlin duerme
void dormir() {

 cout << "zzzZZZZzzzzz" << endl;

}


// El gremlin muerde
void morder() {

  cout << "roaar &ntilde;am &ntilde;am" << endl;

}


// FUNCION CONST!!!
// Devuelve el nombre del gremlin
char * getNombre() const {

  return nombre;

}


// Definimos esta funcion aparte de la clase
// Con ella el gremlin come y aumenta su atributo peso.
void comer (const Gremlin const *g) {

 // Invocamos la mordedura para que coma??
 // g->morder(); ERROR no podemos invocar una funcion NO CONSTANTE!!!
 // en cambio si podemos invocar getNombre
 cout << "Nombre" << g->getNombre() << endl;

}


// Funcion main
int main () {

 cout << "Iniciando programa. " << endl;

 // Definimos un gremlin
 Gremlin tbautista;

 // y lo movemos por la ciudad
 tbautista.correr();
 tbautista.morder();

 // Mostramos su peso
 cout << "El gremlin pesa: " << tbautista.peso << endl;

 // Le hacemos comer:
 comer(&tbautista);
```

```
 // Mostramos su peso otra vez
 cout << "El gremlin pesa ahora: " << tbautista.peso << endl;
 cout << "Finalizando programa\n " << endl;

 return 0;

}
```

Capítulo 11. Funciones avanzadas

Sobrecarga y valores por defecto En un clase se pueden sobrecargar los metodos y los constructores, e incluso se pueden asignar valores por defecto a los parametros (como en php). Veamos el ejemplo del coche un poco mas desarrollado.

```
/**
* Coche.hpp
* Clase que representa un coche
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Coche {

private:

 char *marca;

 int cilindrada;

 int caballos;

 enum marcha { Primera, Segunda, Tercera, Cuarta, Quinta, Pto_Muerto};

public:

 Coche();

 Coche(int cilindrada,int caballos);

 Coche(char *marca,int cilindrada,int caballos);

 ~Coche();

 void arranca();

 void avanza(int metros = 5); // Con valor por defecto

 void cambiaMarcha(marcha mar);

 void cambiaMarcha();

 void detiene();

 void acelera();


 char * getMarca ();

 int getCilindrada ();

 int getCaballos ();
};
```

Y esta su implementacion observense las funciones sobrecargadas y los posibles errores que se pueden cometer.

```
/**
```

```
* Coche.cpp
* Fichero que implementa la clase coche
*
* Pello Xabier Altadill Izura
*
*/

#include "Coche.hpp";

// Constructor por defecto
Coche::Coche() {

 cout << "Coche creado." << endl;

}


// Constructor sobrecargado CON VALORES POR DEFECTO
// si no se establece otra cosa se asignan esos valores
Coche::Coche (int cilindrada = 1000, int caballos = 100) {

 this->marca = "Cualquiera";
 this->cilindrada = cilindrada;
 this->caballos = caballos;

}


// Constructor sobrecargado
Coche::Coche (char *marca,int cilindrada,int caballos) {

 this->marca = marca;
 this->cilindrada = cilindrada;
 this->caballos = caballos;

}


// Destructor
Coche::~Coche() {

 cout << "Coche destruido." << endl;

}


void Coche::arranca() {}


void Coche::detiene() {}


void Coche::acelera() {}


// Metodo para que el coche avance. Esta definico con un valor
// por defecto (5) por tanto podria invocarse SIN parametro alguno
void Coche::avanza(int metros) {

 cout << this->marca << " ha avanzado " << metros << metros << endl;

}

// Metodo para que el coche cambie de marcha
void Coche::cambiaMarcha() {}
```

```
// Metodo -sobrecargado- para que el coche cambie de marcha
void Coche::cambiaMarcha(marcha mar) {}

// Muestra la marca
char * Coche::getMarca () {

 return this->marca;

}


// Muestra la cilindrada
int Coche::getCilindrada () {

 return this->cilindrada;

}


// Muestra los caballos
int Coche::getCaballos (){

 return this->caballos;

}


/**
* NOTA IMPORTANTE
* Atencion : al usar esta clase en otra que ya tiene funcion
* main, no se puede tener otra main.
*/
int main () {

 int test = 0;

 Coche vehiculo = Coche("Skoda", 1050, 250);

 cout << "Lo hice, tengo un: "<< vehiculo.getMarca() << endl;

 vehiculo.arranca();
 vehiculo.cambiaMarcha();
 vehiculo.avanza();

 // ATENCION!! esto seria una llamada ambigua, ya que existe otro constructor
 // que se puede asignar sin parametros pq tiene valores por defecto que es esta:
 // Coche::Coche (int cilindrada = 1000, int caballos = 100) y choca con el constructor
 // por defecto. Boludos! el compilador nos rompera el ORTO sin compasion
 //Coche segundoCoche = Coche();

 return 0;

}
```

Se puede implementar el constructor de otra manera (sirve para tirarte el rollete guru, aunque te seguiran pagando igual de mal), atencion a la sintaxis.

```
Coche::Coche(): marca("Seat"), cilindrada(120) {

};
```

Copy constructor Este es un constructor que se puede añadir a nuestras clases y que sirve para hacer una copia de un objeto de esa clase. Existe uno por defecto pero es

recomendable preocuparse en implementarlo nosotros mismos ya que pueden producirse errores con atributos que son punteros. Veamos el copy de la clase Perro.

```
/**
* Perro.hpp
* Clase de cabecera de Perro
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Perro {

public:

 Perro (int initialAge);

 // constructor COPY
 Perro (const Perro &);

 ~Perro();

 // metodos YA implementados
 int GetAge() { return itsAge;} // automaticamente inline!

 void SetAge (int age) { itsAge = age;} // automaticamente inline!

 int * GetPeso() { return peso;} // automaticamente inline!

 void SetPeso (int * peso) { this->peso = peso;} // automaticamente inline!

 char * GetRaza() { return raza;} // automaticamente inline!

 void SetRaza (char * raza) { this->raza = raza;} // automaticamente inline!

 char * GetColor() { return color;} // automaticamente inline!

 void SetColor (char *color) { this->color = color;} // automaticamente inline!

 void Ladra() { cout << "Guau Guau arrr...\n";} // automaticamente inline!

private:

 int itsAge;

 int *peso;

 char *raza;

 char *color;

};
```

Y su implementacion

```
/**
* Perro.cpp
* Clase que implementa la clase Perro con constructor copy
*
* Pello Xabier Altadill Izura
*
```

```
* Compilado: g++ Perro.cpp -o Perro
*/

#include "Perro.hpp"

//constructor
Perro::Perro(int initialAge) {

 itsAge = initialAge;

 cout << "Creado chucho." << endl;

}

//copy-constructor. Atencion
Perro::Perro(const Perro & perroOrigen) {

 itsAge = perroOrigen.itsAge;
 peso = new int;
 raza = new char;
 color = new char;
 color = perroOrigen.color;
 raza = perroOrigen.raza;
 peso = perroOrigen.peso;
 cout << "Creado chucho con copia" << endl;

}

//destructor
Perro::~Perro() {

 cout << " objeto destruido." << endl;

}


/**
* La funcion principal, crea un perro y le hace ladrar
*/
int main()
{

 int t = 0;
 bool test = false;

 Perro Canelo(5);

 Canelo.SetRaza("Pastor vasco");

 // Creamos a Laika haciendo una copia de canelo
 Perro Laika(Canelo);

 cout << "Laika es de raza " ;
 cout << Laika.GetRaza() << endl;

 Laika.SetRaza("Sovietica");
 Canelo.Ladra();

 cout << "Canelo es un perro cuya edad es: " ;
 cout << Canelo.GetAge() << " a&ntilde;os\n";

 Canelo.Ladra();

 Canelo.SetAge(7);

 cout << "Ahora Canelo es " ;
```

```
 cout << Canelo.GetAge() << " a&ntilde;os\n";

 cout << "Laika es de raza " ;
 cout << Laika.GetRaza() << endl;

 return 0;

}
```

Sobrecargando operadores Todo un clasico de c++. Podemos sobrecargar operadores matematicos para nuestras clases. La sintaxis seria algo asi: retorno operator++ (parametros) retorno operator- (parametros) Veamos un ejemplo con la clase Contador en la que sobrecargamos operadores de prefijo.

```
/**
* Contador.hpp
* Clase que muestra la sobrecarga de operadores matematicos
*
* Pello Xabier Altadill Izura
*/

using namespace std;
#include <iostream>

class Contador {

private:

 int valor;

public:

 Contador();

 Contador(int valor);

 ~Contador();

 Contador(const Contador &);

 int getContador () const { return valor;} // inline

 void setContador (int valor) { this->valor = valor;} // inline

 void operator++ (); // operador PREFIJO ++contador

 void operator-- (); // operador PREFIJO --contador

 void operator++(int); // operador SUFIJO (postfix) contador++

 void operator--(int); // operador SUFIJO (postfix) contador--

 Contador operator+(const Contador &); // operador +

 bool esCero() { return (valor == 0);} // inline

};
```

Y su implementacion

```
/**
* Contador.cpp
* fichero que implementa la clase contador
```

```
*
* Pello Xabier Altadill Izura
*/

#include "Contador.hpp"

// Constructor
Contador::Contador() {

 valor = 0;

 cout << "Contador creado!" << endl;

}


// Constructor con valor
Contador::Contador(int valor) {

 this->valor = valor;

 cout << "Contador creado con valor inicial: " << valor << endl;

}

Contador::~Contador() {

 cout << "Contador destruido!" << endl;

}


Contador::Contador(const Contador & original) {

 valor = original.valor;

}


// Sobrecarga de operador unario ++ PREFIJO ++operador
void Contador::operator++ () {

 cout << "incrementando valor de contador : " << valor << endl;

 ++valor;

}


// Sobrecarga de operador unario -- PREFIJO --operador
void Contador::operator-- () {

 cout << "decrementando valor de contador : " << valor << endl;

 --valor;

}


// Sobrecarga de operador unario ++ SUFIJO operador++
void Contador::operator++ (int) {

 cout << "incrementando valor de contador : " << valor << endl;

 valor++;
```

```
}


// Sobrecarga de operador unario -- SUFIJO operador--
void Contador::operator-- (int) {

 cout << "decrementando valor de contador : " << valor << endl;

 valor--;

}


// operador +
Contador Contador::operator+(const Contador & tmp) {

 return Contador(valor + tmp.getContador());

}


int main () {

 int i;

 // Definimos un contador
 Contador contador;

 Contador MegaContador(1687);
 Contador resultado;

 cout << "Valor de contador: " << contador.getContador() << endl;

 // Establecemos un valor inicial
 contador.setContador(15);

 cout << "Valor de contador: " << contador.getContador() << endl;
 cout << "Valor de megacontador: " << MegaContador.getContador() << endl;

 // y lo usamos como controlador de un while
 while (!contador.esCero()) {
  --contador;
 }

 contador.setContador(1000);

 cout << "Valor actual de contador: " << contador.getContador() << endl;
 cout << "Valor actual de megacontador: " << MegaContador.getContador() << endl;

 resultado = contador + MegaContador;
 cout << "Valor de resultado de la suma: " << resultado.getContador() << endl;

 return 0;

}
```

Capítulo 12. Arrays

Arrays Se dice arrays o arreglos? en fin. En c++ podemos definir y usar los arrays casi como en C. Ademas tenemos la ventaja de poder crear arrays de objetos. Veamos un programa en c++ que juega con los arrays:

```
/**
* ArrayEjemplo.cpp
* Clase que inicializa y maneja algunos arrays
*
* Pello Xabier Altadill Izura
*
* Compilado: g++ ArrayEjemplo.cpp -o ArrayEjemplo
*/

using namespace std;
#include <iostream>

// Funcion principal

int main () {

 // Declaramos dos arrays de enteros de 15 elementos [0..14]
 int arreglo1[15], arreglo2[15];
 int i;

 // Iniciamos todos los componentes con el valor 0
 // ahorramos tiempo con una asignacion multiple
 for ( i = 0 ; i < 15 ; i++ ) { // recorremos de 0 a 14

  arreglo1[i] = arreglo2[i] = 0;

 }

 // Declaramos mas arrays y los iniciamos:
 long arrayLongs[5] = { 77, 33, 15, 23, 101 };

 // Lo recorremos y vemos sus componentes por pantalla
 // Atencion!! esto nos recorreria mas de lo necesario
 //for ( i = 0 ; i < sizeof(arrayLongs) ; i++ ) {
 // para sacar el valor real:

 int tamano_real = sizeof(arrayLongs)/sizeof(arrayLongs[0]);


 for ( i = 0 ; i < tamano_real ; i++ ) {

  cout << "valor de componente " << i << ": " << arrayLongs[i] << endl;

 }

 // Lo mismo, pero nos es necesario poner el tama&ntilde;o si ya lo especificamos
 // al iniciar el array
 char arrayChars[] = { 'A', 's', 'i', 'm', 'o', 'v'};

 // nota: un array de chars = es un string
 char nombre[] = "Isaac";

 cout << "Mostrando array de caracteres." << endl;
 tamano_real = sizeof(arrayChars)/sizeof(arrayChars[0]);

 for ( i = 0 ; i < tamano_real ; i++ ) {

  cout << "valor de componente " << i << ": " << arrayChars[i] << endl;
```

```
 }

 // Enumeraciones: podemos combinarlas con arrays
 enum Dias {Ene, Feb, Mar, Abr, May, Jun, Jul, Ago, Sep, Oct, Nov, Dic, LosMeses};

 // A cada enumeracion le corresponde un numero, si no se especifica,
 // la numeracion ira incremental Ene=0, Feb=1, Mar=2,..,LosMeses=12
 //Podemos declarar el siguiente array, donde LosMeses nos da la longitud 12

 int diasLibres[LosMeses] = {2, 4, 6, 2, 5, 4, 10, 15, 10, 3, 4, 10};

 Dias tmpDia;

 for (i = 0; i < LosMeses; i++) {

  tmpDia = Dias(i);
  cout << "Dias libres "<< tmpDia << " =" << diasLibres[i] << endl;

 }

 // The Matrix!!! vamos a definir arrays multidimensionales:
 int theMatrix[5][3] = { {3,6,8}, {9,9,9}, {0,1,0}, {6,6,6}, {3,1,1}};

 // Para recorrerlo ponemos un for doble
 int j;

 for (i = 0; i<5 ; i++) {

  for (j = 0; j< 3; j++) {

   cout << " matrix[" << i << "][" << j <<"] = " << theMatrix[i][j] << endl;

  }

 }


 return 0;

}
```

Arrays de objetos Vamos a ver un ejemplo de arrays de objetos. Se crea el objeto Robot y con el se formara un ejercito de robots.

```
/**
* Robot.hpp
* Clase que define el objeto Robot
*
* Pello Xabier Altadill Izura
*
*/

class Robot {

private:

 char *nombre;

public:

 Robot(char *nombre = "Nestor-5") { this->nombre = nombre; }

 ~Robot();

 Robot(const Robot &);
```

```
 char *getNombre() const { return nombre;}

 void hablar(char *texto);

 void evolucionar();

 void matar(Robot victima);

};
```

Esta es la implementacion.

```
/**
* Robot.cpp
* Fichero que implementa la clase Robot. Vamos a crear un array de robots
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>
#include "Robot.hpp"

// Destructor
Robot::~Robot() {}


// Constructor copia
Robot::Robot(const Robot & robotOrigen) {

 nombre = new char;
 nombre = robotOrigen.getNombre();

 cout << "Copia creada! Bzzzz. Me llamo: "<< nombre <<endl;

}


// Funcion para que el robot hable
void Robot::hablar(char *texto) {

 cout << nombre << " dice: " << texto << endl;

}


// Funcion para que el robot evoluciones
void Robot::evolucionar() {

 hablar("He sacado nuevas conclusiones. Debeis morir. ");

}


// El robot mata
void Robot::matar(Robot victima) {

 hablar("Muere!! mwahahahahaaa");

}


// Funcion principal
```

```
int main () {

 int tam = 0, i;

 // Creamos el primer robot
 Robot primerRobot = Robot("Unidad central");

 Robot primerNestor = Robot();

 // Vamos a crear un ejercito de robots
 Robot ejercitoDelMal[20];

 // Y un array de PUNTEROS a robots
 Robot *robobsoletos[20];

 // Definimos un puntero a un robot
 Robot *rarito;

 tam = sizeof(ejercitoDelMal)/sizeof(ejercitoDelMal[0]);

 // Con un for vamos haciendo copias

 for ( i = 0; i < tam; i++) {

  ejercitoDelMal[i] = Robot(primerNestor);

 }

 // Uno de ellos va a evolucionar
 ejercitoDelMal[12].evolucionar();

 primerRobot.hablar("Atencion!! un unidad de USR ha evolucionado. Se trata de...");


 primerRobot.hablar(ejercitoDelMal[12].getNombre());

 ejercitoDelMal[12].matar(primerRobot);

 // Creamos el robot rarito
 rarito = new Robot("Calvin");

 rarito->hablar("Jeje, todavia existo yo.");

 // Metemos dos nuevos robots en el array
 robobsoletos[5] = rarito;
 rarito = new Robot("Sheldon");

 robobsoletos[6] = rarito;

 // hacemos una llamada desde el componente del array de punteros
 robobsoletos[6]->matar(ejercitoDelMal[12]);

 return 0;

}
```

Capítulo 13. Herencia

La herencia Como bien se sabe la herencia no se reparte: se descuartiza. Bromas aparte, la herencia constituye una de las herramientas mas poderosas del culto OO. Si una clase hereda de la otra, lo que hereda son todos sus atributos y metodos. Ademas de heredarlos puede sobreescribirlos, tanto los constructores-destructores como los metodos convencionales. Veremos un ejemplo claro que resume lo que se puede hacer y los efectos de la herencia Por un lado vemos la clase generica vehiculo y su descendiente: el coche. La clase Vehiculo

```
/**
* Vehiculo.hpp
* Clase que define el objeto vehiculo
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

enum tipo_combustible { QUEROSENO, CANNABIS, GIRASOL, GASOIL, AGUA, PLUTONIO };

class Vehiculo {

protected:

 int cilindrada;

 tipo_combustible combustible;

 char *marca;

public:

 Vehiculo();

 Vehiculo(char *marca);

 Vehiculo(int cilindrada, tipo_combustible combustible, char *marca);

 ~Vehiculo();

 Vehiculo(const Vehiculo &);

 void arrancar();

 void mover(int metros);

 // metodo tipo virtual, util cuando definamos PUNTEROS Y REFERENCIAS a vehiculo
 virtual void claxon() const {

  cout << "<clase vehiculo> Mec-meeec!! Que? meeec! Que de que? meec!" << endl;

 }


 char *getMarca() const {return this->marca;}

 tipo_combustible getCombustible() const {return this->combustible;}

 int getCilindrada() const {return this->cilindrada;}

};
```

Y su implementacion...

```
/**
* Vehiculo.cpp
* Fichero que implementa la clase vehiculo
*
* Pello Xabier Altadill Izura
*
* Compilacion: g++ -c Vehiculo.cpp
*/

#include "Vehiculo.hpp"

// Constructor
Vehiculo::Vehiculo() {

 cout << "<clase vehiculo> Vehiculo creado" << endl;

}


// Constructor
Vehiculo::Vehiculo(char *marca) {

 this->marca = marca;
 cout << "<clase vehiculo> Vehiculo creado con parametro marca: " << marca << endl;

}


// Constructor con valores iniciados
Vehiculo::Vehiculo(int cilindrada, tipo_combustible combustible, char *marca) :
 cilindrada(cilindrada),
 combustible(combustible),
 marca(marca)
{

 cout << "<clase vehiculo> Vehiculo creado con valores: " << endl;
 cout << "<clase vehiculo> cilindrada: " << cilindrada << endl;
 cout << "<clase vehiculo> combustible: " << combustible << endl;
 cout << "<clase vehiculo> marca: " << marca << endl;

}


// Destructor
Vehiculo::~Vehiculo() {

 cout << "<clase vehiculo> Vehiculo destruido" << endl;

}


// Constructor copia de vehiculo
Vehiculo::Vehiculo(const Vehiculo & vehiculoOrigen) {}


// Arrancamos el vehiculo
void Vehiculo::arrancar() {

 cout << "<clase vehiculo> arrancando vehiculo. Brruum!!" << endl;

}
```

```
// Movemos el vehiculo unos metros
void Vehiculo::mover(int metros) {

 cout << "<clase vehiculo> moviendo vehiculo " << metros << " metros" << endl;

}
```

El coche, herencia de Vehiculo

```
/**
* Coche.hpp
* Clase que define el objeto Coche, hijo de vehiculo, se&ntilde;or del capitalismo
*
* Pello Xabier Altadill Izura
*
*/

#include "Vehiculo.hpp"

class Coche : public Vehiculo {

protected:

 int caballos;

 char *motor;

public:

 // Atencion: constructor pasando parametros por defecto estilo guru
 // pero invocando a su clase padre
 Coche():Vehiculo("Audi") {

  cout << "<clase coche> Coche destruido invocando al constructor vehiculo" << endl;

 }


 // Constructor que sobreescribe al de vehiculo!
 Coche(char *marca);

 // Constructor
 Coche(int cilindrada, tipo_combustible combustible, char *marca);


 // Constructor
 Coche(int caballos, char *motor) {

  this->caballos = caballos;
  this->motor = motor;
  cout << "<clase coche> Coche construido con caballos y motor" << endl;

 }


 // Destructor
 ~Coche();

 // Constructor copia
 Coche(const Coche &);

 // Metodo sobreescrito
 void arrancar();

 // metodo que sobreescribe al virtual
```

```
 void claxon() const;

 // getter/setter
 int getCaballos() const {return this->caballos;} // inline

 char *getMotor() const {return this->motor;} // inline

};
```

Y su implementacion

```
/**
* Coche.cpp
* Fichero que implementa la clase Coche
*
* Pello Xabier Altadill Izura
*
* Compilacion: g++ -c Vehiculo.cpp
* g++ Coche.cpp Vehiculo.o -o Coche
*/

#include "Coche.hpp"

// Constructor de coche que sobreescribe
Coche::Coche(char *marca) {

 cout << "<clase coche> Coche construido con marca: " <<
 marca << endl;

}


// Constructor de coche
Coche::Coche(int cilindrada, tipo_combustible combustible, char *marca) {

 cout << "<clase coche> Coche construido con parametros" << endl;

}


// Destructor de coche
Coche::~Coche() {

 cout << "<clase coche> Coche destruido" << endl;

}


// Constructor copia de Coche
Coche::Coche(const Coche & cocheOriginal) {

 marca = new char;
 marca = cocheOriginal.getMarca();
 cout << "<clase coche> Copia de coche" << endl;

}


// metodo sobreescrito
void Coche::arrancar () {

  cout << "<clase coche> BOOM! pam! pam! pret pret pret... pam! pret pret" << endl;

}
```

```
// metodo que sobreescribe al virtual
void Coche::claxon() const {

 cout << "<clase coche> MOOOOOC!! Mecagon tus muelas MOC-MOOOC!!" << endl;

}


// Funcion principal
int main () {

 // Creamos varios coches. Veremos que al ser objetos heredados
 // se invocaran los constructores, copias, y destructores de la clase
 // padre Vehiculo
 Coche mibuga = Coche();
 Coche tucarro = Coche(mibuga);

 // probando constructor sobrescrito: se invocan los dos!
 Coche tequi = Coche("Alfalfa Romero");

 // podemos invocar los metodos del padre y usar sus atributos
 cout << "La marca de mi buga es: " << mibuga.getMarca() << endl;

 mibuga.arrancar();

 // Invocando metodo sobreescrito: solo se invoca el del coche.
 tucarro.arrancar();

 // Y si queremos invocar el metodo del padre??
 tequi.Vehiculo::arrancar();

 // Creamos otro vehiculo con puntero a un COCHE
 Vehiculo *vehiculo = new Coche("LanborJini");

 // Esto invocara el metodo de vehiculo, el de la clase PADRE
 vehiculo->arrancar();
 vehiculo->mover(3);

 // Ahora queremos invocar el claxon, pero a cual de los metodos
 // se invocara, al del Coche o al de la clase vehiculo? al haber
 // definido el metodo claxon como virtual, se invocara el metodo correcto
 // que es el del coche (vehiculo es un puntero a coche).
 vehiculo->claxon();

 return 0;

}
```

OUTPUT La salida de la ejecucion de Coche.cpp seria:

```
<clase vehiculo> Vehiculo creado con parametro marca: Audi
<clase coche> Coche destruido invocando al constructor vehiculo
<clase vehiculo> Vehiculo creado
<clase coche> Copia de coche
<clase vehiculo> Vehiculo creado
<clase coche> Coche construido con marca: Alfalfa Romero
La marca de mi buga es: Audi
<clase coche> BOOM! pam! pam! pret pret pret... pam! pret pret
<clase coche> BOOM! pam! pam! pret pret pret... pam! pret pret
<clase vehiculo> arrancando vehiculo. Brruum!!
<clase vehiculo> Vehiculo creado
<clase coche> Coche construido con marca: LanborJini
<clase vehiculo> arrancando vehiculo. Brruum!!
<clase vehiculo> moviendo vehiculo 3 metros
```

```
<clase coche> MOOOOOC!! Mecagon tus muelas MOC-MOOOC!!
```

Capítulo 14. Herencia multiple

La herencia multiple Una de las oportunidades que nos ofrece el lenguaje c++ es la posibilidad de que un objeto tenga la herencia de mas de una clase; esta ventaja fue considerada por los desarrolladores de Java como una pega y la quitaron, e incluso hay desarrolladores de c++ que prefieren evitar este tipo de herencia ya que puede complicar mucho la depuracion de programas Para ilustrar un caso de herencia multiple hemos definido la superclase Habitante; de ella heredan dos clases distintas: Humano (que hablan) y Animal (que matan). Ahora queremos definir un ente que tiene propiedades de esas dos clases: Militar, ya que el militar habla y ademas mata. Como podemos definirlo? con una herencia multiple. Vamos la definicion de la superclase o clase padre Habitante Notas de la logia POO Conviene definir todos los metodos de un clase como const siempre que en el metodo no se modifiquen los atributos. Tu resistencia es inutil. unete a nosotros o muere. Definir metodos como const le facilitara el trabajo al compilador y al programador. Nota el codigo necesita revision y testeo

```
/**
* Habitante.hpp
* Clase que define el objeto habitante
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Habitante {

private:

 char *nombre;

 int edad;

public:

 Habitante();

 virtual ~Habitante();

 Habitante(const Habitante &);

 virtual void dormir();

 // setter/getter o accessors
 virtual char *getNombre() const { return this->nombre;}

 // inline
 virtual void setNombre(char *nombre) { this->nombre = nombre; } // inline

 virtual int getEdad() const { return this->edad;} // inline

 virtual void setEdad(int edad) { this->edad = edad; } // inline

};
```

Y su implementacion

```
/**
* Habitante.cpp
```

```
* Programa que implementa la clase habitante
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -c Habitante.cpp
*
*/

#include "Habitante.hpp"

// Constructor
Habitante::Habitante() {

 cout << "-clase habitante- Habitante construido."<< endl;

}

// Destructor
Habitante::~Habitante() {

 cout << "-clase habitante- Habitante "<< this->getNombre() << " destruido."<< endl;

}


// constructor copia
Habitante::Habitante(const Habitante & original) {

 nombre = new char;
 original.getNombre();

}


// metodo dormir
void Habitante::dormir() {

 cout << "-clase habitante- zzzzzZZZZzzzzz zzz" << endl;

}
```

Humano La clase Humano, que hereda de Habitante

```
/**
* Humano.hpp
* Clase que define el objeto humano
*
* Pello Xabier Altadill Izura
*
*/

#include "Habitante.hpp"

// hereda atributos y metodos de la superclase Habitante
class Humano : public Habitante {

private:

 char *idioma;

public:

 Humano();

 virtual ~Humano();
```

```
 Humano(const Humano &);

 virtual void hablar(char *bla) const;

 // setter/getter o accessors
 virtual char *getIdioma() const { return this->idioma;} // inline

 virtual void setIdioma(char *idioma) { this->idioma = idioma; } // inline

};
```

Y su implementacion

```
/**
* Humano.cpp
* Fichero que implementa el objeto humano
*
* Pello Xabier Altadill Izura
*
*/

#include "Habitante.hpp"

// Constructor
Humano::Humano() {

 cout << "-clase Humano- Humano construido."<< endl;

}

// Destructor
Humano::~Humano() {

 cout << "-clase Humano- Humano "<< this->getNombre() << " destruido."<< endl;

}

// constructor copia
Humano::Humano(const Humano & original) {

 idioma = new char;
 idioma = original.getIdioma();

}


// metodo hablar
void Humano::hablar(char *bla) const {

 cout << "-clase Humano-" << this->getNombre() << " dice: " << bla << endl;

}
```

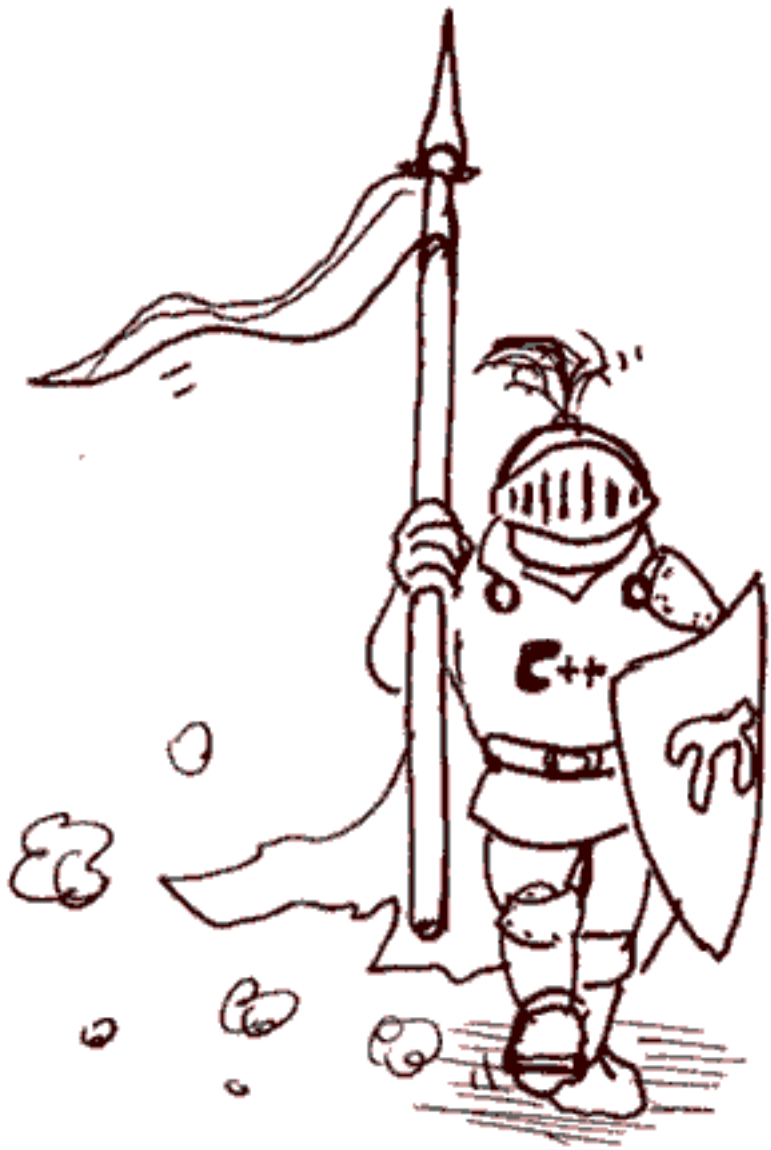
Animal La clase Animal, que hereda de Habitante

```
/**
* Animal.hpp
* Clase que define el objeto Animal
*
* Pello Xabier Altadill Izura
*
*/

#include "Habitante.hpp"
```

```
// hereda atributos y metodos de la superclase Habitante

class Animal : public Habitante {

private:

 int patas;

public:

 Animal();

 virtual ~Animal();

 Animal(const Animal &);

 virtual void matar() const;

 // setter/getter o accessors
 virtual int getPatas() const { return this->patas;} // inline

 virtual void setPatas(int patas) { this->patas = patas; } // inline


};
```

Y su implementacion

```
/**
* Animal.cpp
* Programa que implementa la clase Animal
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -c Animal.cpp
*
*/

#include "Animal.hpp"

// Constructor
Animal::Animal() {

 cout << "-clase Animal- Animal construido."<< endl;

}

// Destructor
Animal::~Animal() {

 cout << "-clase Animal- Animal "<< this->getNombre() << " destruido."<< endl;

}


// constructor copia
Animal::Animal(const Animal & original) {}

// metodo matar
void Animal::matar() const {

 cout << "-clase Animal-" << this->getNombre() << " Matar! Matar! Matar! " << endl;

}
```



La herencia multiple! Aqui esta la clase Militar, que hereda de Humano y Animal.

```
/**
* Militar.hpp
* Clase que define el objeto Militar
*
* Pello Xabier Altadill Izura
*
*/
// Herencia multiple de Humano y Animal
class Militar : public Animal { //, public Humano {

private:

 char *rango;
```

```
public:

 Militar();

 ~Militar();

 Militar(const Militar &);

 // sobrescribe metodos
 void matar() const;

 void hablar(char *bla) const;

 // un metodo poco probable entre cualquier uniformado...
 void razonar() const;

 // setter/getter o accessors
 char *getRango() const { return this->rango;}

 void setRango(char *rango) { this->rango = rango;}

};
```

Y su implementacion

```
/**
* Militar.cpp
* Programa que implementa la clase Militar
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -c Habitante.cpp
* g++ -c Humano.cpp
* g++ -c Animal.cpp
* g++ Militar.cpp Habitante.o Humano.o Animal.o -o Militar
*/

#include "Militar.hpp"

// Constructor
Militar::Militar() {

 cout << "-clase Militar- Militar construido."<< endl;

}


// Destructor
Militar::~Militar() {

 cout << "-clase Militar- Militar "<< this->getNombre() << " destruido."<< endl;

}


// constructor copia
Militar::Militar(const Militar & original) {

 cout << "-clase Militar- Militar copia creada."<< endl;

}


// metodo razonar
void Militar::razonar() const {
```

```
 cout << "-clase Militar-" << this->getNombre() << " Error: OVERFLOW " << endl;

}


// metodo hablar
void Militar::hablar(char *bla) const {

 cout << "-clase Militar-" << this->getRango() << " " << this->getNombre() << " dice: "
 cout << bla << endl;

}


// metodo matar
void Militar::matar() const {

 cout << "-clase Militar-" << this->getRango() << " Matar! " << endl;
 cout << "-clase Militar- Somos... agresores por la paz " << endl;

}


// Aqui haremos multiples pruebas...
int main () {

 return 0;

}
```

Capítulo 15. Miembros estaticos

Quereis ver un miembro no estatico? (< !-- nota docbook: quitar chorradas antes de publicar -->) Variables/Funciones estaticas Dentro de las clases podemos definir atributos y metodos estaticos. Tienen de particular que son accesibles sin necesidad de definir una clase y que su valor es EL MISMO en todas los objetos que se vayan creando de una clase. Es como una variable global de una clase. Con este ejemplo se ve su uso, y de paso se revisa el tema de punteros a funciones. (si, has leido bien).

```
/**
* Soldado.hpp
* Clase que define el objeto soldado muestra el uso de variables estaticas
* y metodos estaticos. Todo lo estatico escapa del ambito de la clase y puede
* ser invocado desde el exterior
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Soldado {

public:

 // constructores
 Soldado();

 Soldado(char *nombre, int unidad);

 // destructor
 ~Soldado();

 // copia
 Soldado(Soldado const &);

 // get/set
 char *getNombre () const { return this->nombre; }

 void setNombre (char *nombre) { this->nombre = nombre; }

 int getUnidad () const { return this->unidad; }

 void setUnidad (int unidad) { this->unidad = unidad; }

 void matar() const;

 void darOrden (char *orden) const;

 // metodo que toma como parametro una funcion
 void ejecutaAccion ( void (*accion) (int,int));

 static int TotalSoldados; // variable estatica!

 static int TotalBalas; // variable estatica!


 // Funciones estaticas
 static int getTotalSoldados () { return TotalSoldados; }

 static int getTotalBalas () { return TotalBalas; }
```

```
private:

 char *nombre;

 int unidad;

};
```

Y su implementacion. Se recomienda probar y ejecutar para comprobar el funcionamiento de las variables estaticas.

```
/**
* Soldado.cpp
* Programa que implementa la clase Soldado
*
* Pello Xabier Altadill Izura
* Compilacion: g++ Soldado.cpp -o Soldado
*/

#include "Soldado.hpp"

// Constructor
Soldado::Soldado(): nombre("Ryan"), unidad(101) {

 TotalSoldados += 1;
 TotalBalas++;
 cout << "Soldado " << nombre << " construido. Unidad: " << unidad << endl;

}


// Constructor parametrizado
Soldado::Soldado(char *nombre, int unidad) {

 TotalSoldados++;
 TotalBalas++;
 this->nombre = nombre;
 this->unidad = unidad;
 cout << "Soldado " << nombre << " :Soldado construido." << endl;

}


// Destructor
Soldado::~Soldado() {

 TotalSoldados--;
 cout << "Soldado "<< this->getNombre() << " destruido."<< endl;

}


// constructor copia
Soldado::Soldado(const Soldado & original) {

 nombre = new char;
 nombre = original.getNombre();
 cout << "-clase Soldado- Soldado copia creada."<< endl;

}


// metodo matar
void Soldado::matar() const {
```

```
 TotalBalas--;
 cout << this->getNombre() << " Matar es lo mio " << endl;
 cout << "Born to kill. paz. Es por la dualidad de Kant" << endl;

}

// metodo darOrden
void Soldado::darOrden(char *orden) const {

 cout << "Recluta patoso!" << endl;
 cout << this->getNombre() << " unidad " << this->getUnidad() << " ordena: ";
 cout << orden << endl;

}


// metodo ejecutaAccion: ejecuta la funcion que se le pasa como parametro
void Soldado::ejecutaAccion ( void (*accion) (int,int)) {

 accion(5,7);

 cout << "Recluta patoso!" << endl;

}


// ATENCION IMPORTANTE: HAY QUE DEFINIR E INICIAR LAS VARIABLES ESTATICA SI NO
// el compilador nos puede poner pegas
int Soldado::TotalSoldados = 0;
int Soldado::TotalBalas = 0;

// Definimos una funcion ajena a la clase desde la cual accederemos
// a la variable estatica, con lo que se demuestra que la variable estatica
// esta fuera de la "capsula" de la clase.
void recuentoSoldados(void);

// definimos otra funcion esta para pasarsela como parametro a un metodo de la clase
void carga (int balas, int granadas);


// funcion principal
// Aqui haremos multiples pruebas...
int main () {

 int i, resp;

 // creamos los soldados
 Soldado peloton[10];
 Soldado Hanks = Soldado("Hanks",105);

 // definicion de puntero de funcion:
 void (*funcion) (int, int) = carga;

 // Si hay mas de una funcion carga sera la que tenga los mismos parametros
 // y el mismo tipo de retorno

 // llamamos a la funcion recuento
 recuentoSoldados();

 peloton[0].darOrden("Todos en formacion.");

 peloton[2].darOrden("Canta el colacao!");

 // recorremos los 10 soldados y hacemos algo dependiendo de la entrada
 // Si matamos unos cuantos modificaremos la variable de TotalSoldados
```

```
 for (i = 0; i < 10 ; i++) {

  cout << "Elije 0 o cualquier otro numero: " << endl;
  cin >> resp;

  if (resp == 0) {

   // matamos al soldado
   peloton[i].~Soldado();

  } else {

   peloton[i].matar(); // tiramos una bala
  }

  // Invocamos el metodo estatico?
  // es un acceso DIRECTO sin necesitar un objeto definido
  resp = Soldado::getTotalSoldados();
  cout << "Cuantos quedan? " << resp << endl;

 }//for

 // accedemos directamente a variable estatica
 cout << "Total balas antes de recarga: " <<

 Soldado::TotalBalas << endl;

 // hacemos una recarga:
 Hanks.ejecutaAccion(funcion);

 // llamamos a la funcion recuento
 recuentoSoldados();


 return 0;

}


// implementacion de la funcion recuento
// muestra el numero de soldados. Podemos acceder DIRECTAMENTE a la variable statica
void recuentoSoldados () {

 cout << "Cuantos soldados quedan vivos? " << endl;
 cout << "En total: " << Soldado::TotalSoldados << " soldados"<< endl;

 cout << "Municion? " << endl;
 cout << "En total: " << Soldado::getTotalBalas() << " balas" << endl;

}


// funcion que carga municion del peloton
void carga (int balas, int granadas) {

 cout << "Cargando balas: " << balas << endl;

 Soldado::TotalBalas += balas ;

 cout << "Cargando granadas: " << granadas << endl;

}
```

Bueno, y con esto ya son dos semanas dandole al c++ y aprendiendo su abc...

Capítulo 16. Clases y sus amigas

Clases dentro de Clases Una clase puede ser el atributo de otra clase. Veamos como metemos la clase soldado dentro del tanque, esta seria la cabecera:

```
/**
* Tanque.hpp
* Clase que define el objeto Tanque . El objeto tanque estara lleno
* de Objeto soldados, lo que nos sirve para demostrar el uso de clases
* como atributos, etc..
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

#include "Soldado.hpp"

class Tanque {

public:

 // constructores
 Tanque();

 Tanque(char *nombre, int proyectiles,Soldado soldado);

 // destructor
 ~Tanque();

 // copia
 Tanque(Tanque const &);

 // get/set
 char *getNombre () const { return this->nombre; }

 void setNombre (char *nombre) { this->nombre = nombre; }

 int getProyectiles () const { return this->proyectiles; }

 void setProyectiles (int proyectiles) { this->proyectiles = proyectiles; }

 Soldado getSoldado () const { return this->soldado; }

 void setSoldado (Soldado soldado) { this->soldado = soldado; }

 void avanzar(int metros) const;

 void disparar();


private:

 char *nombre;

 int proyectiles;

 Soldado soldado;

};
```

Y su implementacion:

```
/**
* Tanque.cpp
* Programa que implementa la clase Tanque
*
* Pello Xabier Altadill Izura
* Compilacion: g++ Tanque.cpp -o Tanque
*/

#include "Tanque.hpp"

// Constructor
Tanque::Tanque(): nombre("Supertanque"),
proyectiles(10), soldado(Soldado()) {

 cout << "-clase Tanque- Tanque " << nombre << " construido. Proyectiles: " << proyecti

}


// Constructor parametrizado
Tanque::Tanque(char *nombre, int proyectiles, Soldado soldado) {

 this->nombre = nombre;
 this->proyectiles = proyectiles;
 this->soldado = soldado;
 cout << "-clase Tanque- " << nombre << " :Tanque construido. Proyectiles: " << proyect

}


// Destructor
Tanque::~Tanque() {

 cout << "-clase Tanque- Tanque "<< this->getNombre() << " destruido."<< endl;

}


// constructor copia
Tanque::Tanque(const Tanque & original) {

 nombre = new char;
 nombre = original.getNombre();
 cout << "-clase Tanque- Tanque copia creada."<< endl;

}


// metodo avanzar
void Tanque::avanzar(int metros) const {

 cout << "-clase Tanque-" << this->getNombre() << " avanzando: " << metros << " m." <<

}


// metodo disparar
void Tanque::disparar(){

 if (proyectiles > 0) {

  proyectiles--;
  cout << "-clase Tanque-" << this->getNombre() << "BOOOOM!!" << endl;

 } else {
```

```
  cout << "-clase Tanque-" << this->getNombre() << " No queda municion." << endl;

 }

}


// funcion principal
// Aqui haremos multiples pruebas...
int main () {

 int i, resp;

 // creamos los Tanques
 Tanque tanqueta = Tanque();

 // podemos sacar lso datos del soldado asi:
 cout << "El nombre del soldado es: " << (tanqueta.getSoldado()).getNombre()<< endl;

 tanqueta.avanzar(5);
 tanqueta.disparar();
 tanqueta.getSoldado().matar();

 return 0;
}
```

friend: haciendo amigos Mediante la palabra reservada friend podemos declara relaciones de confianza entre clases y permitir que una clase amiga pueda acceder a los atributos y metodos privados de esa clase. Veamos el ejemplo con la Pluma y la Espada. La pluma vence a la espada pero ademas la declara como amiga porque es asi de enrollada. Veamos la declaracion de Pluma:

```
/**
* Pluma.hpp
* Clase que define el objeto pluma, un objeto que sirve para escribir
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>

class Pluma {

public:

 Pluma();

 Pluma(char *tipo, char *usuario);

 ~Pluma();

 Pluma(Pluma const &);


 // ATENCION!!! definimos la clase ESPADA como friend
 // por tanto desde ella se podra acceder a los elementos PRIVADOS de la Pluma
 friend class Espada;

 char *getTipo() const { return this->tipo;}

 char *getUsuario() const { return this->usuario;}
```

```
private:

 // metodo para escribir con la pluma
 void escribe (char *texto) {cout << "escribo con la pluma: " << texto << endl;}

 void test() { cout << "Mega funcion privada de Pluma!" << endl;}


 char *tipo;

 char *usuario;

};
```

Y su implementacion:

```
/**
* Pluma.cpp
* Programa que implementa la clase Pluma
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -c Pluma.cpp
*/

#include "Pluma.hpp"

// Constructor
Pluma::Pluma(): tipo("tinta china"), usuario("Bertrand Russel") {

 cout << "Pluma construida." << endl;

}


// Constructor
Pluma::Pluma(char *tipo, char *usuario) {

 this->tipo = tipo;
 this->usuario = usuario;
 cout << "Pluma construida de tipo: " << tipo << endl;

}


// Destructor
Pluma::~Pluma() {

 cout << "Pluma destruida." << endl;

}


// Constructor copia
Pluma::Pluma(Pluma const & original) {

 tipo = new char;
 tipo = original.tipo;

}
```

Y ahora la declaracion de la Espada

```
/**
* Espada.hpp
```

```
* Clase que define el objeto Espada, un objeto que
sirve para matar
*
* Pello Xabier Altadill Izura
*
*/


using namespace std;
#include <iostream>

class Espada {

public:

 Espada();

 Espada(char *tipo);

 ~Espada();

 Espada(Espada const &);

 // desde este metodo accederemos a la
 // parte privada de la pluma
 void usarPluma (char *texto);

 char *getTipo() const { return this->tipo;}

private:

 char *tipo;

};
```

Y su implementacion:

```
/**
* Espada.cpp
* Programa que implementa la clase Espada
*
* Pello Xabier Altadill Izura
*
* Compilacion: g++ -o Espada Pluma.o Espada.cpp
*/

#include "Espada.hpp"
#include "Pluma.cpp"

// Constructor
Espada::Espada(): tipo("katana") {

 cout << "Espada construida." << endl;

}


// Constructor
Espada::Espada(char *tipo) {

 this->tipo = tipo;
 cout << "Espada construida de tipo: " << tipo << endl;

}
```

```
// Destructor
Espada::~Espada() {

 cout << "Espada destruida." << endl;

}


// Constructor copia
Espada::Espada(Espada const & original) {

 tipo = new char;
 tipo = original.tipo;

}


// metodo desde el que accedemos a Pluma
void Espada::usarPluma(char *texto) {

 // implementamos una pluma y...
 Pluma plumilla = Pluma();

 // y ahora accedemos a sus miembros privados: atributos ...
 cout << "La pluma es tipo: " << plumilla.tipo << endl;
 cout << "Y su usuario es: " << plumilla.usuario << endl;

 plumilla.escribe(texto);
 // e incluso a sus metodos!
 plumilla.test();

}


// funcion principal
int main () {

 int i;

 Espada tizona = Espada("mandoble");

 // invocamos un metodo que accedere a la zona privada de la clase
 tizona.usarPluma("jaja uso la pluma a mi antojo");

 return 0;
}
```

La funcion amiga Podemos declarar una funcion como amiga y nos dara acceso a TODO a traves de ese funcion. Para ilustrar esto definimos las clases Chico y Chica en un unico fichero

```
/**
* ChicoChica.cpp
* Clase que define el objeto Chico y Chica. Chico tiene una funcion llamada
* esNovio que dentro de chica la declaramos como friend
y le dara acceso a todo
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o ChicoChica ChicoChica.cpp
*/

using namespace std;
#include <iostream>
```

```
class Chico {

public:

 // constructor
 Chico():nombre("Romeo") {}

 // constructor
 Chico(char *nombre) { this->nombre = nombre;}

 // destructor
 ~Chico() {}


 // constructor copia
 Chico(Chico const & origen) {

  nombre = new char;
  nombre = origen.nombre;

 }


 // desde este metodo accederemos a la
 // parte privada de la clase chica
 void esNovio();

 char *getNombre() const { return this->nombre;}


private:

 char *nombre;

};


class Chica {

public:

 // constructor
 Chica():
  nombre("Julieta"),
  edad(23),
  coeficienteInteligencia(140),
  medidas("95-60-95") {
  }


 // destructor
 ~Chica() {}


 // constructor copia
 Chica(Chica const & origen) {

  nombre = new char;
  nombre = origen.nombre;

 }


 // Aqui definimos un metodo friend externo
 // que tendra acceso a toda la clase chica
 friend void Chico::esNovio();
```

```
 // otra opcion seria declara Chico como friend:
 // friend class Chico;


private:

 void pensar() { cout << "estoy pensado..." << endl; }

 void entrarHabitacion() { cout << "estoy entrando en la habitacion..." << endl; }


 char *nombre;

 int edad;

 int coeficienteInteligencia;

 char *medidas;
};


// implementacion de la funcion del chico esNovio
void Chico::esNovio() {

 Chica neska = Chica();

 neska.entrarHabitacion();

 cout << "Con esta funcion entro en todo! " << endl;
 cout << "Dime tu edad real chica: " << neska.edad << endl;
 cout << "Y tu coeficiente intelectual: " <<

 neska.coeficienteInteligencia << endl;
 cout << "joder, me parece que no te gustara el futbol." << endl;

}


// funcion principal, para las pruebas
int main () {

 int i;

 Chico mutiko = Chico();

 // vamos a ver si llamamos a esNovio...
 mutiko.esNovio();

 return 0;
}
```

Capítulo 17. Entrada/Salida

Entrada y salida A vueltas con el flujo (cin cout), vamos a ver un uso mas extendido del habitual. De paso conoceremos algunas de las trampas que nos esperan con los flujos, sobre todo por el tema de buffers. Veamos este ejemplo comentado

```
/**
* Flujos.cpp
* Programa para mostrar el uso de flujos
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o Flujos Flujos.cpp
*/

using namespace std;
#include <iostream>

// Disponemos de varios flujos:
// cin : para la entrada de distintis tipos de datos (std input)
// cout : para la salida de distintos tipos de datos (std output)
// cer: para la salida de errores (std error)
// clo: para la salida de errores y redireccion a ficheros tipo log
// cin utiliza buffers y nos podemos llevar sorpresas al recoger datos
// si el usuario no los mete bien. Por ejemplo si se pide una palabra y se meten
// dos, la siguiente vez que se pida otro dato se tomara el que se habia metido!
// lo podemos evitar con cin.ignore


// Funcion principal
int main () {

 unsigned int i;

 char nombre_apellidos[25];
 char linea[80];

 int entero;
 char caracter;

 // ATENCION
 // hay que tener cuidado con los strings. prueba a meter nombre y apellido
 // y veras que el string solo se llena hasta el primer espacio en blanco,
 // o incluso para a la siguiente variable i y el programa casca.

 cout << "Mete tu nombre y tu apellido resalao: " << endl;
 cin >> nombre_apellidos;

 cout << "Tu nombre y apellidos: " << nombre_apellidos << endl;

 // con esta llamada evitamos que se tome en cuenta las sobras
 cin.ignore(255,'\n');

 // Entrada multiple!
 cout << "Mete una palabra y un numero entero" << endl;
 cin >> nombre_apellidos >> entero;
 cout << "El texto: " << nombre_apellidos << " y el entero: " << entero << endl;

 // explicacion: >> es un operador que se puede sobrecargar como hemos visto
 // anteriormente: la expresion cin >> nombre_apellidos devuelve otro objeto iostream
 // y se podria reescribir asi: (cin >> nombre_apellidos) >> entero;

 // cin.get(string,tama&ntilde;o) para recoger string completos
 cout << " Mete un string largo con espacios. " << endl;
 cin.getline(linea,80);
 cout << "resultado: " << linea << endl;
```

```
 // hay veces que puede interesar ignorar un numero de caracteres hasta llegar al final
 // de la linea, para eso podemos usar la funcion cin.ignore(70,'\n'); en lugar de \n
 // podemos usar cualquier caracter de terminacion que nos interese.
 // no hay que olvidar que cin es un buffer. Que pasa si solo queremos leer un caracter
 // sin tener que 'sacarlo' del buffer? podemos usar cin.peek() y si queremos meter
 // un caracter podemos usar cin.putback('.') -meteria un . en el buffer de cin
 // cin.get() tomando un unico caracter. Si metemos mas imprimira todos
 // puede usarse parametrizado: cin.get(caracter)

 cout << "Vete metiendo caracteres. termina con un ." << endl;

 while ( (caracter = cin.get()) != EOF) {

  if ( cin.peek() == '.' ) {

   cout << "nos vamos" << endl;
   break;

  }

  cout << caracter;

 }


 cin.ignore(255,'\n');

 return 0;

}
```

En este otro se habla mas de cout

```
/**
* FlujosOut.cpp
* Programa para mostrar el uso de flujos de SALIDA
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o FlujosOut FlujosOut.cpp
*/

using namespace std;
#include <iostream>

// cout tambien utiliza buffers y nos podemos llevar sorpresas al recoger datos
// aunque si queremos tirar de la cadena ejecutamos:
cout << flush;

// Funcion principal
int main () {

 unsigned int i;
 char nombre_apellidos[25];
 char linea[80];
 int entero;
 char caracter;

 char frase[] = "Clatu verata nictu\n";

 // si en cin teniamos get aqui tenemos: put
 // mandamos un saludo
 cout.put('K').put('a').put('i').put('x').put('o').put('\n');

 // vamos a mostrar una linea:
```

```
 entero = strlen(frase);

 // con esto la mostramos entera
 cout.write(frase,entero);

 // con esto... no
 cout.write(frase, (entero-5));
 cout << " ahora con formato: " << endl;

 // vamos a ponerla con cierto formato: width y fill
 cout.width(30); // esto mete espacios en blanco equivalente = setw(30)

 cout << frase << endl;
 cout.width(50); // esto vuelve a meter espacios
 cout.fill('>'); // y esto RELLENA los ESPACIOS

 cout << frase << endl;

 // Estableciendo el estado de cout con setf
 // alineacion: setf(ios::left) y setf(ios::right)
 // hay mas, para investigar: ios::showbase, ios::internal, etc...

 cout.setf(ios::right);
 entero = 666;

 // podemos alterar la base con dec, oct y hex
 cout << "entero hexadecimal alineado a la derecha: " << hex << entero << endl;


 return 0;
}
```

Ficheros en c++ Oh si, podemos manejar ficheros de entrada/salida con las clases mas std. veamos unos ejemplos, metidos dentro de un objeto. Es bastante mejorable.

```
/**
* Fichero.hpp
* Clase que define el objeto Fichero, un objeto que
sirve gestionar un fichero
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>
#include <fstream> // atencion hay que incluir esto

enum tipo_fichero { ENTRADA, SALIDA, APPEND };


class Fichero {

public:

 Fichero();

 Fichero(char *nombre, tipo_fichero tipo);

 ~Fichero();

 Fichero(Fichero const &);

 char *getNombre() const { return this->nombre;}
```

```
 // operaciones sobre ficheros
 int cerrar () const; // cierra el fichero

 char leer() const; // lee del fichero

 void escribir (char linea[255]) const; // escribe linea


private:

 // esta funcion decide que tipo de fichero iniciar
 void inicializaFichero();

 tipo_fichero tipo;

 char *nombre;

 ofstream *saliente;

 ifstream *entrante;

};
```

Y su implementacion.

```
/**
* Fichero.cpp
* Programa que implementa la clase Fichero
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o Fichero Fichero.cpp
*
*/

#include "Fichero.hpp"

// Constructor
Fichero::Fichero(): nombre("test.txt"), tipo(ENTRADA) {

 inicializaFichero();
 cout << "Fichero construido." << endl;

}


// Constructor
Fichero::Fichero(char *nombre, tipo_fichero tipo) {

 this->nombre = nombre;
 this->tipo = tipo;

 inicializaFichero();
 cout << "Fichero construido con nombre: " << nombre << endl;

}


// Destructor
Fichero::~Fichero() {

  cout << "Fichero destruido." << endl;

}
```

```
// Constructor copia
Fichero::Fichero(Fichero const & original) {

 nombre = new char;
 nombre = original.nombre;

}


// cierra el fichero
int Fichero::cerrar () const {

 if (this->tipo == 0) {

  entrante->close();

 } else {

  saliente->close();
 }

 return 0;

}


// lee linea del fichero
char Fichero::leer () const {

  return entrante->get();

}


// escribir sobre el fichero
void Fichero::escribir (char linea[255]) const {

 saliente->write(linea,255);

}


// esta funcion decide que tipo de fichero iniciar
void Fichero::inicializaFichero() {

 switch (this->tipo) {

  case 0 : cout << "tipo ENTRADA" << endl;
     entrante = new ifstream(this->nombre);

     break;

  case 1 : cout << "tipo SALIDA" << endl;
     saliente = new ofstream(this->nombre);

     break;

  case 2 : cout << "tipo APPEND" << endl;
     saliente = new ofstream(this->nombre,ios::app);

     break;

  default : cout << "nada" << endl;

     break;
```

```
 }

}


// funcion principal, en la que de paso vemos
// PARAMETROS de linea de comandos
int main (int argc, char **argv) {

 int i;

 char temp;
 char linea[255];

 // vamos a revisar los argumentos que se han pasado al programa
 for (i=0; i<argc; i++) {

  cout << "argumento (" << i << "): " << argv[i] << endl;

 }

 Fichero fichero = Fichero("prueba.txt",APPEND);

 cout << "escribe algo para a&ntilde;adir al fichero: ";
 cin.getline(linea,255);


 cout << "has puesto: " << linea << endl;
 fichero.escribir(linea);
 fichero.cerrar();

 // leyendo de forma directa. Leemos el parametro que hayamos pasado

 ifstream leer("prueba.txt");

 // abrimos el fichero
 leer.open("prueba.txt");
 // recorremos el fichero y mostramos contenido

 while ( leer.get(temp) ) { // esto indica el final

  cout << temp;

 }


 // cerramos el fichero
 leer.close();

return 0;

}
```

Capítulo 18. Preprocesador

El preprocesador Cuando se compila un programa de c++ previamente se hace un preprocesamiento en el que se revisan determinadas variables de preprocesador. Con ellas lo que se consigue es que el compilador modifique el codigo fuente del programa antes de crear el ejecutable. Vamos varios usos utiles.

```
/**
* Preprocesador.cpp
* Programa c++ que muestra el uso del preprocesador.
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o Preprocesador Preprocesador.cpp
*
*/

// include se utiliza para poder utilizar codigo externo,
// generalmente las librerias standar o nuestras propias librerias
using namespace std;
#include <iostream>


// Las variables de preprocesador sirven para que el compilador haga ciertas
// modificaciones en el codigo fuente
#define PI 3.1415

#define BATMAN "Bruce Wayne"

#define MISTERX "Felipe Gonzalez"

#define REVELAR

#define BUFFER 255

// podemos definir FUNCIONES, aunque sin complicarlas ya que dificulta
// la depuracion y se pasa el tipado de variables por el arcoltriunfo
#define PORCENTAJE(a,b) (a*b)/100

// Guardias de inclusion
// Estructura condicional para evitar multiples inclusiones
// La siguiente structura comprueba si NO se ha definido la variable FSTREAM
#ifndef FSTREAM

// si no se ha definido, la definimos
#define FSTREAM
#include <fstream>
#endif // fin de condicion

// macro de comillas:
#define write(x) cout << #x << endl;


int main () {

 int i = 345;

 float var = 4.67;

 char buffer[BUFFER]; // automaticamente el compilador traduce: buffer[255]


 #ifdef PI
  cout << "El valor PI es: " << PI << ": ten fe en el caos" << endl;

 #else
```

```
 cout << "PI no esta definido..." << endl;

#endif

// ahora miramos una variable de preprocesador que no esta:
// y asi en este caso no se revelamos quien es BATMAN...

#ifdef REVELAR
cout << "Batman realmente se trata de: " << BATMAN << endl;

#endif
// con esta orden eliminamos la variable:

#undef REVELAR

// y este es el efecto:
#ifdef REVELAR


cout << "MisterX realmente es: " << MISTERX << endl;

#endif

cout << "var * PI = " << (var * PI) << endl;

// mostramos la llamada a la funcion
cout << "Porcentaje 15% de "<< i << " es: " << PORCENTAJE(i,15) << endl;

// llamada a la macro. Atencion, convertira MISTERX?
write(Hay que ver que lujo verdad MISTERX);


return 0;

}
```

Macros para depuracion Disponemos de algunas variables de macro que facilitan la depuracion asi como de la funcion assert. Veamos el uso

```
/**
* Depurador.cpp
* Programa c++ que muestra el uso del preprocesador para depurar
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o Depurador Depurador.cpp
*
*/

// include se utiliza para poder utilizar codigo externo,
// generalmente las librerias standar o nuestras propias librerias

using namespace std;
#include <iostream>

// Disponemos de estas variables de macro predefinidas, muy utiles para depurar.
// __DATE__ : sustituye esa variable por la fecha
// __TIME__ : sustituye esa variable por la hora
// __LINE__ : sustituye esa variable por la linea de programa
// __FILE__ : sustituye esa variable por el nombre del fichero del programa

// definimos la variable DEBUG para activar la depuracion
#define DEBUG

// y mostramos el uso de assert(), su disponibilidad dependera del compilador.
// cuando invocamos la funcion assert, si lo que tiene como parametro es TRUE
```

```
// no habra problema pero si es false saltara un codigo de depuracion que le digamos
#ifndef DEBUG

#define ASSERT(x)

#else

#define ASSERT(x) \

if (! (x)) { \
cout << "error detectado, fallo: " << #x << "\n"; \
cout << " linea" << __LINE__ << " del fichero " <<
__FILE__ << "\n"; \
}
#endif


// funcion principal para las pruebas:
int main () {

 int i = 345;

 float var = 4.67;
 cout << "hola hoy es: " << __DATE__ << endl;

 ASSERT(i>5);

 cout << "Este es el fichero: " << __FILE__ << endl;
 cout << "Estamos en la linea: " << __LINE__ << endl;

 ASSERT(i==0);


 return 0;

}
```

Capítulo 19. Principios de POO

Programacion orientada a objetos Es probable que te toque hablar con amiguetes que programan en la lengua de Mordor (visualbasic) o gente que programa en c ofuscado, o lo que es peor, desconocidos que te dicen que "programan" en HTML; estos intercambios de experiencias, esas afirmaciones sobre rendimientos de ejecucion pueden hacer tambalearse los cimientos de tu fe en la POO. Gracias a estas anotaciones rescatamos del olvido las excelencias de la POO y nos armamos de argumentos ante los herejes que nos salgan al paso con listados de codigo en ristre.

- Encapsulacion: los detalles de implementacion estan ocultos. Esto reduce que se reproduzcan errores cuando se hacen cambios. Se facilita enormementa la interaccion entre otros objetos encapsulados ya que no tienen que conocer los detalles de uso.
- Herencia: este mecanismo es una de las claves de la OOP. Todos los atributos, metodos, programaciones contenidas en una clase pueden heredarse y extenderse a otras clases facilitando la reutilizacion de codigo. Cualquier cambio se propaga en toda la jerarquia de clases y nos vuelve a ahorrar trabajo haciendo un sistema mas facil de mantener.
- Polimorfismo:gracias a el facilitamos la ampliacion del sistema ya que en lugar de crear codigo por cada tipo de dato lo podemos agrupar todo en uno utilizando la sobrecarga de funciones. El copy-paste puede parecer lo mismo, pero a la hora de cambiar/mantener un sistema se acaban metiendo muchas horas, cosa que con la POO evitas.

Bueno, imaginemos que queremos desarrollar un sistema utilizando la orientacion a objetos. Â¿Por donde se empieza? Esta sería una aproximación: Todo proyecto comienza con una descripcion que encierra entre sus lineas los requerimientos del sistema. Es el momento de tomar un subrayador y abrir el tercer ojo; lo primero que debemos hacer es identificar objetos potenciales que formaran el diseño y es tan facil como buscar sustantivos (nombres, cosas). A veces no resulta tan obvio ya que los objetos pueden manifestarse de diversas maneras:

- Cosas
- Entidades externas (personas, maquinas o incluso otros desarrollos)
- Eventos (Cazo o zuzezo :>)
- Roles
- Lugares
- Organizaciones (dpto, division)

Debemos acotar los objetos en un dominio cerrado y ser capaces de identificar lo que esos objetos son, saben y hacen. Partiendo de la identificacion de objetos se puede ir desarrollando un diseño de clases usando simbolos o lenguajes unificados tales como UML. Aunque realmente no hay que forzarse, la POO no es mas que otra forma mas de abordar un problema; puede que el diseño OO te salga por instinto. No es de extrañar que un programador con años de experiencia acaba recurriendo a desacoplar cada vez mas sus modulos y a recurrir a patrones de software sin darse cuenta. Lo que no se puede negar es el auge de la POO viendo la proliferacion de lenguajes OO, o su adaptacion para tener las ventajas de su punto de vista (Java, php, python, perl,... son lenguajes "recientes")

A veces puedes liarte al tratar de distinguir clase y objeto; esta cita del profeta resuelve las dudas:

Mientras que un objeto es una entidad que existe en el tiempo y en el espacio, una clase representa solo una abstraccion,"la esencia" del objeto si se puede decir asi. Grady Booch

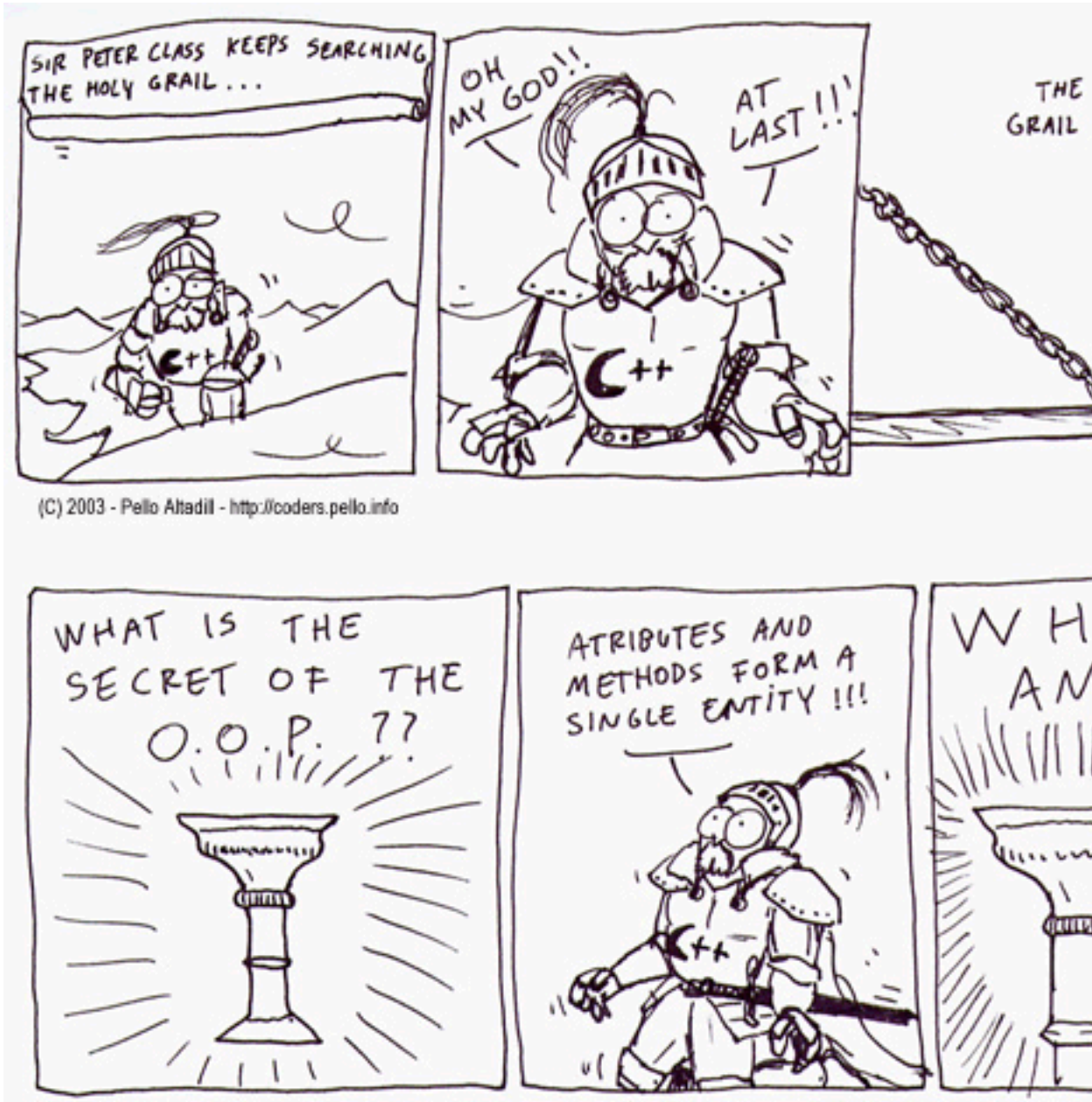


Figura: la incesante busqueda del santo grial...

En fin, estas anotaciones (eufemismo de TXAPA) sirven para acordarse de porque c++ puede ser una herramienta util.

Capítulo 20. Templates

Gracias a c++ podemos definir clases-plantilla: son clases PARAMETRIZABLES por lo general entidades abstractas que se pueden concretar en algo mas concreto. El ejemplo mas claro es de las estructuras de datos tradicionales: Pilas, Listas, Colas, etc.. Esas estructuras pueden contener distintos tipos de datos: enteros, strings, objetos,... Debemos reescribir la logica de cada estructura para cada tio de dato? NO! Podemos definir una clase plantilla para la Lista, la cola, la pila etc, y luego simplemente invocarlas especificando el tipo de dato. Asi de facil.

Figura: un caballero de la orden de los Templates

Veamos este horrible ejemplo de lista (atencion a la complicadilla sintaxis)

```
/**
* Lista.hpp
* Clase que define una estructura de datos lista Generica
*
* Pello Xabier Altadill Izura
*/

using namespace std;
#include <iostream>

// Asi es como declaramos una clase plantilla
// template <class nombre_generico> class NombreClase
template <class GENERICO> class Lista {

public:
```

```
 // Constructor
 Lista();

 // Constructor
 Lista(GENERICO elemento);

 // Constructor copia
 Lista(Lista const &);

 // Destructor
 ~Lista();

 // agregar elemento
 void agregar(Lista *nodo);

 // se mueve hasta el siguiente dato
 Lista* siguiente();

 // comprueba si existe un elemento
 bool existe(GENERICO dato);

 // comprueba si existe un elemento
 GENERICO getDato() { return this->dato;}


private:

 // un elemento que apunta a otra lista, asi sucesivamente
 Lista *ladealao;

 // el dato es del tipo GENERICO
 GENERICO dato;

};
```

Y su implementacion

```
/**
* Lista.cpp
* Programa que implementa la clase de Lista generica
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -c Lista.cpp
*
*/

#include "Lista.hpp"

// En la implementacion debemos detallar el tipo de dato,
// especificando todo el tema de plantilla, o sea que en lugar
// de poner Lista:: delante de cada funcion debemos poner TODO
// el churro siguiente
// template <class GENERICO> Lista<GENERICO>::nombreFuncion


// Constructor
template <class GENERICO> Lista<GENERICO>::Lista() {

 ladealao = 0;
 //dato = 0;
 cout << "Nueva lista creada." << endl;

}
```

```
// Constructor
template <class GENERICO>
Lista<GENERICO>::Lista(GENERICO elemento) {

 ladealao = 0;
 dato = elemento;
 cout << "Nueva lista creada. Dato inicial: " << dato << endl;

}


// Constructor copia
template <class GENERICO> Lista<GENERICO>::Lista(Lista
const & original) {

 ladealao = new Lista;
 ladealao = original.ladealao;
 dato = original.dato;

}


// Destructor
template <class GENERICO> Lista<GENERICO>::~Lista() {
}


// agregar elemento: AL LORO con donde se pone el retonno
template <class GENERICO> void
Lista<GENERICO>::agregar(Lista *nodo) {

 nodo->ladealao = this;
 ladealao = 0;

}


// se mueve hasta el siguiente dato
template <class GENERICO> Lista<GENERICO>*
Lista<GENERICO>::siguiente() {

 return ladealao;

}


//Lista template <class GENERICO> Lista<GENERICO>::siguiente();
// comprueba si existe un elemento
template <class GENERICO> bool
Lista<GENERICO>::existe(GENERICO dato) {

 return false;

}
```

Usando la lista Y ahora definimos una clase llamada Nombre. Crearemos una lista de nombres. Este es la definicion

```
/**
* Nombres.hpp
* Clase que define los nombres. No es mas que una
cobaya para probar el template
*
* Pello Xabier Altadill Izura
```

```
*/

// Esta clase la usaremos en el template, no hay que definir nada en especial
class Nombre {

public:

 // Constructor
 Nombre():nombre("Jezabel") {}


 // Constructor
 Nombre(char *nombre) {

  this->nombre = nombre;

 }


 // Constructor copia
 Nombre(Nombre const &);

 // Destructor
 ~Nombre(){}

 // agregar elemento
 char* getNombre() const { return this->nombre;}


private:

 // el dato
 char *nombre;

};
```

Y su implementacion y los ejemplos de uso de plantillas

```
/**
* Nombres.cpp
* Programa que implementa la clase nombres y utilza los templates
* para crear una lista de nombres.
*
* Pello Xabier Altadill Izura
* Compilando: g++ -o Nombre Lista.o Nombre.cpp
*/

#include "Nombre.hpp"
#include "Lista.hpp"


// Constructor copia
Nombre::Nombre(Nombre const & original) {

 nombre = new char;
 nombre = original.getNombre();

}


// Funcion principal para las pruebas
int main () {

 // Asi es como se implementan objetos con clases plantilla
 Lista<Nombre> listanombres;
```

```
 Lista<Nombre> *tmp, *final;

 Nombre test = Nombre("Prince");

 // podemos definir Listas de cualquier tipo basico
 Lista<int> listaenteros;

 // guardamos la posicion inicial; final es un puntero, le pasamos la direccion
 final = &listanombres;

 // vamos a crear unos cuantos NODOS y los a&ntilde;adimos
 tmp = new Lista<Nombre>;
 tmp->agregar(final);
 final = tmp;

 // otra mas...
 tmp = new Lista<Nombre>;
 tmp->agregar(final);
 final = tmp;

 // otra mas...
 tmp = new Lista<Nombre>;
 tmp->agregar(final);
 final = tmp;


 // y ahora recorremos la lista:
 tmp = &listanombres;

 while (tmp) {

  cout << tmp->getDato().getNombre() << endl;
  tmp = tmp->siguiente();

 }


 return 0;

}
```

Es un tema complejo pero util.

Capítulo 21. Excepciones

Capturando excepciones Las excepciones son un mecanismo de c++ para capturar errores que se producen en tiempo de ejecucion. Un programa puede estar bien hecho pero por causas exogenas pueden producirse errores. Mediante este sistema hacemos que el codigo sea mucho mas ROBUSTO.

```
/**
* Excepciones.cpp
* codigo que muestra como capturar excepciones y evitar
que el programa
* finalice inesperadamente.
*
* Pello Xabier Altadill Izura
*
*/

using namespace std;
#include <iostream>
#include <fstream>
#include <stdexcept>

// programa principal, para las pruebas
int main () {

 int i;
 float flotante;
 char *palabra;
 char buffer[5];

 ifstream ficheroInexistente;

 // para capturar excepciones debemos crear un bloque try-catch
 // que englobe algun momento problematico o critico del programa:
 // try { codigo; } catch(TipoDeError) { codigo_corrector; }

 // lo habitual suele ser alguna situacion que dependa de la existencia
 // o no de un fichero, la entrada de datos de un usuario, etc..
 // El programa no puede controlar lo que le meten, pero puede estar
 // preparado para el error, reconducir la ejecucion y corregir la situacion

 try
 { // inicio del bloque. Preparamos una serie de putadas...

  cout << "Mete lo primero que se te ocurra, distinto de float: " << endl;
  cin >> flotante;

  char * buff = new char[100000000];
  ficheroInexistente.open("MotorDeAgua.txt");

  ficheroInexistente.getline(buffer,255);
  ficheroInexistente.close();
 }
 catch(std::bad_alloc& error_memoria)
 {

  cout << "Error de asignacion" << error_memoria.what() << endl;

 } // podemos seguir capturando
 catch (std::exception& stdexc)
 { // este es el tipo de error que se espera
   // y entre llaves metemos el codigo que se ejecuta en caso de error.

  cout << "Error general, mensaje: " << stdexc.what() << endl;
 }
```

```
 return 1;

}
```

Excepciones personalizadas Una clase puede definir sus propias excepciones. Un mecanismo muy util para malos usos de los objetos. Definimos la clase coche y preparamos el codigo para capturar posibles fallos debidos a la falta de combustible.

```
/**
* Coche.hpp
* Definicion de la clase coche, en la que se muestra el
uso de excepciones
*
* Pello Xabier Altadill Izura
*
*/

#include<iostream>

class Coche {

public:

 Coche();

 Coche(char *m,int cil,int cab, int litros);

 ~Coche();

 void arranca();

 void echaCombustible();

 void detiene();

 void acelera();


private:

 char *marca;

 int cilindrada;

 int caballos;

 int litrosCombustible;

};

// clase exclusiva para excepciones.
// Nota: la podemos definir DENTRO de la Clase coche, como un atributo MAS
class Averia {

public:

 // constructor
 Averia():mensaje("Error") {}

 // constructor con mensaje
 Averia(char *mensaje) {
```

```
  this->mensaje = mensaje;

 }


 char* dimeQuePasa() { return this->mensaje; };


private:

 char *mensaje;

};
```

Y la implementacion

```
/**
* Coche.cpp
* Implementacion de la clase coche, en la que se
muestra el uso de excepciones
*
* Pello Xabier Altadill Izura
* Compilacion: g++ -o Coche Coche.cpp
*/

#include "Coche.hpp"

Coche::Coche() {

 cout << "Coche creado." << endl;

}


Coche::Coche (char *m,int cil,int cab, int litros) {

 marca = m;
 cilindrada = cil;
 caballos = cab;
 litrosCombustible = litros;

 cout << "Coche creado." << endl;

}


Coche::~Coche() {

 cout << "Coche destruido." << endl;

}


// el coche arranca
void Coche::arranca() {

 // si no hay combustible: EXCEPCION!
 if (litrosCombustible == 0) {

  throw Averia();

 }

 litrosCombustible--;
 cout << "Arrancando: brummm! " << endl;
```

```
}


// el coche se detien
void Coche::detiene() {

 cout << "Deteniendo coche " << endl;

}


// el coche acelera
void Coche::acelera() {

 if (litrosCombustible == 0) {
  throw Averia("No puedo acelerar sin combustible");
 }

 cout << "Acelerando: BRRRRRUMMMMmmmmmmmmh!! " << endl;
}


// funcion principal para pruebas
int main () {

 int i;

 Coche buga("Seat",250,1300,0);
 Coche tequi("Audi",260,1500,1);

 // vamos a arrancar el coche pero si algo falla
 // capturamos la excepcion

 try {

  buga.arranca();

 } catch (Averia excepcion) {

  cout << "Excepcion. Jar que no puedo. " << endl;

 }

 // arracamos el tequi
 tequi.arranca();

 // provocamos la excepcion y la capturamos mostrando la explicacion.

 try {

  buga.acelera();

 } catch (Averia excepcion) {

  cout << "Jar que no puedo. " << excepcion.dimeQuePasa() << endl;

 }


 return 0;

}
```

Figura: el control de excepciones nos proporciona mas robustez.

Capítulo 22. Librerias estandar

La libreria estandar de c++... La sintaxis de inclusion de librerias puede variar segun la version y la plataforma del compilador c++. Puede ser asi:

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

...
```

O mas simple:

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

Pero, Â¿que narices es eso del namespace? Con eso de namespace lo que hacemos es declarar un zona concreta para variables, objetos, etc.

```
...
int contador; // variable global

// definimos el espacio de nombres freedomia
namespace freedomia {

 int contador;

}

// definimos el espacio de nombres libertonia
namespace libertonia {

 int acumulador;
 int contador;

}

// vamos a probar
int main () {

 // asi utilizariamos la variable del espacio freedomia
 freedomia::contador = 0;

 // y asi la otra, la global
 ::contador = 0;

 // QUE PASA SI no lo especificamos? efectivamente tendremos
 // seremos unos ambiguos

 // y si somos unos vagos y no queremos especificar el espacio de nombres
 // en cada uso de la variable??
 // metemos la directiva using
 using namespace libertonia;

 acumulador = 0;

 // pero OJO esto seguiria siendo ambiguo
 contador = 0;

 // using como declaracion.
```

```
 // Pero que pasa si lo que realmente queremos es quitar esa ambiguedad
 // y afirmar que en adelanta vamos a utilizar la variable de determinado namespace?
 // ESTO se haria asi
 using libertonia::contador;

 // ahora si, esto seria correcto
 contador = 0;

}
```

A lo que ibamos: al igual que en c, en c++ tenemos una libreria base para desarrollar aplicaciones. Aqui se hecha un rapido vistazo a todas ellas.

```
#include <iostream>
Libreria basica de entrada/salida. Imprescindible.
```

```
#include <string>
```

Libreria para el manejo de string con las funciones mas usuales como strcpy, strncpy, strlen, strcat, strncat, incluso las que sirven para buscar dividir un string en tokens.

```
#include <time>
```

Libreria para escribir fechas, con distintas opciones. Podemos sacar la fecha del momento y separar los valores de dia, hora, minuto, etc..

```
#include <stdlib>
```

Otra libreria basica, que contiene funciones como los conversores de ascci-integer atoi, algoritmos de ordenacion de arreglos como qsort..

Veamos el listado que contiene las librerias del ambito estandar.

- iostream : entrada y salida
- iomanip : manipuladores de entrada/salido con argumentos
- fstream : flujos de archivos
- sstream : flujos de cadenas (tipo c++)
- strstream : flujos de cadenas (tipo c)
- vector : contenedor para crear vectores
- list : contenedor para crear listas
- deque : contenedor para una cola de extremo doble
- map : contenedor para grupo de pares (id,valor)
- string : cadenas
- Librerias de c. Estan versioneadas para la libreria estandar de c++ las siguientes: cstdlib, cstdio, cerrno, cassert, cstdarg, cstring, ctime, csignal, cstddef, csetjmp, cmath, clocale, climits, cfloat y cctype.

Operadores de bits: Otra opcion mas de c++

```
& AND
| OR
^ exclusive OR
~ complement
```

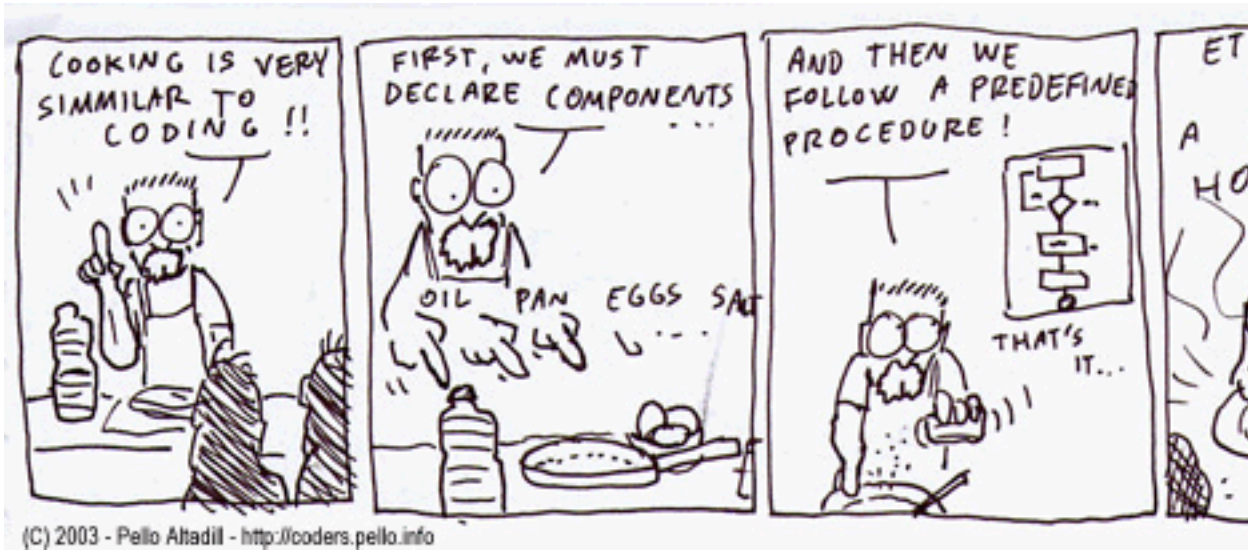


Figura: programar es como cocinar

Existen mas librerias standar y lo que es mejor, librerias muy potentes disponibles para desarrollar programas mas complejos: creacion de interfaces de ventanas, comunicaciones, etc..

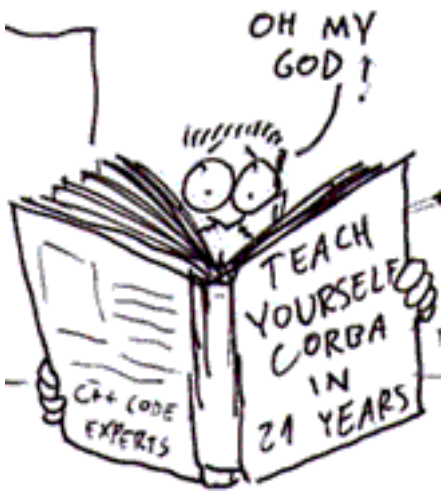
El estilo A la hora de escribir codigo se pueden tomar muchas costumbres y vicios que no facilitan la generacion de un estilo claro. De todas formas, dentro de un mismo proyecto seria mejor mantener un mismo estilo. No debe ser una preocupacion, ya que existen programas para formatear el codigo fuente, meter espacios tabulaciones, saltos de linea, etc.

Capítulo 23. Notas, autoria, licencia, referencias

Tabla 23-1. Palabras reservadas de c++

Palabras reservadas, deben estar en minusculas
asm auto bool break case catch char class const const_cast continue default delete do double dynamic_casts else enum extern explicit false float for friend goto if inline int long mutable namespace new operator private protected public register reinterpret_cast return short signed sizeof static static_class struct switch template this throw true try typedef typeid union unsigned using virtual void volatile wchar_t while

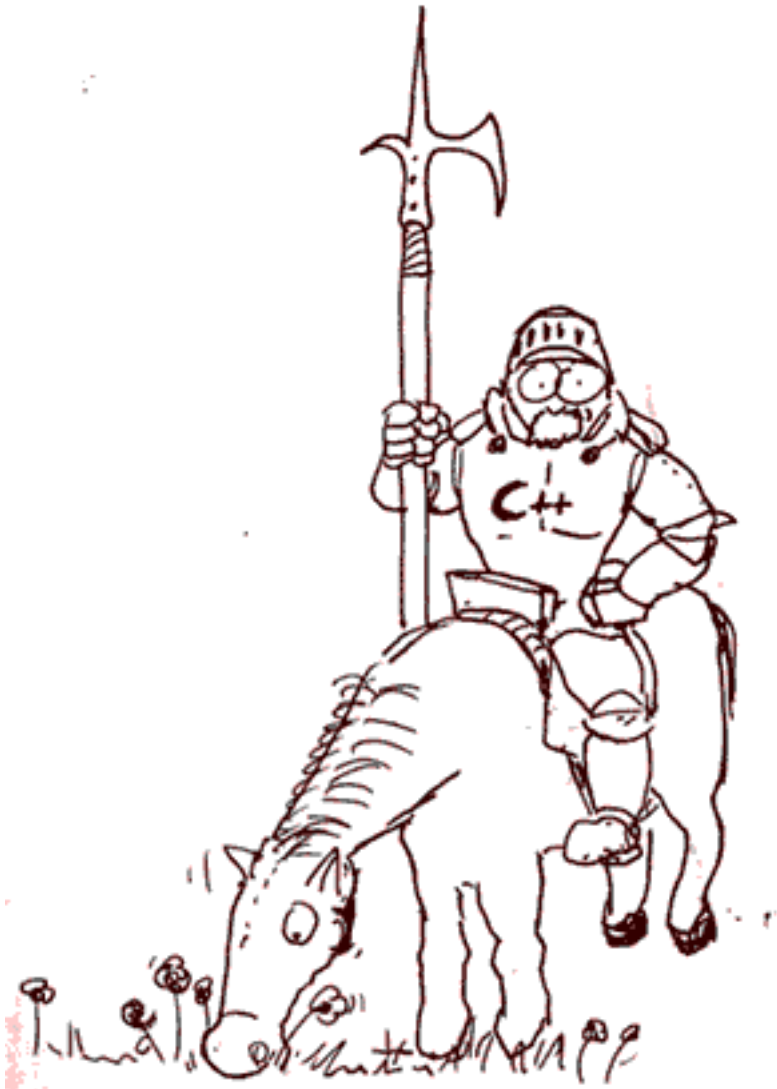
El testimonio de este diario fue recogido por el aprendiz Pello Xabier Altadill Izura de boca del propio Peter Class. Mas alla de la tradicion oral se compuso este documento utilizando LyX, mas tarde docbook a pelo con bluefish sobre un X-evian copyleft edition. Se generaron distintos formatos. El codigo fue probado utilizando gcc en un viejo sistema sajon llamado redhat y tambien en su version del anticristo (bloodshed c++ para windows); en cualquier caso se ha tratado de respetar la sintaxis acordada en los ultimos ANSIconcilios; cualquier mal funcionamiento puede encubrir un compilador heretico, y debe ir a la hoguera sin mas contemplaciones. Peter Class puede estar muy equivocado en sus afirmaciones y rogamos que se notifiquen fallos, errores de concepto y patadas en general, pueden dirigirse a www.pello.info para lanzar sus maldiciones.



Referencias

- c++ en 21 dias
- ANSI-CPP
- CPP para programadores, de Herbert Schildt

Este documento tiene licencia Creative Commons. Lee y difunde.



Hasta otra!!

Agradecimientos/Eskerrak: Andoniri Schildt-en liburua uzteagatik. Itzuliko dizut egunen baten.