

# Introducción a C++

David Pinelo

Marzo – Abril de 2009



# Índice

- Sintaxis
- Variables
- Funciones
- Operadores
- Clases
  - Constructores y destructores
- Herencia
- Sobrecarga
- Polimorfismo
- Funciones virtuales. Clases abstractas
- Namespaces
- Templates

# Requisitos para seguir el curso

- Se presumen conocimientos previos de programación
- Se debe estar familiarizado con el lenguaje C, y su sintaxis

# Origen

- Tiene su origen en el lenguaje C
- En 1980, Bjarne Stroustrup añade al lenguaje C características como las clases, chequeo del tipo de los argumentos de una función y su conversión si es posible etc, dando como resultado que en ese momento se llamó C con clases<sup>1</sup>
- Se desarrolló en la AT&T Bell Laboratories.
- En 1983 este lenguaje fue rediseñado y comenzó a utilizarse fuera de ATT. Se introdujeron las funciones virtuales, la sobrecarga de funciones y operadores. Y aparece C++

# Origen (II)

- Posteriormente C++ ha sido ampliamente revisado para añadir nuevas características como la herencia múltiple, las funciones miembro static y const, miembros protected, tipos de datos genéricos o plantillas, y la manipulación de excepciones. Se revisaron aspectos de la sobrecarga y manejo de memoria, se incremento la compatibilidad con C
- En 1989 se convocó el comité de la ANSI2 que más tarde entró a formar parte de la estandarización ISO. El trabajo conjunto de estos comités permitió publicar en 1998 el estándar ISO C++ (ISO/IEC 14882) de forma que el lenguaje pase a ser estable y el código generado utilizable por distintos compiladores y en distintas plataformas.

# Origen (III)

- Dentro de esta estandarización se incluye un conjunto de clases, algoritmos y plantillas que constituyen la librería estándar de C++3. Esta librería introduce facilidades para manejar las entradas y salidas del programa, para la gestión de listas, pilas, colas, vectores, para tareas de búsqueda y ordenación de elementos, para el manejo de operaciones matemáticas complejas, gestión de cadenas de caracteres, tipos de datos genéricos, etc.
- A este conjunto de algoritmos, contenedores y plantillas, se los denomina habitualmente por las siglas STL (Standard Template Library).

# Estructura de un programa

- Es una colección de definiciones y declaraciones
  - Definiciones de tipos de datos
  - Definiciones de datos globales
  - Definición de funciones
  - Definiciones de clases
  - Una función especial llamada `main()` que inicia la ejecución
- C++ permite programación por procedimientos o programación orientada a objetos
- Bjarne Stroustrup dice *“C++ is a multi-paradigm programming language that supports Object-Oriented and other useful styles of programming”*

# Primer programa

```
// Hello World program
```

Comentario

```
#include <iostream.h>
```

Permite el acceso a funciones  
De E/S

```
int main() {
```

Inicia el programa. Declara  
Una función

```
cout << "Hola Mundo\n";
```

Escribe en la salida estándar

```
return 0;
```

Devuelve al entorno de ejecución  
Un valor 0

```
}
```

# Primeras consideraciones

- **Comentarios:** Puede utilizarse
  - `//` para comentar una única línea
  - `/*` Para comentar un párrafo `*/`
- **Preprocesador**
  - Inclusión de cabeceras
    - `#include <stdlib>`
    - `#include <time.h>`
    - `#include "micabecera.hpp"`
  - Directiva `define`
    - `#define square(a) (a*a)`
  - Condicionales
    - `#ifdef`
    - `#endif`

# Algunos includes básicos

- E/S básica: `iostream.h`
- E/S manipulación: `iomanip.h`
- Standard Library: `stdlib.h`
- Funciones fecha y tiempo: `time.h`

```
// C++ Algunos tipos básicos
#include <iostream.h>
int main() {
    int integer1, integer2, sum;
    cout << "Introduzca el primer entero\n";
    cin >> integer1;
    cout << "Introduzca el segundo entero\n";
    cin >> integer2;
    sum = integer1 + integer2;
    cout << "La suma es " << sum << endl;
    return 0;
}
```

# Variables

- Variables: Nombres para localizaciones en memoria
- Todas las variables deben tener un tipo (no todos los lenguajes obligan a esto).
- Todas las variables deben ser declaradas antes de ser usadas
- Los tipos de variables definidos en C++ son
  - `int char float double bool`
- Pueden declararse nuevos tipos (ya los veremos)
- Los nombres de las variables
  - Pueden contener caracteres alfanuméricos y `_`
  - No pueden empezar por un número
  - Son case sensitive

# Variables (II)

- Prácticas habituales para nomenclatura
  - Acogerse a algún tipo de nomenclatura (GNU, o Linux Kernel coding, por ejemplo)
  - Los nombres deben ser auto explicativos
  - Se utilizan principalmente sustantivos
  - Notación húngara: utilizar prefijos para identificar el tipo de una variable

```
void vVariableVacía;  
bool bOperaciónValida;  
char cTeclaPulsada;  
int nNúmeroCeldas;  
long lTotalUnidades;  
float fPrecioUnitario;  
double dAnguloMinimo;  
EBoole eResultado
```

# Variables (III)

- Se permite la declaración de variables en cualquier lugar de un bloque entendiéndose como bloque el conjunto de sentencias encerradas entre llaves { }
- El ámbito de una variable es desde el punto en el que esta está declarada hasta el final del bloque en el que se encuentra
- Debe tenerse cuidado con esta regla: puede llevar a problemas de legibilidad

```
void funcion(void)
{
    int i=8;
    for (int j = 0 ; j < i ; j++ )
    {
        printf("Hola mundo");
        int k=3;
        printf("valor de k %d",k);
    }
    if(--i==0) return;
}
```

# Variables (IV)

- Una variable local oculta a una variable global si comparten nombre
- Se dispone del operador `::` *operador de resolución de visibilidad* o *scope*. Este operador, antepuesto al nombre de una variable global que está oculta por una variable local del mismo nombre, permite acceder al valor de la variable global. No es posible, sin embargo, acceder a una variable local que esté oculta por otra variable local

```
int i=0;
int main()
{
    int i;
    While( ::i < 10 )
    {
        ++::i;
        for( i=0 ; i<10 ; i++)
            printf("%d : %d\n", ::i, i);
    }
    return 1;
}
```

# Variables (V)

- Cada variable tiene un *storage class* o tipo de almacenamiento, que determina el tiempo que la variable existe en memoria
- Por ejemplo, las variables globales (declaradas fuera de todo bloque o función) se crean una sola vez y viven durante toda la ejecución
- Puede controlarse el almacenamiento con
  - `auto` La variable se crea cada vez que se accede al bloque en el que está definida
  - `register` Lo mismo que `auto` pero se le dice al compilador que la cree lo más rápido posible (utilizando un registro del procesador)
  - `static` Se crea una sola vez, incluso siendo una variable local
  - `extern` Hace referencia a una variable global declarada en cualquier parte

# Variables (VI)

- Usos prácticos de los tipos de almacenamiento
  - Las variables locales son `auto` por defecto
  - Las variables globales son `static` por defecto
  - Declarar una variables como `static` hace que la variable recuerde el último valor

- En C++ se introduce el operador referencia `&`

```
int &blah = count;  
// blah es la misma variable que count
```

- Una referencia no es una copia de la variable referenciada, sino que es la misma variable con un nombre diferente. Su principal aplicación está en el paso de parámetros por referencia a las funciones
- Una referencia no es un puntero o algo externo a la variable referenciada... es la misma variable con un nombre distinto

# Operadores

- **Operadores matemáticos**

- + - \* / %
- Siguen reglas de *precedencia y asociatividad*

<i>Operador</i>	<i>Asociatividad</i>	<i>Precedencia</i>
()	Izquierda a derecha	Alto
* / %	Izquierda a derecha	Medio
+ -	Izquierda a derecha	Bajo

- **Operadores lógicos (muy poca prioridad)**

- > Mayor que
- >= Mayor o igual que
- < Menor que
- <= Menor o igual que
- == Igual que
- != Distinto de

# Modificador const

- Se añade para indicarle al compilador que el valor no puede ser cambiado

```
const double factor = 5.0/9.0;  
const double offset = 32.0;  
celcius = (fahr - offset)*factor;
```

- Si se intenta cambiar al valor, el compilador emitirá un mensaje de error
- Usos:
  - Asegurar la integridad de variables así definidas en el modelado
  - Evitar que otros objetos, clases o funciones puedan modificar el valor de una variable pasada como `const`

# Estructuras de control

- C++ dispone de diversas estructuras de control
  - `if`
  - `if/else`
  - `switch`
  - `while`
  - `for`
  - `do/while`
- Es muy importante seleccionar y unificar un mismo estilo de código para todo el proyecto (estilo K&R, ANSI, GNU, Linux...)
- Como en C, disponemos de los operadores `++` y `--` para incrementar o decrementar variables
- Hay nuevos operadores

```
foo = foo + 17;  
// Podemos abreviarlo así  
foo += 17;
```

```
foo -= 17;  
foo *= 17  
foo /= 17;
```

# Más operadores

- Se dispone del operador booleano `&&` ejecuta un AND lógico a dos condiciones

```
( cond1 && cond2 )
```

- Operador booleano `||` que ejecuta un OR lógico a dos condiciones

```
( cond1 || cond2 )
```

- Operador unario negación `!` que niega una condición

```
( !cond1 )
```

# Tipos de datos complejos

- `struct` y `union` existen al igual que en C, pero se simplifica la declaración y uso de las mismas

```
//declaración del patrón de la estructura complejo
struct complejo
{
    float x;
    float y;
};
void main()
{
    //petición de variables
    int j;
    complejo micomplejo;
    // En C habría que poner "struct complejo micomplejo"
    //código
    j=sizeof(complejo);
}
```

# Tipos de datos complejos (II)

- Se introduce un nuevo tipo de datos `enum` para enumeraciones (en C existía pero sólo para `int`)

```
enum notas {suspense, aprobado, bien, notable, sobresaliente};
notas nota1 = bien;
notas nota2;
nota2 = notas(2); //conversión explícita.
// nota2 = 2; sería en C pero que en C++ es incorrecto
```

```
enum colores {rojo=3,verde=5, azul, amarillo=8};
int i = verde;
```

- Se introducen las uniones anónimas para definir un conjunto de campos ubicados en la misma zona de memoria

```
struct datos /*definición del patrón de estructuras*/
{
    int tipo;
    union _dato /*union dentro de la estructura*/
    {
        char caracter;
        int entero;
        float decimal;
    } dato;
};

int main()
{
    struct datos midato;
    midato.tipo=2;
    midato.dato.entero=5;
}
```

# Funciones

- Todas las funciones en C++ tienen un tipo. Aquellas que no devuelven nada, tienen tipo `void`
- Las funciones reciben parámetros que son tratados como variables locales dentro del cuerpo de la función
- Las funciones reciben una copia de las variables pasadas en los parámetros. El paso de referencias se hace a través del operador `&`
- Como en C, se definen prototipos de funciones en los archivos `.h` o `.hpp` para evitar problemas de *scope* o ámbito.
- Las funciones se pueden llamar a sí mismas: C++ permite la *recursividad*

# Funciones `inline`

- Con este modificador indicamos al compilador que consideramos conveniente que las llamadas realizadas a esta función sean sustituidas por el cuerpo de código de la función.
  - El programa resultante es más rápido, al evitarse el salto a la función
  - A costa de un código ejecutable más extenso
- Es recomendable utilizar el modificador `inline` en funciones pequeñas, que son llamadas en pocos lugares
- Para poder asignar el modificador `inline` a una función, dicha función debe estar definida antes de que sea invocada, de lo contrario el compilador no lo tendrá en cuenta.
  - las funciones `inline` son normalmente definidas en los ficheros de cabecera.

# Funciones inline (II)

```
inline int maximo(int a, int b)
{
    return((a>b)?a:b);
}
int main()
{
    int a,b;
    scanf("%d%d",&a,&b);
    printf("El máximo de %d y %d es %d",a,b,maximo(a,b));
}
```

- Recuerdan a las macros de C, pero ayudan con el tipado de los argumentos

# Funciones sobrecargadas

- Declarar y definir varias funciones distintas que tienen un mismo nombre.
- En el momento de la compilación se decide si se llama a una u otra función dependiendo del *número y/o tipo de los argumentos* actuales de la llamada a la función, pero no en el tipo de valor de retorno

```
struct complejo
{
    float real, imaginario;
}
float suma(float a, float b)
{
    return (a+b);
}
float suma(complejo a, complejo b)
{
    complejo c;
    c.real=a.real+b.real;
    c.imaginario=a.imaginario+b.imaginario;
    return c;
}
int main()
{
    complejo a={1.0F,1.5F},b={0.0F,1.1F},c;
    float d=3.0F,e=8.2F,f;
    c=suma(a,b); //utiliza la función suma de complejos
    f=suma(d,e); //utiliza la función suma de enteros
}
```

# Funciones sobrecargadas (II)

- Hay que tener cuidado con la posibilidad de que se produzcan ambigüedades en la decisión de que función va a ser ejecutada.
  - Romper la ambigüedad mediante una conversión explícita

```
void imprime(double a);
void imprime(long a);
int main()
{
    int b=8;
    imprime(b);
    imprime(static_cast<long>(b));
    imprime((double)b);
}
```

# Parámetros por defecto

- Es posible definir valores por defecto para cada uno de los argumentos, de forma que si en el uso de una función no son indicados por el programador, dichos parámetros tomaran el valor previsto

```
void imprimeVector(float v[], int tam=3, bool linea=false)
{
    printf("(");
    For( int i=0 ; i<tam ;i++ )
        printf(" %f",v[i]);
    if(linea)
        printf(" )\n");
    else
        printf(" ) ");
}
int main()
{
    float vector[5]={1.0F,2.0F,3.0F,4.0F,5.0F};
    imprimeVector(vector);
    imprimeVector(vector,5);
    imprimeVector(vector,5,true);
}
```

- Una vez omitido un argumento en una llamada, hay que omitir todos los posteriores
- Los parámetros por defecto deben situarse en el prototipo y no en la definición

# Reserva dinámica de memoria

- En C la reserva de memoria se hacía con las funciones `malloc` y `free` de la librería `stdlib`
- En C++ se introducen `new` y `delete`
- **new** permite asignar memoria perteneciente al área de almacenamiento libre para un objeto o para una matriz de objetos. Devuelve un puntero al espacio de memoria reservado, siendo el puntero del tipo especificado

```
int num=6, i;  
int *a = new int;  
int *b = new int[num];  
int (*c)[3] = new int[8][3];  
int *d[3];  
for (i=0 ; i<3 ;i++ )  
    d[i]=new int[8];
```

- El operador `[]` tiene preferencia sobre el `*` lo que explica la sintaxis de la línea 4

# Reserva dinámica de memoria

- **Nota:** En el caso de los vectores el modo de proceder es exactamente igual que en C. Un vector al final es un puntero que apunta a una zona de memoria en donde comienzan a situarse los elementos del vector de forma ordenada. De tal forma, que la dirección corresponde con la del primer elemento y tiene por tanto este tipo. Recordemos, que la razón principal para proceder así, esta en la aritmética de punteros y en los sistemas de indirección, logrando de esta forma que uno de los mecanismos básicos del lenguaje tenga correspondencia directa con el modo de funcionamiento de un microprocesador.
- El operador `new` no necesita de la conversión forzada
- *A recordar:* En la generación de un objeto, se realiza una llamada al constructor, e incluso se permite la inicialización

# Reserva dinámica de memoria

- El operador `delete` destruye un objeto creado por el operador `new`, liberando el sistema operativo la memoria ocupada por dicho objeto.
- A diferencia de lo que ocurría en C con la función `free`, `delete` puede ser utilizado sobre un puntero nulo (apunta a cero) en cuyo caso no realiza ninguna operación

```
delete a;  
delete [] b;  
delete [] c; //obsérvese que se considera c como vector  
for( i=0 ; i<3 ; i++ )  
    delete [] d[i]; //para cada new un delete
```

# Operaciones de E/S

- En C: funciones `printf` y `scanf` utilizables en C++
- En C++: esta operación se simplifica y generaliza introduciéndose el concepto de flujo
- *Un flujo* es un objeto que hace de intermedio entre el programa y el destino u origen de la información
- Cuando un programa en C++ se ejecuta, se crean automáticamente tres flujos identificados por
  - Un flujo de entrada estándar (normalmente el teclado): `cin`
  - Un flujo de salida estándar (habitualmente la pantalla): `cout`
  - Dos flujos hacia la salida estándar de error (pantalla): `cerr`, y `clog`
- En primer lugar, para poder incluir la librería estándar `iostream.h`

# Operaciones de E/S

```
#include<iostream.h>
int main()
{
    {
        int i=2,j=3;
        double dato=5.3;
        char a='a',b[]="hola";
        cout << i;
        cout << i << j << endl;
        cout << "el valor de dato es " << dato << endl;
        cout << "el carácter a=" << a << "y la cadena b" << b << endl;
    }
    {
        double b;
        int i;
        char c, d[50];
        cout << "introduzca un número real :";
        cin >> b; //al teclear 3.5, en b se almacena este valor
        cout << "el valor introducido es: " << b << endl;
        cout << "introduzca un real seguido de un entero y un char:";
        cin >> b >> i >> c;
        cout << "introduzca su nombre:";
        cin >> d;
    }
}
```

# Clases

- Las clases en C++ se pueden considerar como la evolución de las estructuras. Constan de
  - Un conjunto de datos miembro
  - Un conjunto de métodos o funciones miembro
  - Unos niveles de acceso a los datos y métodos de la clase
  - Un identificador o nombre asociado a la clase
- Tres momentos en la definición y utilización de una clase:
  - Declaración
  - Definición
  - Instanciación

# Clases (II)

```
//comienzo de la declaración
```

```
class complex
{
    private:
        double real;
        double imag;
    public:
        void estableceValor(float re, float im) ;
        float obtenModulo(void) ;
        void imprime() ;
};
//fin de la declaración
```

```
//comienzo de las definiciones
```

```
void complex::estableceValor(float re, float im)
{
    real=re ;
    imag=im ;
}
float complex::obtenModulo(void) {
    return (sqrt(real*real+imag*imag));
}
void complex::imprime(void) {
    cout << real << `+` << imag << `i`;
}
//fin de las definiciones
```

```
//comienzo del programa de ejemplo
void main()
{
    // creación de un objeto de la
    // clase complex
    complex micomplejo;
    // asigno a micomplejo el valor
    // 3.2+1.8i
    micomplejo.estableceValor(3.2,1.8);
    //imprimo el modulo
    cout << "El modulo de ";
    micomplejo.imprime();
    cout<< " es
    "<<micomplejo.obtenModulo();
}
```

# Clases (III)

- Formalmente

```
class <identificador>
{
    [<nivel de acceso a>:]
    <lista de miembros de la clase>
    [<nivel de acceso b>:]
    <miembros de la clase>]
    [<...>]
}[lista de objetos];
```

```
<tipo> <iden_clase>::<iden_metodo> (<argumentos>)
{
    [código del método]
}
```

# Clases (IV)

- Los datos miembro o atributos de un objeto son distintos para cada objeto, sin embargo las funciones o métodos son comunes a todos los objetos de una misma clase.
- Es decir, cada objeto almacena sus propios datos, pero para acceder y operar con ellos, todos comparten los mismos métodos definidos en su clase.
- Para que un método conozca la identidad del objeto en particular para el que ha sido invocado, C++ proporciona un puntero al objeto denominado `this`.

```
void complex::estableceValor(float re, float im)
{
    this->real=re ;
    this->imag=im ;
}
```

# Clases amigas

- La palabra clave `friend` es un modificador que puede aplicarse tanto a clases como a funciones para inhibir el sistema de protección de atributos y funciones de una clase. La idea general es que la parte privada de una clase solo puede ser modificada o consultada por la propia clase o por los amigos de la clase (tanto métodos como funciones)
  - La amistad no puede transferirse: si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C
  - La amistad no puede heredarse
  - La amistad no es simétrica

# Clases amigas (II)

- Funciones amigas de una clase

```
#include <iostream>
class A
{
public:
    A(int i) {a=i;};
    void Ver() { cout << a << endl; }
private:
    int a;
    friend void Visualiza(A); // "Ver" es amiga de la clase A
};
// La función Visualiza puede acceder a miembros privados
// de la clase A, ya que ha sido declarada "amiga" de A
void Visualiza(A Xa)
{
    cout << Xa.a << endl;
}
void main()
{
    A Na(10);
    Visualiza(Na); // imprime el valor de Na.a
    Na.Ver(); // Equivalente a la anterior
}
```

# Clases amigas (III)

- Métodos de una clase amigos de otra clase

```
#include <iostream>
class A; // Declaración previa (forward)
class B
{
public:
    B(int i){b=i;};
    void Ver() { cout << b << endl; };
    bool EsMayor(A Xa); // Compara b con a
private:
    int b;
};

class A
{
public:
    A(int i=0)
    void Ver()
private:
    // Función amiga tiene acceso
    // a miembros privados de la clase A
    friend bool B::EsMayor(A Xa);
    int a;
};

bool B::EsMayor(A Xa)
{
    return (b > Xa.a);
}

void main(void)
{
    A Na(10);
    B Nb(12);
    Na.Ver();
    Nb.Ver();
    if(Nb.EsMayor(Na)) cout << "Nb es mayor que Na" << endl;
    else cout << "Nb no es mayor que Na" << endl;
}
```

# Clases amigas (IV)

- Clase amiga de otra clase
  - Cuando lo que se desea es que todos los métodos de una clase tengan la capacidad de acceder a la parte privada de otra

```
class C1
{
    ...
};
class C2
{
    ...
    friend class C1;
    ...
};
```

# Constructores y Destructores

- Constructor

- Funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara
- Los constructores tienen el mismo nombre que la clase
- No retornan ningún valor
- No pueden ser heredados
- Deben ser públicos (no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado)

# Constructores (I)

```
#include <iostream>
class pareja
{
public:
    // Constructor
    pareja(int a2, int b2);
    // Funciones miembro de la clase "pareja"
    void Lee(int &a2, int &b2);
    void Guarda(int a2, int b2);
private:
    // Datos miembro de la clase "pareja"
    int a, b;
};

//definición del constructor
pareja::pareja(int a2, int b2)
{
    a = a2;
    b = b2;
}
```

```
void main(void)
{
    ...
    pareja par1(12, 32);
    // Ilegales:
    pareja par1;
    pareja par1();

    ...
}
```

# Constructores (II)

- Cuando se especifique un constructor para una clase, el compilador crea uno por defecto sin argumentos al que se denomina como **constructor de oficio**
- Para declarar objetos usando el constructor por defecto o un constructor que se haya declarado sin parámetros no se debe usar el paréntesis

- `pareja par2();` // NO
- `pareja par2;` // VÁLIDO

- `explicit` es una palabra reservada para evitar estas situaciones

```
class A {
    public:
        A(int);
};
void f(A) {}
void g()
{
    A a1 = 37; // Declaración MUY confusa
    A a2 = A(47);
    A a3(57);
    A1 = 67; // Asignación MUY confusa
    f(77); // Llamada MUY confusa
}
```

```
class A {
    public:
        explicit A(int);
};
void f(A) {}
void g()
{
    A a1 = 37; // Ilegal
    A a2 = A(47); // OK
    A a3(57); // OK
    a1 = 67; // Ilegal
    f(77); // Ilegal
}
```

# Inicialización de objetos

```
pareja::pareja(int a2, int b2)
{
    a = a2;
    b = b2;
}
```

Se puede sustituir el constructor por:

```
pareja::pareja(int a2, int b2) : a(a2), b(b2) {}
```

En C++ se prefiere este segundo estilo

- Es posible sobrecargar el constructor

```
class pareja
{
public:
    // Constructor anterior
    pareja(int a2, int b2) : a(a2), b(b2) {}
    // Constructor por defecto
    pareja() : a(0), b(0) {}
    // Otro constructor
    pareja(double num);
    ...
}
```

# Constructor de copia

- Sirve para inicializar un objeto como copia de otro
- Si no se define ninguno, el compilador asigna uno de oficio

```
class pareja
{
    ...
    pareja(const pareja &p):a(p.a),b(p.b){}
    ...
}
void main(void)
{
    pareja p1(12,32);
    pareja p2(p1); //Uso del constructor de copia
    ...
}
```

# Destructor

- Es llamado cuando el objeto va a dejar de existir por haber llegado al final de su vida
  - Si el objeto local ha sido definido dentro de un bloque `{...}`, el destructor es llamado cuando el programa llega al final de ese bloque
  - Si el objeto es `global` o `static` su duración es la misma que la del programa, y por tanto el destructor es llamado al terminar la ejecución del programa
  - Los objetos creados con reserva dinámica de memoria (en general, los creados con el operador `new`) no están sometidos a las reglas de duración habituales, y existen hasta que el programa termina o hasta que son explícitamente destruidos con el operador `delete`. En este caso la responsabilidad es del programador, y no del compilador o del sistema operativo

# Destructor

- El operador `delete` llama al destructor del objeto eliminado antes de proceder a liberar la memoria ocupada por el mismo
- El destructor es siempre *único* (no puede estar sobrecargado) y *no tiene argumentos* en ningún caso. *Tampoco tiene valor de retorno*

```
~pareja();
```

# Operadores sobrecargados

- Al igual que se podían sobrecargar funciones, se pueden sobrecargar métodos de clases, y también operadores
- En C++ los operadores pasan a ser unas funciones como otra cualquiera, pero que
- permiten para su ejecución una notación especial. Por ejemplo, el operador '+' realizará la suma aritmética o de punteros de los operandos que pongamos a ambos lados
- Se pueden sobrecargar casi todos los operadores, con las siguientes limitaciones
  - Se pueden sobrecargar todos los operadores excepto ".", ":", ":", ".\*", "::", "?:"
  - Los operadores = [ ] -> new y delete, sólo pueden ser sobrecargados cuando se definen como miembros de una clase
  - Los argumentos para los operadores externos deben ser tipos enumerados o estructurados: struct, union o class

# Operadores sobrecargados (II)

Prototipo para los operadores:

```
<tipo> operador <operador> (<argumentos>);
```

Definición para los operadores:

```
<tipo> operador <operador> (<argumentos>)  
{  
    <sentencias>;  
}
```

```
typedef struct _complex{  
    float real;  
    float imag;  
}complex;
```

```
complex operator +(complex x,complex y)  
{  
    complex z;  
    z.real = x.real + y.real;  
    z.imag = x.imag + y.imag;  
    return z;  
}
```

```
#include <iostream.h>  
void main()  
{  
    complex a = {5.0F,3.0F}, b = {3.0F,1.2F}, c;  
    c = a + b;  
    cout << c.real << "+" << c.imag << "i" << endl;  
}
```

# Sobrecarga de operad. binarios

- Cuando un operador binario es a su vez miembro de una clase, se asume que el primer operando de la operación es el propio objeto de la clase donde se define el operador. Sólo será necesario especificar un operando, puesto que el otro es el propio objeto

```
<tipo> operator<operador binario>(<tipo> <identificador>);
```

- Habitualmente <tipo> será para un objeto de la misma clase, pero no es necesario

# Sobrecarga de operad. binarios

```
#include <iostream.h>
```

```
class Tiempo
```

```
{
```

```
public:
```

```
Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}
```

```
void Mostrar(){cout << hora << ":" << minuto << endl;};
```

```
Tiempo operator+(Tiempo h);
```

```
Tiempo operator+(int mins);
```

```
private:
```

```
int hora;
```

```
int minuto;
```

```
};
```

```
Tiempo Tiempo::operator+(Tiempo h)
```

```
{
```

```
Tiempo temp;
```

```
temp.minuto = minuto + h.minuto;
```

```
temp.hora = hora + h.hora;
```

```
if(temp.minuto >= 60)
```

```
{
```

```
temp.minuto -= 60;
```

```
temp.hora++;
```

```
}
```

```
return temp;
```

```
}
```

```
}
```

```
Tiempo Tiempo::operator +(int mins)
```

```
{
```

```
Tiempo temp;
```

```
temp.minuto = minuto + mins;
```

```
temp.hora = hora + temp.minuto/60;
```

```
temp.minuto = temp.minuto % 60;
```

```
return temp;
```

```
}
```

```
void main(void)
```

```
{
```

```
Tiempo Ahora(12,24), T1(4,45);
```

```
T1 = Ahora + T1;
```

```
T1.Mostrar();
```

```
// Utilización de objetos
```

```
// sin referencia directa
```

```
(Ahora + Tiempo(4,45)).Mostrar();
```

```
(Ahora+45).Mostrar();
```

```
}
```

# Sobrecarga de =

- Si no se sobrecarga este operador, el compilador asignará uno por defecto que realizará una copia literal de lo que hay en la memoria de un objeto sobre el otro. Este operador por defecto es válido siempre que no se esté utilizando memoria dinámica

```
// Veamos el código
class Cadena
{
public:
    Cadena(char cad) {
        cadena = new char[strlen(cad)+1];
        strcpy(cadena, cad);
    }
    Cadena() {
        cadena = NULL;
    }
    ~Cadena() {
        delete[] cadena;
    }
    void Mostrar() {
        cout << cadena << endl;
    }
    void RellenarDeGuiones() {
        For( int i = 0 ; i < strlen(cadena) ; i++ )
            cadena[i]='-';
    }
private:
    char *cadena;
};

Cadena::Cadena(char cad) {
    cadena = new char[strlen(cad)+1];
    strcpy(cadena, cad);
}

Cadena::RellenarDeGuiones() {
    For( int i = 0 ; i < strlen(cadena) ; i++ )
        cadena[i]='-';
}

Cadena &Cadena::operator=(const Cadena &c)
{
    if(this != &c) {
        delete[] cadena;
        if(c.cadena) {
            cadena = new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
        } else {
            cadena = NULL;
        }
    }
    return *this;
}

// genera errores
// imprevistos
C2.RellenarDeGuiones();
C1.Mostrar();
C2.Mostrar();
}
```

# Sobrecarga de operad. unarios

- Cuando se sobrecargan en una clase el operando es el propio objeto de la clase donde se define el operador. Por lo tanto los operadores unarios dentro de las clases en un principio no requieren de operandos

```
<tipo> operator<operador unitario>();
```

```
class Tiempo
{
    ...
    Tiempo operator++();
    ...
};
Tiempo Tiempo::operator++()
{
    minuto++;
    while (minuto >= 60) {
        minuto -= 60;
        hora++;
    }
    return *this;
}
```

```
void main(void)
{
    ...
    T1.Mostrar();
    ++T1;
    T1.Mostrar();
    ...
}
```

# Sobrecarga del operador [ ]

- Se usa para acceder a valores de objetos de una determinada clase como si se tratasen de arrays. Los índices no tienen por qué ser de un tipo entero o enumerado, no existe esa limitación
- Utilidad
  - Estructuras dinámicas de datos: (listas, árboles, vectores dinámicos)
  - Arrays asociativos, donde los índices sean por ejemplo, palabras

```
class vector
{
    public:
        ...
        // Podría permitir acceder utilizando
        // v[0.4]
        double operator[] (double ind);
        ...
};
```

# Más sobrecarga

- `new` y `delete`: Si los sobrecargamos dentro de la clase, cada vez que hagamos un `new` a nuestra clase se llamará a nuestro operador. Si los sobrecargamos globalmente se llamará a nuestras funciones cada vez que hagamos un `new` o un `delete` de cualquier cosa (clases o variables). Esta característica nos permite hacer *contabilidad de punteros*, para ver si liberamos todo lo que reservamos o liberamos lo mismo más veces de la cuenta
- Se puede incluso sobrecargar el operador llamada `()`
- `cast` Este operador se utiliza para hacer conversiones implícitas y no implícitas: Se sobrecarga para así asegurar la conversión entre tipos de datos “conflictivos” o muy distintos en los que queremos garantizar una conversión segura

```
class ComplejoC {  
    public:  
        // Permite hacer un cast de ComplejoC a double  
        operator double ();  
}
```

# Miembros `static`

- Atributos `static`: Se utilizan cuando se hace necesario de disponer de una variable común a todos los objetos de una clase
- No es un atributo específico de un objeto sino que más bien es un *atributo de la clase*; esto es, sólo hay una copia que comparten todos los objetos de la clase.
- El atributo existe y puede ser utilizado aunque no exista ningún objeto de la clase.
- Debe ser inicializado de forma global una única vez

```
class libro
{
    public:
        ...
        static int contador;
        ...
};
// Aquí se inicializa, Y SE CREA. No se puede acceder a
// ella antes de esta línea. Suele estar en el .cpp
// ya que los archivos .h se pueden leer varias veces
// ( con lo cual ocurrirían múltiples inicializaciones )
int libro::contador=0;
```

# Miembros `static` (II)

- Un método declarado como `static` carece del puntero `this` por lo que no puede ser invocado para un objeto de su clase, sino que se invoca en general allí donde se necesite utilizar para la operación para la que ha sido escrito
- Es imposible que un método `static` pueda acceder a un miembro no `static` de su clase pero sí puede acceder a un miembro `static`

# Herencia

- La herencia permite definir una clase modificando una o más clases ya existentes. Estas modificaciones consisten habitualmente en añadir nuevos miembros (variables o funciones), a la clase que se está definiendo, aunque también se puede redefinir variables o funciones miembro ya existentes
- La clase de la que se parte en este proceso recibe el nombre de *clase base*, y la nueva clase que se obtiene se denomina *clase derivada*
- En algunos casos una clase no tiene otra utilidad que la de ser clase base para otras clases que se deriven de ella. A este tipo de clases base, de las que no se declara ningún objeto, se les denomina clases base abstractas y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas

# Herencia. Niveles de acceso

Tipo de dato de la clase base	Clase derivada de una clase base <code>public</code>	Clase derivada de una clase base <code>private</code>	Clase sin relación de herencia
<code>private</code>	No accesible	No accesible	No accesible
<code>protected</code>	<code>protected</code>	<code>private</code>	No accesible
<code>public</code>	<code>public</code>	<code>private</code>	Accesible ( <code>.</code> , <code>-&gt;</code> )

- Para indicar que una clase deriva de otra es necesario indicarlo en la declaración de la clase derivada, especificando el modo (`public` o `private`) en que deriva de su clase base

```
class <Clase_Derivada> : [public|private] <Clase_Base>
```

- Si un miembro heredado se redefine en la clase derivada, el nombre redefinido oculta el nombre heredado que ya queda invisible para los objetos de la clase derivada. Podemos acceder al miembro de la clase base con el operador **scope** `::`

```
clase_base::miembro_oculto()
```

# Herencia. Construcción

- No se heredan
  - Constructores
  - Destrucciones
  - Funciones `friend`
  - Funciones y datos estáticos de la clase
  - Operador de asignación ( = ) sobrecargado
- El constructor de la clase derivada debe llamar al constructor de la clase base. Se define el *inicializador base* que indica la forma de llamar a los constructores de las clases base y así poder inicializar las variables miembros heredadas

```
decimal(int val, int dec) : numero(val) {  
    valor = dec;  
}
```

# Herencia. Construcción

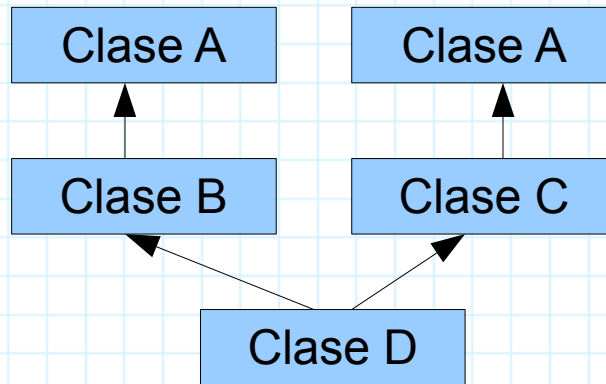
- Los constructores se llaman en el orden de derivación de las clases
- Los destructores se llaman en orden inverso

# Herencia Múltiple

```
class <c_derivada> :  
    [public|private] <c_base_1>, [public|private] <base_2>, ...  
{  
    ...  
}
```

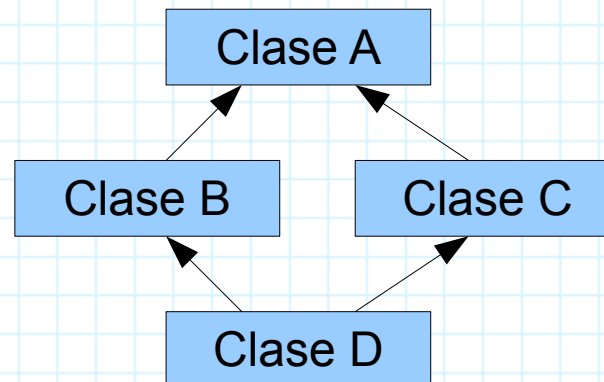
- Las clases base se construyen en el orden en el que aparecen en la declaración de la clase derivada de izquierda a derecha
- Los inicializadores base siguen esta misma sintaxis
- Cuando es necesario utilizarlos, se colocan tras la clase base separados por comas
- La herencia múltiple es obviada en otros lenguajes orientados a objetos y desaconsejada por muchos autores

# Clases base virtuales



- La clase D hereda dos veces de la clase A. Se produce una situación de ambigüedad, ya que tenemos duplicados los datos y funciones de A en D. Solución:

```
class ClaseB : virtual public ClaseA {};  
class ClaseC : virtual public ClaseA {};  
class ClaseD : public ClaseB, public ClaseC {};
```



# Herencia. Conversiones

```
// Analicemos
```

```
ClaseA ca(2);
ClaseB cb(8,4);
ca.funcion();
cb.funcion();
cb = ca; // ERROR DE COMPILACIÓN
ca = cb; // Válido, el padre sí puede contener el hijo
ClaseA *c1=new ClaseA(1);
ClaseB *c2=new ClaseB(2,3);
ClaseA *c3=c1,*c4=c2;
ClaseB *c5;
c5 = static_cast<ClaseB *>(c3);
c5->mostrar(); //error en ejecución NO DETECTABLE en
               // compilación. c5 proviene de un
               // puntero a ClaseA en todo caso, ¿a
               // qué apuntaría funcion()?

c5 = static_cast<ClaseB *>(c4);
c5->mostrar();
delete c1;
delete c2;
```

```
void funcion ();
```

```
};
```

# Herencia. Conversiones (II)

- Existe en C++ la posibilidad de que al realizar una conversión de tipos se compruebe si el objeto apuntado por un puntero es o no asignable a un puntero de una clase derivada: `dynamic_cast`
- Dos condiciones
  - Esta operación solo puede aplicarse sobre tipos polimórficos: esto ocurre cuando se tiene que alguno de los métodos de la clase es `virtual`
  - Hay que indicar al compilador que debe incluir información sobre los tipos de clase en el código

```
// SOLUCIÓN AL CASO ANTERIOR
virtual ~ClaseA(){}
// Y el final de la función main anterior lo
// reescribiremos de la forma siguiente:
c5= dynamic_cast<ClaseB *>(c3);
if(c5!=NULL) c5->funcion();
c5= dynamic_cast<ClaseB *>(c4);
if(c5!=NULL) c5->funcion();
```

# Herencia. Operador de copia

```
class Derivada : public Base
{
    int numero;
public:
    Derivada &operator=(const Derivada &d)
    {
        Base::operator = (d);
        numero = d.numero;
        return *this;
    }
};
```

# Métodos virtuales

- Es un método de una clase base que puede ser redefinido en cada una de las clases derivadas de esta, y que una vez redefinido puede ser accedido por medio de un puntero o una referencia a la clase base. La llamada se resuelve en función del objeto referido en vez de en función de con qué se hace la referencia.
- Cuando una clase tiene algún método virtual (bien directamente, bien por herencia) se dice que dicha clase es *polimórfica*

```
virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];
```

# Entendiendo los métodos virtuales

```
class Persona {  
public:  
    virtual void VerNombre() const = 0;  
};  
  
class Estudiante : public Persona {  
public:  
    void VerNombre() const override {  
        cout << "Est: " << nombre << endl;  
    }  
};  
  
class Empleado : public Persona {  
public:  
    void VerNombre() const override {  
        cout << "Emp: " << nombre << endl;  
    }  
};  
  
void main() {  
    Persona *Pepito = new Estudiante("Jose");  
    Persona *Carlos = new Empleado("Carlos");  
    Carlos->VerNombre(); // Imprime "Carlos"  
    Pepito->VerNombre(); // Imprime "Jose"  
    delete Pepito;  
    delete Carlos;  
  
    Estudiante *Juan = new Estudiante("Juan");  
    Empleado *Alberto = new Empleado("Alberto");  
    Juan->VerNombre(); // Imprime "Est: Juan"  
    Alberto->VerNombre(); // Imprime "Emp: Jose"  
  
    Estudiante David("David");  
    Persona &pDavid = David;  
    pDavid.VerNombre(); // Imprime "David"  
  
}
```

```
void VerNombre() {  
    cout << "Est: " << nombre << endl;  
}  
};
```

# Entendiendo los métodos virtuales

- Si redefinimos la clase persona

```
class Persona {  
    public:  
        Persona(char *n) { strcpy(nombre, n); }  
        virtual void VerNombre() {  
            cout << nombre << endl;  
        }  
    protected:  
        char nombre[30];  
};
```

- Ahora, al llamar a `Pepito->VerNombre()` se invoca a la función `VerNombre` de la clase `Estudiante`, y al llamar a `Carlos->VerNombre()` se invoca a la función de la clase `Empleado`, a pesar de que tanto `Pepito` como `Carlos` son punteros a la clase `Persona`

# Métodos virtuales

- Una vez que una función es declarada como virtual, lo seguirá siendo en las clases derivadas, es decir, la propiedad virtual se hereda
- Si la función virtual no se define exactamente con el mismo tipo de valor de retorno y el mismo número y tipo de parámetros que en la clase base, no se considerará como la misma función, sino como una función superpuesta
- El nivel de acceso no afecta a la virtualidad de las funciones. Es decir, una función virtual puede declararse como privada en las clases derivadas aun siendo pública en la clase base, pudiendo por tanto ejecutarse ese método privado desde fuera por medio de un puntero a la clase base
- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado

# Métodos virtuales

- Una llamada a un método normal se resuelve siempre en función del tipo de la referencia o puntero utilizado
- Una llamada a un método virtual especificando la clase, exige la utilización del operador de resolución de ámbito ::, lo que suprime el mecanismo de virtualidad. Evidentemente este mecanismo solo podrá utilizarse para el método de la misma clase del contenedor o de clases base del mismo
- Por su modo de funcionamiento interno (es decir, por el modo en que realmente trabaja el ordenador) las funciones virtuales son un poco menos eficientes que las funciones normales

# ¿Cómo se implementa?

- Sólo a título informativo: Cada clase que utiliza funciones virtuales tiene un vector de punteros, uno por cada función virtual, llamado *v-table*. Cada uno de los punteros contenidos en ese vector apunta a la función virtual apropiada para esa clase, que será, habitualmente, la función virtual definida en la propia clase. En el caso de que en esa clase no esté definida la función virtual en cuestión, el puntero de *v-table* apuntará a la función virtual de su clase base más próxima en la jerarquía, que tenga una definición propia de la función virtual. Esto quiere decir que buscará primero en la propia clase, luego en la clase anterior en el orden jerárquico y se irá subiendo en ese orden hasta dar con una clase que tenga definida la función buscada.
- Cada objeto creado de una clase que tenga una función virtual contiene un puntero oculto a la *v-table* de su clase. Mediante ese puntero accede a su *v-table* correspondiente y a través de esta tabla accede a la definición adecuada de la función virtual. Es este trabajo extra el que hace que las funciones virtuales sean menos eficientes que las funciones normales

# Constructores y destructores virtuales

- Supongamos que tenemos una estructura de clases en la que en alguna de las clases derivadas exista un destructor. Un destructor es una función como las demás, por lo tanto, si

destruimos un puntero a una clase derivada, el destructor de la clase base no se llama. Esto puede ser un problema si queremos que se llame al destructor de la clase base cuando se destruya un objeto de la clase derivada. La solución es declarar el destructor de la clase base como virtual.

```
void main() {  
    Base b = new Derivada();  
    // Uso b  
    delete b;    // OJO!!! Estoy llamando al  
                // destructor de Base  
    // SOLUCIÓN: Declara el destructor de  
    // Base como virtual  
}
```

```
class Base  
{  
    public:  
        Base() {};  
        ~Base() {};  
}
```

```
class Derivada  
{  
    public:  
        Derivada() {};  
        ~Derivada() {};  
}
```

# Funciones virtuales puras

- La definición de una función como virtual pura hace necesaria la definición de esa función en las clases derivadas, a la vez que imposibilita su utilización con objetos de la clase base

```
virtual <tipo> <nombre_función>(<lista_parámetros>) = 0;
```

- No hace falta definir el código de esa función en la clase base
- *No se pueden definir objetos de la clase base, ya que no se puede llamar a las funciones virtuales puras*
- Sin embargo, es posible definir punteros a la clase base, pues es a través de ellos como será posible manejar objetos de las clases derivadas

# Clases abstractas

- Aquella clase que contiene una o más funciones virtuales puras. Se llaman así puesto que no puede existir ningún objeto de esa clase
- Si una clase derivada no redefine una función virtual pura, la clase derivada la hereda como función virtual pura y se convierte también en clase abstracta
  - No está permitido crear objetos de una clase abstracta
  - Siempre hay que definir todas las funciones virtuales de una clase abstracta en sus clases derivadas, no hacerlo así implica que la nueva clase derivada será también abstracta

# Namespace

- Permiten agrupar entidades, como clases, objetos o funciones bajo un mismo nombre. Es la forma en la que el ámbito global (*global scope*) puede ser dividido en subámbitos
- Permite declarar identificadores sin que éstos se solapen

```
namespace nombre_del_namespace
{
    ... //declaraciones y/o definiciones de variables
    // funciones, clases, ...
}
```

```
namespace miNamespace      miNamespace::a;
{                          miNamespace::b;
    int a, b;
}
```

- Un namespace puede contener namespace anidados

# Namespace (II)

- No pueden hacerse declaraciones de namespace dentro de bloques (p.e.: dentro de una función)
- Se dice que la declaración namespace es una declaración *abierta*, en el sentido de que pueden aparecer, en un contexto (que puede abarcar varios ficheros), varias declaraciones de un namespace, en cuyo caso se irá incrementando el contenido del namespace
- Cuando en un fichero de código aparezca una declaración namespace sin un identificador de namespace, las declaraciones que este contenga sólo podrán usarse en ese fichero. A este namespace se le llamaría cerrado, porque no va a ser posible extenderlo en otros ficheros de código
- Aunque la declaración de un namespace puede contener las definiciones de las entidades declaradas en el mismo, esto no tiene porqué ser así; es más, la mayoría de las veces será más aconsejable incluir sólo las declaraciones

# Templates. Breve introducción

- La generalidad es una propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o alguno de sus miembros
- Esta propiedad no es imprescindible en un lenguaje de programación orientado a objetos y ni siquiera es una de sus características
- Apareció mucho más tarde que el resto del lenguaje, al final de la década de los ochenta
- La utilidad principal de este tipo de clases o funciones es la de agrupar variables cuyo tipo no esté predeterminado
  - El funcionamiento de una pila, una cola, una lista, un conjunto, un diccionario o un array es el mismo independientemente del tipo de datos que almacene

# Templates

- C++ utiliza una palabra clave específica `template` para declarar y definir funciones y clases genéricas. En estos casos actúa como un especificador de tipo y va unido al par de ángulos `< >` que delimitan los argumentos de la plantilla

```
template <T> void fun(T& ref); // función genérica
template <T> class C { /*...*/ }; // clase genérica
```

- Ejemplo:

```
double max(double a, double b)
int max(int a, int b)
```

- Puede ser fácil

```
// <class T> es la
template <class T>
{
    return (a > b)
}
```

La idea fundamental es que el compilador deduce los tipos concretos de los parámetros de la plantilla de la inspección de los argumentos actuales utilizados en la invocación

```
int i, j;
UnaClase a, b;
...
int k = max(i, j);
UnaClase c = max(a, b);
```