

# Programación en C++ (1) : aspectos procedurales

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009



# The lands of C++



J.B. Hayet

Programación, Septiembre 2009

# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++
- 5 Tipos en C++



# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++
- 5 Tipos en C++



# C++ : Antecedentes

El paradigma OO es aun mas viejo que el C : la mayoría de los paradigmas se pensaron al alba de inteligencia artificial (años 60-70); además, el **desarrollo creciente de la industria del software** y la necesidad de reutilización limpia de código hizo que la mayoría de los lenguajes estructurados iban a tener “versiones objeto”.



# C++ : Genesis

- El C fue desarrollado alrededor de 1972 por D. Ritchie en los Bell Labs
- El C++ fue desarrollado en ATT por B. **Stroustrup**, que le llamó primero “C with classes” (lo que sí era simplemente al principio), en 1980,
- 1983 : primera implementación de C++,
- 1985 : especificación de C++, en *The C++ Programming Language*,
- 1990 : *templates* y excepciones,
- 1998 : norma ANSI/ISO.



# C++ : Genesis

Algo **paradójico** : C++ no corresponde a una idea de proyecto definida, a algo bien planeado; es como un trabajo de laboratorio hecho por una persona aislada; misma remarca para Unix, y C; lenguajes muy bien pensados y especificados (como ADA) no han tenido tanto éxito (entienda el que pueda...).

El C++ nació en parte por las necesidades de un proyecto de Stroustrup que buscaba un lenguaje con a la vez elementos de bajo nivel y características de mas alto nivel ...



# C++ : Genesis

Mucha información en la pagina personal de Stroustrup:  
<http://www.research.att.com/~bs/>



# C++ : hoy

Aprovechó el éxito de C (el pasaje de C a C++ es muy fácil); se impuso y el resultado es que muchas librerías, APIs... existen ahora en C++.

Un ejemplo ilustrativo del éxito de la POO y de C++ son las **interfaces gráficas**, por ejemplo Qt.



# C++ : hoy

La competencia:

- Java
- C#
- Objective-C

Es relativamente simple entender lo básico de los otros al conocer el C++



# C++ : lo básico

Por la genesis, se puede entender que C++ toma muchas cosas de C; de hecho, casi todo programa escrito en C va a ser compilado por un compilador C++.

Ahora, se puede programar en C o C++ : para ser coherente, no mezclar los estilos (por ejemplo las salidas hacia flujos).

El C++ viene con la STL (Standard Template Library) que da mucho mas que la librería estandar (estructuras de datos, algoritmos, templates... ).



# C++ : lo básico

Una de las filosofías esenciales de C permanece : la gran **libertad** y **responsabilidad** dada al programador.

**Cuidado** : código valido puede generar graves errores (en particular con respecto a memoria).

No obstante, podemos estructurar mejor con POO.



# C++ : lo básico

Por lo POO, C++ alcanza niveles de abstracción mas altos; sin embargo, **no significa necesariamente que será menos eficiente**

En general, el sobre-costo es poquito... (determinación dinámica del tipo, métodos virtuales... )



# C++ : lo básico

hola.cpp

```
// Nuestro primer programa en C++
#include <iostream>
int main() {
    std::cout << "Hola , mundo" << std::endl;
    return 0;
}
```



# C++ : lo básico

Unas remarcas :

- La extensión de los archivos no es mas .c sino **.cpp** ; también se puede usar .C o .cc ; para los archivos *header*, se conserva .h
- Similaridades fuertes con C : función main, definición de las funciones, sintaxis, directivas preprocesador...
- Comentarios con “//” o “a la C”



# C++ : lo básico

Unas remarcas :

- El C++ introduce una nueva manera de manejar los flujos hacia archivos (aunque la manera a la C es también posible).
- Esta manera no se presenta como función sino como **operador** : es una de las grandes características de poder definir nuevos operadores, sobrecargar operadores existentes.



# C++ : lo básico

Unas remarcas : el C++ hereda también de los “malos” aspectos de C : manejoamiento de la memoria que responsabiliza al programador, no *Garbage collector* como en Java



# C++ : lo básico

Nuevas palabras reservadas :

- catch, try, throw : excepciones.
- delete, new : alocacion memoria.
- class : definición de clase.
- protected, private, public : diferentes tipos de membresía
- virtual, friend, inline : funciones virtuales, amigas o inline.
- template : patrones.
- namespace, using : espacios de nombre.
- operator : operadores.



# C++ : lo básico

Tipos en C++ vs. tipos en C :

- los tipos son los mismos que en C (enteros/floatantes) con la adición remarcable del tipo booleano. **bool** que toma como valor true o false (convertible en entero),
- las **enumeraciones** (enum) son mas estrictas (valor verificado, diferencia con int),
- los nombres de estructuras/Enums pueden ser utilizados tal cual sin necesitar un typedef.



# C++ : lo básico

hola.cpp

```
#include <iostream>
int main(int argc, char *argv[]) {
    bool condicion = (argc>1);
    if (condicion)
        std::cout << "Hola , " << argv[1] << std::endl;
    else
        std::cout << "Hola , mundo" << std::endl;
    return 0;
}
```



# C++ : lo básico

**Referencias:** es un elemento importante para el confort del programador que ha sido introducido en C++.

Una referencia es un alias sobre un nombre de variable que puede estar visto como apuntador constante y dereferenciado automáticamente (indirección automática!).

La ventaja: manipulas objetos/variables como si fuera por valor pero en realidad es por apuntador. Eso es por default en Java, y se puede usar en C++. Se terminan las pesadillas con los apuntadores... .



# C++ : lo básico

... bueno la verdad no se terminaron completamente. Perdón por las falsas esperanzas.



# C++ : lo básico

Se definen variables **muy cerca** de su uso efectivo; típicamente :

```
for (int i=0;i<10;i++)
    std :: cout << i << std :: endl;
```

Mas generalmente, se pueden definir dentro de los bloques de las estructuras de control...



# C++ : lo básico

Los mas que lleva C++

- Soporte OO : creación de clases...
- Espacios de nombre : contextos.
- Tipo booleano.
- Excepciones.
- Templates.
- Funciones inline.
- Referencias.
- Cast “funcionalizado”.



# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++
- 5 Tipos en C++



# Espacios de nombres

Una de las limitaciones mas desagradables con C : combinando librerías de varios tipos, varias fuentes, puede llevar conflictos sobre nombres, en particular **nombres de funciones** y **nombres de variables globales**; el problema se hace mas agudo con mas librerías !

En midi.h de midi.dll

```
int checkFile(const char *fileName);
```

En otralib.h de otralib.dll

```
int checkFile(const char *fileName);
```



# Espacios de nombres

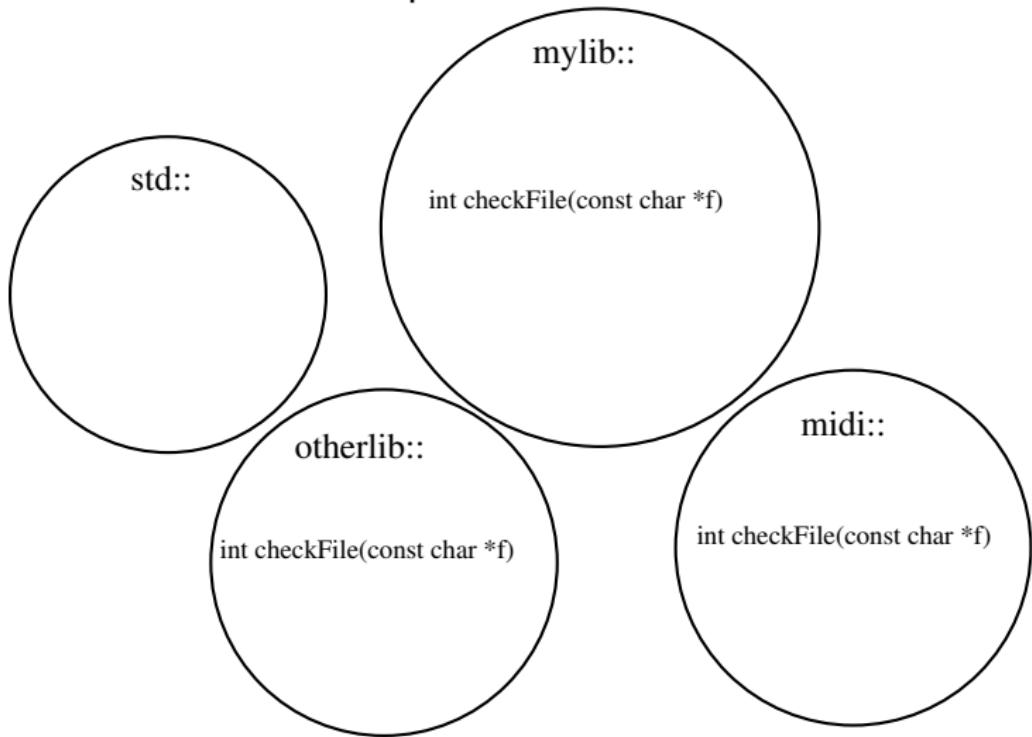
- Hay momentos en que no podemos hacer nada ya que no son librerías nuestras.
- Requiere atención y esfuerzos para no provocar conflictos.

En C++, hay un mecanismo para evitar esos líos : **espacios de nombres**, o **namespace**, que definen **unidades léxicas herméticamente separadas**.



# Espacios de nombres

No conflictos entre espacios distintos !



# Espacios de nombres

- Los espacios de nombres están **representados por simples nombres** (std,mylib...)
- Acceder a funciones, variables, estructuras definidas dentro de un *namespace*, usar el **operador “::”**,

```
int i=0;  
mylib :: checkFile(" myfile.txt" );
```

- Equivalente de package en Java

```
import java.awt.event.*;  
...  
ActionEvent myEvent = new ActionEvent();
```



# Espacios de nombres

En particular, para tener acceso a las funciones de la librería estándar (la *Standard Template Library*, por ejemplo), usar el *namespace std*:

```
std :: cout << "Me\u00fclcuro\u00f1en\u00f1Umecuaro" << std :: endl;
```

# Espacios de nombres

Hay manera de evitar que se tenga que repetir, dentro de un programa, el prefijo correspondiendo a un espacio de nombres. Consiste en usar la palabra llave **using** :

```
using namespace std;  
...  
cout << "Me\u00f1ecuro\u00f1en\u00f1Umecuaro" << endl;  
  
using namespace mylib;  
...  
checkFile(" myfile .txt" );
```

Vale para **todo el archivo de código** en que esta la palabra llave **using**.



# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++
- 5 Tipos en C++



# Output básico

Todos los mecanismos disponibles en C pueden estar utilizados en C++; ahora, sería **mas coherente usar los que propone C++ y/o quedarse coherentes** (no mezclar cout y printf).



# Output básico con iostream

El código siguiente ilustra el mecanismo muy diferente,

```
// Nuestro primer programa en C++
#include <iostream>
using namespace std;
int main() {
    cout << "Hola , mundo" << endl;
    return 0;
}
```

Mecanismos por **flujos** y operadores inspirados de pipes Unix (sentido intuitivo). El operador se “adapta” al tipo de argumento que se le pasa.

endl = regreso a la linea ('\n') + flush



# Output básico con iostream

Operador único para todos tipos de *output* :

```
double a;  
float b;  
int c;  
  
...  
cout << "Hola , „mundo“"  
     << 145 << " „ " << c  
     << " „ " << b << " „ " << a << endl;
```



# Output básico con iostream

Operador que, como todos operadores en C++, se puede **sobrecargar** (*overload*)

```
typedef struct myStruct_str {  
    ...  
} myStruct;  
...  
myStruct myStruct1;  
cout << myStruct1 << endl
```

Necesita dar una declaración y una definición del operador sobrecargado.



# Output formateado iostream

Para modificar el comportamiento estándar de iostream, se pasa palabras llaves como **operando** del operador “<<” (**manipulador**)

```
// Formateando outputs
cout << "En decimal:"
    << dec << 17 << endl;
cout << "en octal:" << oct << 17 << endl;
cout << "en hex:" << hex << 17 << endl;
```



# Output formateado iostream

Otros modificadores que permiten comportamiento similar al printf  
(precisión, tamaño de los números...)

```
// Formateando outputs
int i=17, j=2991;
cout << setw(10)
    << i << endl
    << setw(10)
    << j << endl;
cout << setprecision(9) << sqrt(2) << endl;
```

Necesita la inclusión

```
#include <iomanip>
```

Otro : setfill.



# Output formateado iostream

Método de formateaje : setf, unsetf

```
cout.setf ( ios::hex, ios::basefield );
cout.setf ( ios::showbase );
cout << 100 << endl;
cout.setf ( ios::dec, ios::basefield );
cout << 100 << endl;
cout.unsetf ( ios_base::showbase );
cout << 100 << endl;
```

Para indicar el formateaje corriente :

```
cout.setf(ios::showbase);
```

Otros : ios::showpos, ios::uppercase, ios::showpoint,  
ios::boolalpha.



# Output formateado iostream

- unos de los flags vienen por **grupos** y si los valores son mas de dos el simple setf puede no funcionar (habría que usar un unsetf con los otros flags),
- pero, como en el ejemplo, se puede especificar como segundo argumento para que grupo el flag se hace exclusivo,

```
cout.setf( ios::hex );
cout.setf( ios::hex, ios::basefield );
```



# Input iostream

El comportamiento es equivalente, con la misma inspiración de los *pipes* Unix, con un objeto-flujo cin, y un operador “>>”

```
int main() {  
    int n;  
    char c;  
    cout << "Enter a number: " ;  
    cin >> n;  
    cout << "Enter a char: " ;  
    cin >> c;  
}
```

el equivalente de stderr : cerr.



# Input iostream : verificar

Métodos eof(), fail(), good(), clear() para verificar *flags* sobre el flujo.

```
int data;
int total = 0;
cin >> data;
while (!cin.eof()) {
    if (cin.fail()) {
        cerr << "Not_a_number" << endl; cin.clear();
        string garbage;
        getline(cin, garbage); // se lee lo que queda
    } else
        total += data;
    cin >> data;
}
cout << "Total_is_" << total << endl;
```



# Input iostream : verificar

```
badbit    a fatal error has occurred  
eofbit    EOF has been found  
failbit    a nonfatal error has occurred  
goodbit    no errors have occurred
```



# Input iostream : caracteres

```
char ch;
```

```
cin >> ch; // Lee el proximo caracter no-espacio  
cin.get(ch); // Lee cualquier caracter
```

La segunda versión es no formateada.



# Manipulación de archivos

Proceso mucho mas simple que en C. Igual que con fprintf, las funciones de I/O sobre archivos se presentan de manera similar a las I/O estándar : objetos-flujos.

Requieren el uso de

**#include <fstream>**

# Manipulación de archivos

Para abrir un archivo en lectura, se crea un objeto ifstream, y se podrá usar operadores igualmente que con cin y cout.

```
ifstream in ("archivo.txt");
string palabra;
while(in >> palabra)
    cout << palabra;
in.close();
```

Dos cosas que notar acá : **constructor** de un objeto, y objetos especializados para manejar cadenas de caracteres, string.



# Primera paréntesis

Un constructor es un elemento fundamental del C++ : es un método un poco especial, de **inicialización** de un objeto, que permite usar expresiones como

```
tipo obeto(tipoparams1 params1);  
tipo obeto(tipoparams1 params1, tipoparams2 params2);
```

Posibilidad de **sobrecargar** el método (varias maneras de inicializar un objeto : con parámetros, a partir de nada, a partir de otro objeto del mismo tipo).



# Primera paréntesis

El constructor :

```
ifstream in("archivo.txt");
```

es equivalente a

```
ifstream in;  
in.open("archivo.txt");
```



# Constructor de ostream/istream

Uno de los constructores de fstream permite pasar modos de apertura del archivo (como el *flag* de fopen)

```
ios :: out  
ios :: append  
ios :: noreplace // non-standard
```

y se combinan

```
ofstream logFile("archivo.log",  
                 ios :: out | ios :: binary | ios :: append);
```



# Segunda paréntesis

El tipo `string` permite manejar mas fácilmente las cadenas de caracteres, con operadores mas intuitivos,

```
string s1=" Ejemplo_1" ;
string s2(" Ejemplo_2" );
cout << s1+" _y_" +s2<< endl;
```

Estructuras **dinámicas**, pero no hay que preocuparse de la memoria.



# Manipulación de archivos : getline()

La función `getline()` permite recuperar una linea de texto (terminada por un “`\n`”) desde un flujo, pasandole la cadena que se va a llenar

```
ifstream in("archivo.txt");
string s, line;
while(getline(in, line))
    s += line + "\n";
cout << s;
```

Remarca : se parece que pasamos la cadena por valor; es toda la ventaja de las **referencias** en C++ : se manipulan **como variables** pero corresponden a apuntadores !



# Manipulación de archivos

Copiar el contenido de un archivo a otro

```
ifstream in("archivo.txt");
ofstream out("archivocopy.txt");
string s;
while(getline(in, s))
    out << s << "\n";
```

Flujos en escritura con ofstream.



# Objetos iostream

No perder de vista que los iostream son **objetos** que instancian una **clase**; existen varios **métodos** (funciones específicas a este tipo de objetos) que permiten realizar operaciones con ellos:

```
istream& getline (char* s, streamsize n );
istream& getline (char* s, streamsize n, char delim );
int get();
istream& get ( char& c );
istream& get ( char* s, streamsize n );
istream& get ( char* s, streamsize n, char delim );
```



# Objetos iostream

- Las funciones de clases (métodos) vienen en general sobre-cargadas.
- Leer la documentación de cada clase, por ejemplo,  
<http://www.cplusplus.com/reference/iostream/iostream/>
- Llamar a un método :  
`objeto.método(argumentos);`



# Objetos iostream

Ejemplo : validez de un archivo

```
string fileName("archivo.txt");
ofstream archivo(fileName.c_str()); // Constructor

if (!archivo.good()) {
    cerr << "Error: el archivo " << fileName
        << " no se puede abrir" << endl;
}
```



# Objetos iostream

Ejemplo : control de datos

```
unsigned int dato;  
cin >> dato;  
if (cin.fail()) {  
    cerr << "Eso no es un entero positivo"  
        << endl;  
}
```



# Objetos iostream

Recordarse del uso de eof

```
char ch;  
ifstream fin( "temp.txt" );  
while( !fin.eof() ) {  
    fin >> ch;  
    cout << ch;  
}  
fin.close();
```



# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++ (Actual)
- 5 Tipos en C++



# De C a C++

Muy pocos cambios en el preprocesador : siguen las macros, los defines, la compilación condicional.

Una remarca (que vale también para C) : un identificador creado por un define **puede ser “quitado” por la directiva undef**, después de la cual ya no se usa.



# Include

Es la diferencia mas notable : en el caso de los archivos estándar incluidos, la sintaxis es un poco diferente, **la terminación en .h desaparece.**

```
#include <iostream>
```

Como dicho arriba, el espacio de nombres correspondiente es std.



# Include

Para funciones de la librería estándar C, se modifica,

```
#include <stdio.h>
```

por

```
#include <cstdio>
```

Se puede usar la forma antigua pero no tendrá las mismas propiedades (no *templates*, no espacio de nombre propio).



# Identificador \_\_cplusplus

Un identificador que estará definido cuando el código estará compilado con un compilador C++ : útil para hacer un código compilable en C o C++

```
#ifdef __cplusplus
extern C {
#endif
/* Archivo de definicion */
#ifndef __cplusplus
}
#endif
```



# Outline

- 1 Introducción al C++
- 2 Espacios de nombres
- 3 Mecanismos de input/output, archivos
- 4 Preprocesador C++
- 5 Tipos en C++



# Un nuevo tipo

Además de los tipos recuperados de C, se introduce el tipo **booleano**, muy útil para errores, condiciones... Todos los operadores de relación, de lógica ahora son bool (y ya no int).



# Posibilidades extendidas : struct

Métodos de estructuras :

```
typedef struct testStr {  
    int a;  
    int b;  
    int getA() { return a; }  
} test;  
test mystruct;  
cout << mystruct.getA();
```

No son tan útiles (porque eso lo hacen de manera mas propia las **clases**). De hecho las struct y las clases son casi lo mismo (idea de la diferencia?).



# Referencias

Noción introducida para aliviar un poco la tarea de manipular apuntadores, en particular para pasárselos a funciones :

```
int fonc(myStruct *str) {  
    ...  
    str->a=...;  
}  
myStruct mS;  
int res=fonc(&mS);
```



# Referencias

Hace que las ventajas de usar apuntadores (no se copian estructuras enteras en la pila, sino direcciones) están combinadas a las de pasar valores (no necesitas de razonar explícitamente en términos de dirección en memoria). De hecho, las funciones que toman referencias como parámetros se usan como si tomaran valores :

```
int fonc(myStruct &str) {  
    ...  
    str.a=...;  
}  
  
myStruct mS;  
int res=fonc(mS);
```



# Referencias

El valor del argumento no es copiado en la pila (caso de grandes estructuras); ahora puede ser que no queramos que este valor pueda ser modificado; en este caso, usaremos la palabra llave **const** :

```
int fonc(const myStruct &str) {  
    ...  
}
```



# Apuntadores o referencias ?

Apuntadores :

- Tipo en sí, y variables que apuntan hacia una entidad.
- Fuente de **problemas** : aritmética sobre apuntadores . . .

Referencias :

- Constantes ! Referencian siempre la misma cosa.
- Mas **seguro**.



# Apuntadores o referencias ?

Comparar :

```
int a=1,b=3;  
int &ref1=a,&ref2=b;  
int *pt1=&a,* pt2=&b;  
  
ref1 = ref2;  
pt1 = pt2;
```

