Tema 2: De C a C++. Introducción y conceptos.

Cualquier usuario de C puede reconocer una operación tan sencilla como i++, es decir, dada una variable i se aumenta en 1 su valor. En este sentido se puede ver el origen del nombre del C++.

En este tema no pretendemos hacer una comparación entre el C y el C++, sino que partiendo de conocimientos de C, vamos a ir mostrando, y al mismo tiempo obteniendo, la sintaxis y objetos C++.

En primer lugar debemos destacar que la extensión asociada a los ficheros C++ se denota por .cxx, .cpp o .c++. Aquí utilizaremos la notación .cxx.

Un aspecto importante de cualquier programa son los comentarios del mismo, puesto que permiten su mejor interpretación, y el intercambio con otros programadores. En C++ los comentarios se denotan por //, que afectan a toda la línea, aunque también se puede utilizar la notación /* ... */.

Declaraciones frente a definiciones

La declaración consiste en presentar el esqueleto de la variable o función, mientras que la definición consiste en asignar un valor o contenido.

```
int a;              int fun1(int x, float y);
int a = 7;          int fun1(int x, float y) { ... }
```

Incluir funciones conocidas

Para utilizar funciones desarrollas por otros programadores se utilizan los ficheros de cabecera (headers .h) que contienen las definiciones de las mismas.
En C++, al igual que en C, se utiliza la sentencia:

```
#include <ficheroheader>  (el compilador lo busca en el path)
#include “ficheroheader”   (el compilador lo busca en el directorio
                                desde donde se compila)
```

En C++ no se suele poner la extensión .h de los ficheros en el include, sino directamente el nombre del mismo.

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

#include <stdio>

El primer programa en C++

```
//Un test en C++
//
#include <iostream>
using namespace std;

int main() {
        cout << “Hola C++ ” << endl;
}
```

Para utilizar funciones de entrada salida se incluye el fichero de cabecera iostream, que incorpora funciones de entrada salida (cin y cout).

El cout permite mostrar por pantalla cualquier tipo de dato, pues el operador << está sobrecargado para ello. Del mismo modo el cin se utiliza para introducir datos con el operador >> también sobrecargado.

La línea que incluye el namespace se utiliza en C++ para, entre otras cosas que ahora mismo no son importantes, envolver el conjunto de funciones que se utilizan dentro de un entorno. En este caso estamos utilizando el entorno de las funciones estándar de C++, por eso la utilización del std.

A diferencia del C, en C++, la función principal o main() sólo puede ser de tipo int. Debemos tener en cuenta que uno de los secretos del C++ está en la declaración e identificación del tipo de datos. Posteriormente volveremos a retomar esta función.

La función endl acaba la línea y pasa a la siguiente. También podemos utilizar otros sentencias para texto: \t (tabulador), \n (nueva línea), etc.

Ejecución del programa

Utilizando un compilador GNU bajo linux:

g++ nombrefichero.cxx

que generará un ejecutable a.out. Si queremos obtener un ejecutable con un nombre determinado:

g++ -o holamundo nombrefichero.cxx

iostream

La función cout también permite devolver la información en determinados formatos (decimal, octal, hexadecimal, etc).

```
cout << “un número en decimal “ << dec << 15 << endl;
cout << “un número en octal “ << oct << 15 << endl;
cout << “un número en hexadecimal “ << hex << 15 << endl;
```

Ademas el formato de los diferentes tipos de datos es determinado de manera automática por la función cout, es decir, si es un char, un float, un int, etc.

La concatenación de los caracteres se hace uniendo con “ ”.

```
cout << “esto es una prueba” “ de como concatenar” << endl;
```

La función cin sirve para introducir parámetros desde la terminal:

```
int main() {
	int number;
	cout << “dime un número: “ ;
	cin >> number;
	cout << “el valor es: “ << number << endl;
}
```

Utilizando strings

El C++ ofrece una clase string diseñada para tratar con este tipo de datos de forma sencilla. Para ello sólo debemos incluir el header <string>. Debemos utilizar el namespace std para poder hacer uso de la misma.

```
// Test con string
#include <iostream>
#include <string>
using namespace std;

int main() {
	string s1, s2; // declaración de dos strings.
	string s3 = “Hola, puedo ser asignado”;
```

```
        string s4("y también así");
        s1= s3 + " " + s4; // se puede utilizar el +
        s1 += "7"; // y anexionar cosas.
}
```

Leer y escribir ficheros.

Para poder tratar con ficheros de entrada y salida hay que utilizar: <fstream> e <iostream>.

Abrir un fichero en modo lectura se hace utilizando el objeto ifstream, que luego se utiliza como el cin. Para abrir un fichero en modo escritura se utiliza el ofstream, que luego se trata como el cout.

Una de las funciones más útiles del iostream es el getline(), que permite leer una línea (acabada por un salto de línea) en un objeto string. El primer argumento es el objeto a leer y el segundo el string donde se almacena la información leida.

```
// test del getline, ifstream, ofstream
//
#include <string>
#include <fstream>
using namespace std;

int main() {
        ifstream in("ficheroleer.txt");
        ofstream out("filesalida.txt");
        string s;
        while(getline(in,s)) // descarta el caracter de nueva línea
          out << s << "\n";
}
```

También se puede almacenar todo el fichero en un string, puesto que este objeto presenta muchas herramientas de búsqueda y operaciones con strings.

Utilización de vectores.

El objeto string está bien para almacenar elementos, pero es sólo un contenedor no estructura o seccionado. C++ dispone de un objeto muy útil para solventar este problema: vector< >.
Para su utilización debemos incluir el <vector>.

```
//
//test con vector y muestra de algunas de sus utilidades.
//
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
int main() {
        vector<string> v;
        ifstream in("fichero.txt");
        string line;
        while(getline(in, line))
           v.push_back(line); // método que añade contenido de line.
        for(int i = 0; i < v.size(); i++)
           cout << i << ": " << v[i] << endl;
}
```

Vamos leyendo las líneas del fichero y almacenándolas en elementos del vector v.

También se podría almacenar cada palabra en el vector, utilizando in >> line, que coge cada palabra del objeto in y la almacena en line.
O bien se puden asignar valores a cada elemento de v:

```
        v[3]="hola";
```

El objeto vector también se puede utilizar con cualquier otro tipo de datos: vector<int>, vector<float>, etc.

Funciones que devuelven valores.

Para devolver un valor desde una función se utiliza el comando return:

```
char func(int i) {
        return  'a';
}

int main() {
        int val;
        cin >> val;
        cout << func(val) << endl;
}
```

Sentencias de control en C++.

if-else

```
if(expresión)        if(expresión)
  ejecuta;               ejecuta1;
                       else
                           ejecuta2;
```

while

```
while(expresión) {
        ejecuta;
}
```

do-while

```
do {
      ejecuta;
} while (sentencia);
```

for

```
for(int i = 0; i < maximo; i++) {
        sentencias(i);
}
```

break/continue
Sale del bucle / vuelve a comenzarlo.

switch

```
switch(selección) {
        case valorentero1 : sentencia ; break
        case valorentero2 : sentencia2; break
        .....
        default: sentencia por defecto;
}
```

Introducción a operadores

Las operaciones aritméticas tienen preferencias de ejecución. Algunos símbolos se pueden utililizar a la hora de decrementar o incrementar variables:

i++ = i +1;
++i = (i+1); (se realiza primero la operación).
i-- = i-1;
--i = (i-1); (se realiza primero la operación).

Introducción a los tipos de datos

En C++ se introduce la posibilidad de que cada programador pueda definir su tipo de datos (objetos) y de ahí su gran importancia. Al igual que en C, existen los tipos típicos: int, float, double, etc, cuyo tamaño en memoria depende de la máquina (SO). Para calcular el espacio que ocupa en memoria un tipo de dato se utiliza la función sizeof() dentro del <iostream>.

Debemos mencionar también el tipo de datos bool, que puede tener dos estados: true (entero 1) o false (cero). Este tipo de datos puede aparecer como resultado de comparaciones: &&, ||, !, >, <, etc.

Para los tipos de datos clásicos también existen los especificadores: long, short, signed y unsigned.

Introducción a los punteros.

Los punteros son un concepto ya conocido del C. Básicamente son accesos directos a los datos en memoria (dirección y contenido).
El operador & delante de un dato nos da la dirección de memoria en la que está ese dato o incluso una función. Así en

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

void f(int a) {
	cout << “el valor de entrada es: “ << a << endl;
}

int main() {
	int s;
	cout << (long)&f << endl; // dirección de memoria de f
	cout << (long)&s << endl; // dirección de memoria de s
}
```

Si ejecutaramos este programa, comprobaríamos que las direcciones de memoria de las variables dentro de main() están en regiones bastantes diferentes de las variables de fuera, o la misma

función f.

El puntero nos permite acceder directamente a las direcciones de memoria y modificarlas. El símbolo típico para declarar un puntero es el *, y al igual que las variables o las funciones posee un tipo:

int* a; // puntero tipo entero llamado a.

Aunque C++ admite la notación int* a ó int *a, creemos que es conveniente usar la primera, puesto que indica algo como: intpointer.
Del mismo modo se podría decir para el resto de tipos de datos.

También debemos saber que su declaración es muy sensible a la sintaxis, esto es:

int* a, b,c; //indica un puntero tipo entero a, y dos variables enteras.

En vez de

```
int* a;
int* b;
int* c;
```

La utilidad de los punteros subyace en que contienen una dirección de memoria. Así:

```
int a = 47;
int* ipa = &a; // ipa contiene la dirección de a.
```

Para acceder al contenido de la dirección de memoria se utiliza: *ipa, y para cambiarlo *ipa=100;

Luego, básicamente los punteros pueden ser utilizados:
- Para cambiar objetos externos a una función.
- En técnicas avanzadas de programación que si el tiempo lo permite serán tratadas en este curso.

A continuación nos ocuparemos de la primera utilidad señalada anteriormente.
Generalmente cuando se pasa un argumento a una función, se hace una copia del mismo en el cuerpo de la función. A esto se le llama evaluación por paso de argumento:

```
//
// Test con funciones y punteros.
#include<iostream>
using namespace std;

void f(int a) {
        cout << “a: “ << a << endl;
        a = 5;
        cout << “a: “ << a << endl;
}

int main() {
        int x=47;
        cout << “x= “ << x << endl;
        f(x);
        cout << “x= “ << x << endl;
}
```

En f() la variable a es una variable local a la función, y cuando llamamos a f() dentro de main() con la variable x, estamos haciendo una copia de x en a, pero sin cambiar x.
De alguna manera, cuando llamamos a la función f(), no estamos afectando a variables u objetos externos a ella, pues son variables diferentes con direcciones de memoria diferentes.
Entonces, para poder actuar sobre objetos o variables externas a las funciones podemos utilizar los punteros a éstos. Es decir, en el ejemplo anterior podemos poner:

```
//
//Test utilizando funciones con paso por puntero.
//
#include <iostream>
using namespace std;

void f(int* p) {
        cout << “p= “ << p << endl;
        cout << “*p = “ << *p << endl;
        *p = 5;
        cout << “p= “ << p<< endl;
}

int main() {
        int x=47;
        cout << “x= “ << x <<endl;
```

```
        cout << “&x= “ << &x << endl;
        f(&x);
        cout << “x = “ << x << endl;
}
```

Ahora como lo que pasamos a la función es la dirección de memoria de la variable x, y la función tiene como argumento de entrada un puntoro a esa dirección, las operaciones dentro de la función cambian el valor de la variable externa a f(), x.

Como podemos comprobar, la utilización de los punteros nos facilita la realización de operaciones sobre objetos externos a las misma. Pero además C++ introduce una nueva y más efectiva manera de introducir los parámetros en las funciones: pasos por referencia, que para el programador es una manera mixta de trabajar y a mi entender mucho más sencilla de interpretar.

```
//
// Test con paso por referencia a las funciones.
//
#include <iostream>
using namespace std;

void f(int& r) {
        cout <<”r = “ << r << endl;
        cout << “&r= “ << &r << endl;
        r=5;
        cout << “r = “ << r << endl;
}

int main() {
        int x=47;
        cout << “x= “ << x << endl;
        cout <<”&x= “ << &x <<endl;
        f(x); // la llamada es parecida al paso por valor:cool!!
        cout << “x= “ << x << endl;
}
```

Además dentro de la función la variable r se trata como una variable normal y se le asigna valor de la misma manera. Sin embargo, como lo que se le ha pasado a la función f() es una referencia, ésta afecta al contenido de la variable externa: ambas son la misma región de memoria pero con etiquetas diferentes.

Se han mostrado ejemplos con puntero a tipos int, pero como es de esperar también se pueden utilizar con el resto de los tipos de

datos. Dentro de éstos es de destacar el puntero tipo void, que indica que el puntero puede apuntar a cualquier tipo de dato.

```
//
// Test con void.
//
#include <iostream>
using namespace std;

int main() {
        void* v;
        char c;
        int i;
        float f;
        double df;
        v=&c;
        v=&i;
        v=&f;
        v=&df;
}
```

Este tipo de dato tiene la misma interpretación para las funciones. Sin embargo, hay que tener en cuenta que es muy peligroso y su utilización debe ser con conocimiento de causa. Por ejemplo en el caso anterior si luego queremos operar con el contenido del puntero debemos asignar un tipo de dato, hacer lo que se llama un cast del dato: (int*)v.

Ámbito de acción de las variables.

El ámbito de definición de las variables viene dado por el número de {} que la encierren o tenga por debajo. Una variable declarada al principio de la función main() está definida dentro de todas las {} que estén jerárquicamente por debajo de ella.

Las variables que se definen en las declaraciones de las funciones de contro (for, while, etc) son propias de la función de control, siempre que se declaren en ella, y por tanto, no están definidas fuera de ellas. Ejemplo típico es la múltiple utilización del int i como contador. Esta es una de las grandes ventajas y comodidad del C++.

Se llaman variable globales a aquellas que se definen para el programa principal o función principa y que son utilizadas por otras funciones localizadas incluso en otros ficheros. En el caso que

se utilicen variables globales en ficheros diferentes a donde se definan, se le debe decir al compilador mediante la sentecia extern:

extern int variableglobal;

Esto avisa al compilador que existe una variable definida en cualquier fichero del software en desarrollo. Este mismo razonamiento puede ser aplicado para definir tipos de funciones.

La utilización de las variables globales debe ser cuidadoso, pues puede ocasionar verdaderos quebraderos de cabeza. Para evitar esto se suelen utilizar variables locales a las funciones, que tiene su ámbito definido en el propio cuerpo de la función. Normalmente las variables locales a una función se crean cuando comienza la ejecución de la función y se destruyen al finalizar. Cuando se vuelve a llamar la función vuelve a suceder lo mismo. Sin embargo, si uno quiere que el valor de la variable permanezca a lo largo de la vida del programa, se puede utilizar la sentencia static e incializarla. Cada vez que la función se ejecute el valor de la variable permanece en memoria (incluso si ha cambiado), digamos que la función “recuerda” la información contenida en la variable entre llamadas a la misma. La ventaja de esta declaración es que el ámbito de la variable es dentro del cuerpo de la función, y no puede ser cambiada fuera, como podría ocurrir con una variable global.

```
//
// Test utilizando static
//
#include <iostream>
using namespace std;

void f() {
        static int i = 0;
        cout << i << endl;
}

int main() {
        for (int x=0; x < 10; x++)
           f();
}
```

cada vez que se llama a f(), el valor de i cambia. Si no fuera static el valor de i sería 1.
Otro significado de static es el de reducir su ámbito de acción. Si

una función o variable fuera de todo cuerpo de función es declarada static, estamos diciendo al compilador que esa función/variable es local al fichero que la contiene.

Generación de nuevos tipos de datos.

Los tipos de datos existentes son útiles en C++, pero no suficientes para abarcar sus prestaciones. En particular el concepto de clase en C++ es fundamental, y precisamente se basa en generar nuevos tipos de datos.

La palabra clave más importante para generar nuevos tipos de datos es typedef, que actúa como un alias.

```
	typedef unsigned long ulong;
	typedef int* Intpuntero; // esta permite utilizar Intpuntero x, y;
```

Sin embargo typedef es verdaderamente útil cuando tratamos con tipos u objetos struct. El objeto struct es una manera de almacenar diferentes tipos de datos en una región de memoria, y define un nuevo tipo de dato:

```
//
// Test con struct
//
struct Estructura1 {
	char c;
	int i;
	floa f;
	double d;
}; // importante acabar con “;”.

int main() {
	struct Estructura1 s1, s2;
	s1.c = 'a'; // forma de acceder a los componentes de s1.
	s1.i=1;
	s1.f= 3.14;
	s1.d=0.000017;

	// Lo mismo con s2.
}
```

Pero suel ser más cómodo declararlo como:

```
typedef struct {                        typedef struct Estructura1 {
    char c;
    int i;
    float f;            o bien poner
    double d;
} Estructura1;                          } Estructura1;

int main() {
  Estructura1 s1;
   .....
   .....
}
```

Del mismo modo también podemos definir punteros a este nuevo tipo de datos: punteros a estructuras. Esta es la base o el inicio de la generación de clases en C++, al que iremos convergiendo a medida que avance el curso.
Al igual que con la estructura se accede con "." a cada componente, con el puntero se accede utilizando "->".

```
//
// Test con los punteros a estructuras.
//
typedef struct Estructura1 {
        char c;
        int i;
        float f;
        double d;
}

int main() {
        Estructura1 s1, s2;
        Estructura1* sp = &s1;
        sp->c = 'a';
        sp->i=1;
        sp->f=3.14;
        sp->d=0.0007;
}
```

Arrays.

Es un tipo de composición que permite almacenar un conjunto de

variables de un mismo tipo.

```
int a[10]; // 10 elementos tipo int.
```

y podemos asignar el valor a cada componente, comenzando en 0.

```
a[5]=7;
```

Como veíamos al principio de este capítulo, la forma de acceso a arrays es muy rápida (recordemos la clase vector<>), y almacena su contenido de forma continua en memoria.

Cuando asignamos un puntero a un array, éste apunta al primer elemento del array, y no podemos cambiar el identificador de memoria del array, sino su contenido y para movernos por el array.

```
// Test arrays.

int main() {
        int a[10];
        int* p=a; // apunta a a[0], o sea &a[0]
        for(in i =1; i<10;i++)
            p[i]=i*10;
}
```

Que un puntero a un array apunte a su primer elemento es importante para cuando queremos pasar un array a un función. Si declaramos un array como un elemento de entrada es como declarar un puntero como elemento de entrada:

```
//Test con arrays en funciones.
//
#include<iostream>
#include<string>
using namespace std;

void func1(int a[], int size) {
        for(int i =1; i < size; i++)
                a[i]=i*i-i;
}

void func2(int* a, int size) {
        for(int i =1; i < size; i++)
                a[i]=i*i-i;
}
```

```
void print (int a[], string name, int size) {
        for(int i =1; i < size; i++)
          cout<< name<<"["<<i <<"]" <<"="<< a[i] << endl;
}

int main() {
        int a[5], b[5];
        // valore aleatorios.
        print(a, "a", 5);
        print(b,"b",5);
        //
        func1(a,5);
        func1(b,5);
        print(a, "a", 5);
        print(b,"b",5);
        //
        func2(a,5);
        func2(b,5);
        print(a, "a", 5);
        print(b,"b",5);
}
```

Como podemos comprobar sale lo mismo, que demuestra que como argumento de entrada a funciones se puede usar int a[] ó int* a.

Un caso interesante de este paso de parámetros a funciones es el de la función main() que en realidad es:

int main(int argc, char* argv[]) {}

con argc número de argumentos de entrada y argv[] los propios argumentos. Se debe tener en cuenta que los argv[] es un array de punteros tipo char y que por tanto si queremos trabajar con los valores introducidos desde consola debemos hacer la transformación a tipos de datos operables: atof(), atoi().

nota: el argumento argv[0] es el propio programa a ejectuar.

Operaciones con punteros.

De alguna manera en la sección anterior hemos mostrado que los punteros son alias de los arrays. Sin embargo, los punteros son más flexibles a la hora del manejo del contenido de los arrays, pues

podemos modificarlos para apuntar a caulquier parte del array (sin cambiar el identificador del array).

Con operaciones de punteros nos referimos a la aplicación de operaciones aritméticas con estos elementos.
Quizás la mas interesante es:

```
int d[10];
int* p = d;
p++; //se desplaza al siguiente elemento del array.
```

Y lo mismo pasaría con otros tipos de datos o arrays de struct.
También se pueden utilizar las operaciones --, + y -, donde estos dos últimos tiene menor sentido: no se pueden sumar dos punteros (se utiliza para desplazarlo tanta posiciones como quiera) y la diferencia da el número de elementos entre ambos.

Debugging.

Se llama debugger o debbugging a todas aquellas técnicas que nos permiten chequear la existencia de errores de ejecución en nuestros programas.

nota: viene de la palabra “bug” bicho, por la anéctoda con el primer ordenador que existió.

De alguna manera son “marcas” que se ponen en el código fuente para poder hacer un seguimiento de las secuencias de ejecución.

En primer lugar podríamos hablar de los métodos que se utilizan con el preprocesador: #define, #ifdef ..#undef

#define DEBUG // posiblemente en el fichero .h, cuidado con NDEBUG (reservado en C)

```
#ifdef DEBUG //chequea si esta definido.
    Código de control.
#endif // finaliza la ejecución.
```

Algunos compiladores permiten ejecutar las sentencias #define y #undef desde consola o llamada al mismo.

También se pueden utilizar debuggers en tiempo de ejecución del programa:

```
//
bool debug = false;

int main(int argc, char* argv[]) {
        for(int i= 0; i < argc;i++)
            if(string(argv[i]) == “--debug=on”)
               debug = true;
        bool go= true;
        while(go) {
          if (debug) {
               codigo de control
          } else {
               no se hace debugging.
          }
}
```

Hay una manera de evitar el tener que escribir todo la información (contenido de las variables o salidas a funciones), utilizando el “#”. Cuando se utiliza delante de un argumento en una defición al preprocesador, éste convierte el argumento en un array de caracteres.

```
#define PR(x) cout << #x << “= “ << x << endl;
```

y también se puede utilizar con expresiones:

PR(a+b), PR(a/b), etc.

Existe también una función útil para testear el código. Se trata de la función assert() que da un mensaje y para la ejecución si su argumento no es verdadero. Para evitar que se ejecute se pone #define NDEBUG al principio de todo el programa.

```
int i = 100;
assert(i != 100); // para la ejecución
```

Makefile: Compilación conjunta.

Cuando se utilizan código de muchas personas o bien muchos fichero separados se hace costoso el estar compilando cada uno de ellos y linkándolos. Para evitar esto, se ha desarrollado la utilidad de compilación conjunta o gestionada make.
Básicamente, la idea es generar un archivo texto donde se indican,

con cierta sintaxis, las tareas a realizar y que el compilador sólo hará si es necesario.

```
#Esto es un test para hacer un Makefile.

#Comentarios.
#Macros: definiciones de flags, que se llaman con $
CXX = g++
OFLAG = -o
#Definicion de las extensiones de los ficheros a chequear.
.SUFFIXES : .o .cxx

# el simbolo $< indica cualquier requisito necesario "el fichero que necesita ser compilado"
.cxx.o:
	$(CXX) $(CXXFLAGS) -c $<
all: \
	test \
	testArray \
	testdebug
clean:
	rm -f *.o
	rm -f test testArray testdebug

test: testvector.o
	$(CXX) $(OFLAG) test testvector.o

testArray: testArray.o
	$(CXX) $(OFLAG) testArray testArray.o

testdebug: testdebug.o
	$(CXX) $(OFLAG) testdebug testdebug.o

testvector.o: testvector.cxx
testArray.o: testArray.cxx
testdebug.o: testdebug.cxx
```