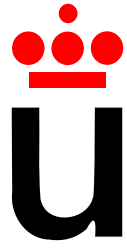
1. Introducción a C++

Julio Vega

julio.vega@urjc.es



Universidad
Rey Juan Carlos

Contenidos

```
/* ---------------------------
   File: helloworld.cpp
   Author: Julio Vega
   Date: 22/07/14
   Goal: this program shows the basics about C++
--------------------------- */
#include <iostream> // library to output data onto the screen

int main() { // function main begins program execution
   std::cout << "Hello world in C++!\n"; // show message
   return 0; // indicate that program ended successfully
} // end function main
```

- Editor recomendable: *Gedit*. Muy extendido, sencillo pero potente.
- Instalar compilador: `sudo apt-get install build-essential`
  - GCC = GNU project C and C++ compiler.
- Compilación: `g++ -o helloworld helloworld.cpp`
  - `-o`: *out*, permite indicar el nombre del ejecutable de salida.
    - Si no se indica, se crea ejecutable por defecto: *a.out*.
- Ejecución: `./helloworld`

- Todo fichero de código debe incluir una cabecera descriptiva.
- La línea `#include <iostream>` es una directiva del preprocesador.
  - Toda línea que comienza por # son procesadas por el preprocesador.
    - El preprocesador es aquel que se ejecuta antes de compilar el programa.
  - En este caso, se indica que debe incluir el contenido de `iostream`.
    - Esta es una librería que permite mostrar/recibir datos del I/O estándar.
    - E.g. para usar `cout` (salida por pantalla).
- Los comentarios de una línea comienzan con //
  - Si tienen más líneas, empiezan con /∗ y terminan con ∗/
- La función `main` comienza la ejecución del programa.
  - Un programa solo puede contener una función `main`.
  - Esta función devuelve un entero (`int`).
- Todo bloque de código va entre llaves ({ y }).
- Toda instrucción acaba en ; excepto las directivas del preprocesador.

- La línea de `std::cout` imprime el mensaje entrecomillado.
  - Está mandando (`<<`, *pipe*) el mensaje a la salida estándar (`std::cout`).
  - `std::` indica que `cout` pertenece al espacio de nombres de *std*.
    - Otros flujos estándar son: `cin` (entrada) y `cerr` (error).
  - \n = \(carácter de escape) + `n`(*new line*) = salto de línea.
    - Otros c. escape: \t (tab), \r (retorno carro), \a (alert sound), \', \".
- La última línea, `return 0;`, indica que el programa termina con éxito.
- Sangría no necesaria (Python sí) pero sí obligatoria en la asignatura.
  - Tamaño uniforme, obligatorio insertar espacios, no tab (config. editor).
- Idioma recomendable: inglés. Reduce tamaños y facilita reutilización.

```cpp
#include <iostream>

int main() {
  std::cout << "Welcome back ";
  std::cout << "to C++!\n";

  std::cout << "Welcome back\nto\n\nC++!\n";
  return 0;
}
```

```
#include <iostream>

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  total = num1 + num2;
  std::cout << "The total is " << total << std::endl;

  return 0;
}
```

```
#include <iostream>

int getTotal (int a, int b){
  return a + b;
}

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  std::cout << "Total: " << getTotal(num1, num2) << std::endl;
  return 0;
}
```

library.h

```
int getTotal (int,int);
```

library.cpp

```
#include "library.h"

int getTotal (int a, int b){
  return a + b;
}
```

main.cpp

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

int main() {
  int num1;
  int num2;
  int total;

  std::cout << "Write the first integer: ";
  std::cin >> num1;

  std::cout << "Write the second one: ";
  std::cin >> num2;

  std::cout << "Total: " << getTotal(num1, num2) << std::endl;
  return 0;
}
```

- Este es un fichero de texto plano que contiene **órdenes**.
  - Las órdenes son ejecutadas por la utilidad `make` siguiendo reglas.
    - Al ejecutar `make` desde el Terminal, busca el fichero *Makefile*.
  - **Explícitas**: dan instrucciones a `make` para construir fich. indicados.
    - `main.o: main.cpp library.h`
    - `>$(CC) $(CFLAGS) -c main.cpp`

      Se lee: para construir el fichero main.o se requiere main.cpp y library.h.
      Y se le da la orden para que make pueda construir ese fichero...
  - **Implícitas**: instrucciones generales a seguir si ∄ regla explícita.
- Las instrucciones de las reglas han de estar debidamente indentadas.
  - Lo habitual para la indentación es usar el tabulador.
  - Si se usa un *indentador* diferente, hay que especificarlo.
    - En este caso se ha establecido el caracter `>` para indentar.
    - Para ello se establece la variable de entorno: `.RECIPEPREFIX = >`
- También puede contener **comentarios**, precedidos por el símbolo `#`.

```
#=======================================
# Example of Makefile
#=======================================

CC = g++
CFLAGS = -Wall -g
.RECIPEPREFIX = >

main: main.o library.o
>$(CC) $(CFLAGS) -o main main.o library.o

main.o: main.cpp library.h
>$(CC) $(CFLAGS) -c main.cpp

library.o: library.h

clean:
>rm *.o main
```

- Las tres variables se podrían declarar en una sola línea.
  - E.g.: `int num1, num2, total;`
  - Esto reduce la legibilidad e impide poner comentarios individuales.
- Recomendable poner espacio tras coma para aumentar legibilidad.
- Todos los identificadores van en minúscula y notación camello.
  - Excepto las constantes, que se escriben todo en mayúscula.
  - C++ es *case sensitive*, por lo que: `num1` $\neq$ `Num1`.
- La notación camello consiste en poner mayúscula cada palabra anexa.
  - En lugar de separarlas por guiones bajos, algo desaconsejable.
  - E.g.: `getTotal()`, son dos palabras (*get* + *total*).
    - Otro ejemplo: `getMaxGradeOfSubject()`.
- Las funciones, además, deben incluir verbo: `int getTotal ();`

- IDs cualquier longitud;recomendable 31 o menos (↑ compatibilidad).
- Nombres de variables que sean significativos (e.g. `counter` VS. `c`).
- Evita IDs que comiencen por _; GCC puede usar nombres así.
- Evita usar palabras muy simbólicas; C++ podría integrarlo en futuro.
  - Hoy podrías usar `object`; mañana C++ podría usarla como *keyword*.
- Deja línea en blanco entre la declaración de variables e instrucciones.
- Coloca espacios a ambos lados de un operador binario (e.g. `a = 2;`)

- Uso de `cin >> myValue;`
  - Para leer enteros: `cin >> myInt;`).
  - Para leer caracteres: `cin >> myChar;`). Si $+1$ char $\Longrightarrow$ *overflow*.
- Uso de `myValue = cin.get()`
  - Para leer enteros: `myInt = cin.get()`. Vierte su código ASCII.
    - Para obtener el ASCII de un char: `static_cast<int>('2')`.
  - Para leer caracteres: `myChar = cin.get()`. Vierte el mismo char.
- Leer cadenas de caracteres (sin usar `string`, de librería `string`).
  - Se puede usar `cin >>`, pero divide la cadena por espacios.
  - O mediante array de chars, con: `cin.getline (myString, long);`

```
char myChar; int myInt; char myString[200];

cout << "Write a char: " << endl; // READING WITH "cin >>"
cin >> myChar; // supposed to be read '2'
cout << "Read: " << myChar << endl << endl; // shows '2'
cout << "Write an int: " << endl;
cin >> myInt; // supposed to be read 2
cout << "Read: " << myInt << endl << endl; // shows 2

cout << "Write another char: " << endl; // READING "cin.get()"
myChar = cin.get (); // supposed to be read '2'
cout << "Read: " << myChar << endl; // shows '2'
cout << "Write another int: " << endl;
myInt = cin.get (); // supposed to be read '2'
cout << "Read: " << myInt << endl << endl; // shows 50

cin.getline (myString, 200); // READING A CHAR ARRAY
cout << "cadena: " << myString << endl;
```

- Algunos caracteres especiales a leer son EOF (End Of File).
  - Este caracter, en sistemas UNIX/Linux, se escribe con *CTRL+d*.
    - En UNIX/Linux, *CTRL+c* cancela proceso; *CTRL+z* lo deja zombie.
  - En Windows, EOF se introduce mediante *CTRL+z*.
- Otro caracter especial es el caracter nulo ('\0').
  - Antes había que ponerlo manual/ al final de cada cadena de char.
  - Ahora ya no es necesario porque lo pone el compilador automática/.
    - E.g. `char greeting[6] = 'H', 'e', 'l', 'l', 'o', '\0';`

```
#include <iostream>

int main () {
  int number;
  bool exit = false;
  do {
    std::cout << "Write a number (EOF to exit): " << std::endl;
    if ((number = std::cin.get()) == EOF) exit = true;
    std::cin.ignore(); // try with this line commented... errors!
  } while (!exit);   // due to how cin works (splits on spaces)
  return 0;
}
```

Comunes de C y C++

```
auto break case char const continue default do double else
enum extern float for goto if int long register return
short signed sizeof static struct switch typedef union
unsigned void volatile while
```

De C++

```
and and_eq asm bitand bitor bool catch class compl
const_cast delete dynamic_cast explicit export false friend
inline mutable namespace new not not_eq operator or or_eq
private protected public reinterpret_cast static_cast
template this throw true try typeid typename using virtual
wchar_t xor xor_eq
```

- Básicos: $+$, $-$, $*$, $/$
- Módulo: %. Vierte el resto de una división entera.
- Potencia: función `pow`. C++ da errores con operadores como $**$ o ^.
- Paréntesis: sirven para agrupar subexpresiones (e.g. `a * (b + c)`).
- Orden precedencia: 1.º paréntesis, 2.º $*$, $/$ o %, 3.º $+$ o $-$.
- El uso redundante de paréntesis $\uparrow$legibilidad expresiones complejas.

- Op. relacionales: $>$, $<$, $>=$, $<=$
- Op. de igualdad: $==$, $!=$
- En operadores que contengan dos símbolos, estos han de estar juntos.
- Invertir los símbolos de un op. supone un error sintáctico.
  - Y, en el peor de los casos, un error lógico. E.g. $=!$ en lugar de $!=$.
- Confundir $==$ por $=$ genera error lógico ($\uparrow\uparrow$difícil detectar).

- Op. básico de asignación (=): `i = i + 3;`
- Op de asignación de suma (+=): `i += 3;`
  - E igualmente: `-=`, `*=`, `/=`, `%=`.
- Operadores unarios ofrecidos por C++ para sumar y restar.
  - Postincremento y postdecremento (usa valor actual y luego modifica).
    - `i++;` o `i--;`
  - Preincremento y predecremento (modifica y luego usa nuevo valor).
    - `++i;` o `--i;`

```
int i;
i = 5;
cout << i << endl; // shows 5
cout << i++ << endl; // shows 5 and increment 1
cout << i << endl; // shows 6
// ----------------------------------------
i = 5;
cout << i << endl; // shows 5
cout << ++i << endl; // increment 1 => shows 6
cout << i << endl; // shows 6
```

- Necesarios para evaluar varias condiciones y tomar una decisión.
- Operadores: && (AND), || (OR), ! (NOT).

```
if (woman && age >= 65) oldWomen++;

if ((practicalExercises < 4) || (practicalFinalExam < 4))
  failSubject = true;

std::cout << std::boolalpha << "Fail subject = " << failSubject;
```

- Buenas prácticas: aprovecha los cortocircuitos por eficiencia.
  - En AND, la condición con más prob. ser `false`, mejor a la izquierda.
  - En OR, la condición con más prob. ser `true`, mejor a la izquierda.
- `boolalpha` es un manipulador de flujo *pegajoso*.
  - Hace que los valores `bool` se impriman como `true` o `false`.
  - *Pegajoso*: se queda activado para el resto del programa.

- Todo problema puede resolverse ejecutando ciertas acciones en orden.
- Algotirmo: acciones a ejecutar + orden en que se ejecutan estas.
- Ejecución secuencial: instrucciones se ejecutan una después de otra.
  - VS. *goto* (1960): permite transferencia de control a cualquier destino.
- Programación estructurada: secuencia + selección + repetición.
  - Supuso todo un reto cambiar a este nuevo paradigma (fin del *goto*).
- Instrucciones selección: `if`, `if...else` y `switch`.
- Instrucciones repetición: `while`, `do...while` y `for`.
- Todo programa C++ se puede hacer con estas 6 inst. + secuencias.
  - Combinando inst. control en dos formas: apilamiento + anidamiento.

Instrucción de selección doble `if...else`

```
if (studentGrade >= 5)
  cout << "Passing grade";
else
  cout << "Failing grade";
```

- C++ ofrece tipo de datos `bool`, con posibles valores: `true` o `false`.
  - También se pueden usar valores enteros: 1 <=> `true` y 0 <=> `false`.
  - Usar valores enteros ↑ compatibilidad con versiones anteriores.
- Se han omitido las llaves de apertura y cierre en bloques `if` y `else`.
  - Esto es posible porque ambos bloques solo contienen una instrucción.
  - Las llaves son solo obligatorias si bloque contiene varias instrucciones.
  - Pero úsalas en caso de ambigüedad o para evitar errores lógicos.

Operador condicional ? . . . :

```
studentGrade >= 5 ? cout << "Passing grade" : cout << "Failing
    grade";
```

- Se lee: *Si calificación >= 5, entonces aprobado, si no, suspenso*.

Instrucciones `if...else` anidadas

```
if (studentGrade >= 9)
  cout << "A grade";
else
  if (studentGrade >= 8)
    cout << "B grade";
  else
    if (studentGrade >= 7)
      cout << "C grade";
    else
      if (studentGrade >= 6)
        cout << "D grade";
      else
        cout << "F grade";
```

- Una instr. `if...else` anidada es más rápido que varios `if` simples.
  - Con anidamiento se puede salir antes de tiempo al cumplirse condición.

Instrucciones `if...else` anidadas (forma popular)

```
if (studentGrade >= 9)
  cout << "A grade";
else if (studentGrade >= 8)
  cout << "B grade";
else if (studentGrade >= 7)
  cout << "C grade";
else if (studentGrade >= 6)
  cout << "D grade";
else cout << "F grade";
```

- Esta sintaxis es idéntica a la anterior, excepto por espaciado y sangría.
- Esta forma es más usada; evita usar mucha sangría y ↑ legibilidad.

El problema del `if` ignorado

```
if (studentGrade >= 9);
  cout << "A grade";
```

- Lo que parece: si grade >= 9 $\Longrightarrow$ *A grade*.
  - Pero en realidad siempre veremos por pantalla *A grade*.
- Al poner ; al final del `if` ¡la hemos convertido en una instrucción!
  - Se ejecuta, se evalúa la condición, y se acaba su *influencia*.
  - A continuación se ejecuta la siguiente instrucción ¡siempre!
- Esto es un error de lógica, un fallo en nuestra implementación.
  - No generar error de compilación, salvo que después apareciese un `else`.

```
if (finalGrade = 10)
  cout << "A grade with honors";
```

- Otro error lógico: confundir comparación == con asignación =.
- Lo que parece: si grade == 10 $\Longrightarrow$ *Matrícula de honor*.
  - Pero en realidad ¡ponemos matrículas a todos los alumnos!

El problema del `else` suelto (1/2)

```
if (studentGrade >= 9)
  if (practicalExercises >= 9)
    cout << "A grade with honors";
else
    cout << "studentGrade < 9";
```

- Lo que parece: si grade $>= 9$ y exerc $>= 9$ $\Longrightarrow$ *A with honors*.
  - Si no (grade < 9) $\Longrightarrow$ *studentGrade* < 9.

El problema del `else` suelto (2/2)

- ¡Pero tal anidamiento no se ejecuta como parece!
- Es un problema de indentación por nuestra parte.
  - Recuerda siempre: el `else` siempre va anidado al último `if`.
- A continuación vemos cómo deberíamos haberlo indentado...
  - ...que es como lo interpreta el compilador.

```
if (studentGrade >= 9)
  if (practicalExercises >= 9)
    cout << "A grade with honors";
  else
    cout << "studentGrade < 9";
```

- Y vemos que lo implementado no tiene sentido $\Longrightarrow$ error lógico.
  - Si el `if` externo es `true` y el `if` interno es `false`.
    - Tendremos como salida que $studentGrade < 9$.
  - Y partíamos de que el `if` externo era `true` ($studentGrade >= 9$).
- Estos son los errores más difíciles de detectar, ¡los de lógica!

- Para realizar distintas acciones según valor el valor de una variable.
  - Esta variable o expresión puede ser tipo carácter o entera.
- Olvidar `break` cuando es necesario es un error lógico.

```
switch (grade) { // goal: get number of A's, B's, C's...
case 'A': // A grade (upper case)
case 'a': // or a (lower case)
  aCounter++; // increment number of As
  break; // break the switch
case 'B': // B grade (upper case)
case 'b': // or b (lower case)
  bCounter++; // increment number of Bs
  break; // break the switch
case 'C': // C grade (upper case)
case 'c': // or c (lower case)
  cCounter++; // increment number of Cs
  break; // break the switch

// [...] It is continued next slide
```

```
case 'D': // D grade (upper case)
case 'd': // or d (lower case)
  dCounter++; // increment number of Ds
  break; // break the switch
case 'F': // F grade (upper case)
case 'f': // or f (lower case)
  fCounter++; // increment number of Fs
  break; // break the switch

case '\n': // ignores \n, \t and blank spaces
case '\t':
case ' ':
  break; // break the switch

default: // catch the rest of characters
  cout << "Grade is wrong!" << endl;
  break; // optional (since it is the last sentence)
} // end of switch
```

- Sirven para repetir una acción o conjunto de estas bajo una condición.
- El resultado de los siguientes tres bucles es exactamente el mismo.

```
int i = 1;
while (i <= 100) {
  cout << "Step n. " << i << endl;
  i += 1;
}
// ------------------------------------
for (i = 1; i <= 100; i += 1)
  cout << "Step n. " << i << endl;
// ------------------------------------
i = 1;
do {
  cout << "Step n. " << i << endl;
  i += 1;
} while (i < = 100);
```

- `while`: es el más polivalente, pero tiene problema de bucle infinito.
  - Error de lógica frecuente: olvido de acción que torne la condición falsa.
- `for`: si el objetivo es recorrer alguna colección (sin otra condición).
  - E.g. recorrer un array, una imagen, etc.
  - Por sintaxis, impide olvidar inicializar contador o acción de condición.
- `do...while`: si se necesita entrar la primera vez sin condición.
  - E.g. pedir valores al usuario hasta que introduzca un valor adecuado.
    - Como mínimo voy a necesitar pedírselo la primera vez, y quizás más.
- Usa adecuadamente todos los tipos; ¡no siempre el bucle `while`!
- Evita estos errores lógicos; recuerda que son muy difíciles de detectar.
  - No olvides revisar si la condición de terminación va a ocurrir.
  - No olvides inicialiar los contadores (normalmente a 0 o a 1).
  - Error dpzmto. en 1: revisa inicializar 0 o 1 y condición con $<$ o $<=$.

- Alteran flujo de control (en instr. de selección y repetición).
- Ya hemos visto el uso de `break` dentro del `switch`.
  - Al igual que en este, dentro de bucle ocasiona salida inmediata.
- Un uso adecuado de `break` y `continue` aumenta eficiencia.
  - Pero, ¿se siguen los ppios. de la programación estructurada?

```
#include <iostream>

int main() {
  int counter;
  for (counter = 1; counter <= 10; counter++) { // 10 steps?
    if (counter == 5) break; // if counter == 5 => out!
    std::cout << counter << " ";
  } // end for
  std::cout << "\nEnd for with counter = " << counter;
  return 0;
}
```

- Al usar `continue` en bucle, salta a la siguiente iteración.
  - Omitiendo las instrucciones restantes de la iteración actual.
- En `for`, se ejecuta incremento y después se evalúa `continue`.

```
#include <iostream>

int main() {
  int counter;
  for (counter = 1; counter <= 10; counter++) { // 10 steps?
    if (counter == 5) continue; // counter == 5 => jump next it.!
    std::cout << counter << " "; // ignored when continue
  } // end for
  std::cout << "\nContinue avoided showing number 5\n";
  return 0;
} // It shows 1 2 3 4 6 7 8 9 10
```

- Son aquellos tipos de datos definidos por el propio usuario.
- Facilitan la legibilidad y el mantenimiento del software.
- Se le pueden asignar diferentes valores limitados.
  - Estos valores se deben definir al declarar el tipo.
- Para declarar un tipo enumerado se usa la *keyword* `enum`.
  - Y luego, entre llaves, se definen los valores posibles.
  - E.g. `enum gender male, female;`
- Una vez definido, se pueden crear variables de tal tipo enum.
  - Como si de cualquier otro tipo básico se tratara.
  - E.g. `gender myGender;`
- Los valores se almacenan en un array, según orden declarado.
  - Así que, como cualquier array, estos comienzan por el índice 0.
  - E.g. para acceder al valor `female` sería `myGender(1)`.

```
enum gender {male, female};
enum weekendday {Saturday, Sunday};
enum errorFlag {ioError, success, working};

int main() {
  gender field;
  weekendday day = Saturday;
  errorFlag flag;
  // weekendday anotherDay = Wednesday; // syntax error
  field = male; // it must be used without quotation marks

  // << are overloaded to show the enum value instead of index
  cout << "The gender is " << field << endl; // it shows male
  cout << "The day is " << day << endl; // it shows Saturday

  if (day == Saturday) flag = success;
  else flag = errorFlag(2); // index 2 of errorFlag is working

  cout << "And the flag shows " << flag << endl; // shows success
}
```

1. Introducción a C++

Julio Vega

julio.vega@urjc.es