

# Lenguaje de Programación: Compiladores de C y C++

José Luis Alonzo Velázquez

Universidad de Guanajuato

Septiembre 2010

## C

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell de AT&T como evolución del anterior lenguaje B, a su vez basado en BCPL.

## C

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell de AT&T como evolución del anterior lenguaje B, a su vez basado en BCPL.

## C++

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

# Compiladores

¿Cómo agregar al path el compilador, para poder compilar desde consola?

- 1 Primero iremos a propiedades del sistema.

# Compiladores

¿Cómo agregar al path el compilador, para poder compilar desde consola?

- 1 Primero iremos a propiedades del sistema.
- 2 Ir a variables de entorno.

# Compiladores

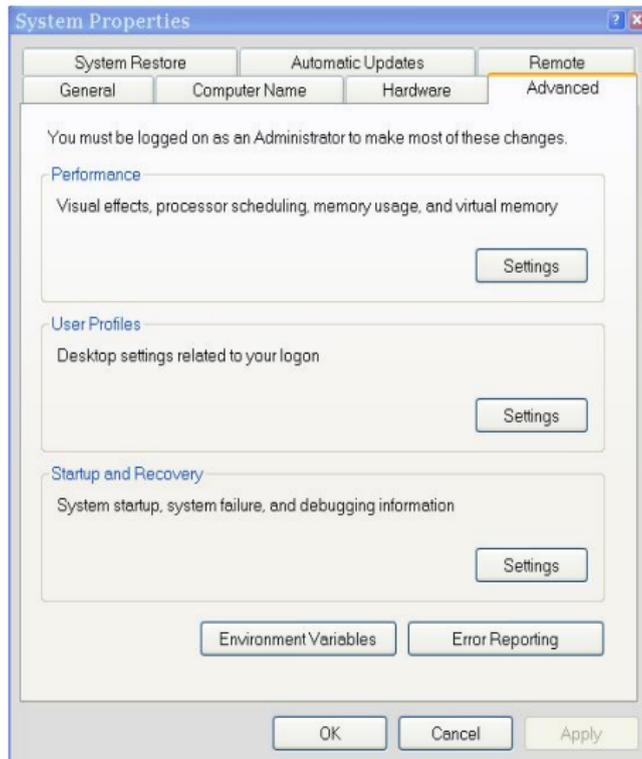
¿Cómo agregar al path el compilador, para poder compilar desde consola?

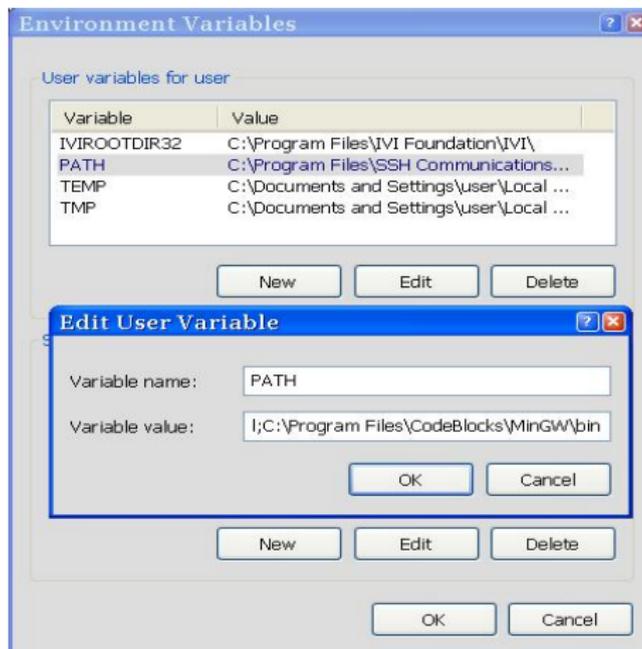
- 1 Primero iremos a propiedades del sistema.
- 2 Ir a variables de entorno.
- 3 Modificar PATH de usuario.

# Compiladores

¿Cómo agregar al path el compilador, para poder compilar desde consola?

- 1 Primero iremos a propiedades del sistema.
- 2 Ir a variables de entorno.
- 3 Modificar PATH de usuario.
- 4 Agregar al final `";C:\ProgramFiles\CodeBlocks\MinGW\bin"`





## ¿Diferencias entre g++ y gcc?

La diferencia entre gcc y g++ es muy sutil. La respuesta queda en el resultado. El g++ puede compilar código 100% C, pero el gcc no puede compilar código de C++ porque no sabe como. Cuando compilas código C en g++ el resultado es código C++ aunque no venga incluida alguna función de C++ por el simple hecho de ser compilado mediante el g++. El código compilado en 100% C es mas pequeño que el mismo código compilado en C++.

Usando de ejemplo este código en C:

```
#include <stdio.h>

int main(){
    printf("Hola Mundo!!!!");
    return 0;
}
```

Compilado con gcc (gcc hola.c -o hola.exe), produce un binario de 9152 bytes.

El mismo código compilado con g++ (g++ hola.c -o hola2.exe), produce un binario de 9277 bytes.

## Cambiando el código a C++:

```
#include <iostream>

using namespace std;

int main(){
    cout << "Hola Mundo!!!!!" << endl;
    return 0;
}
```

## Cambiando el código a C++:

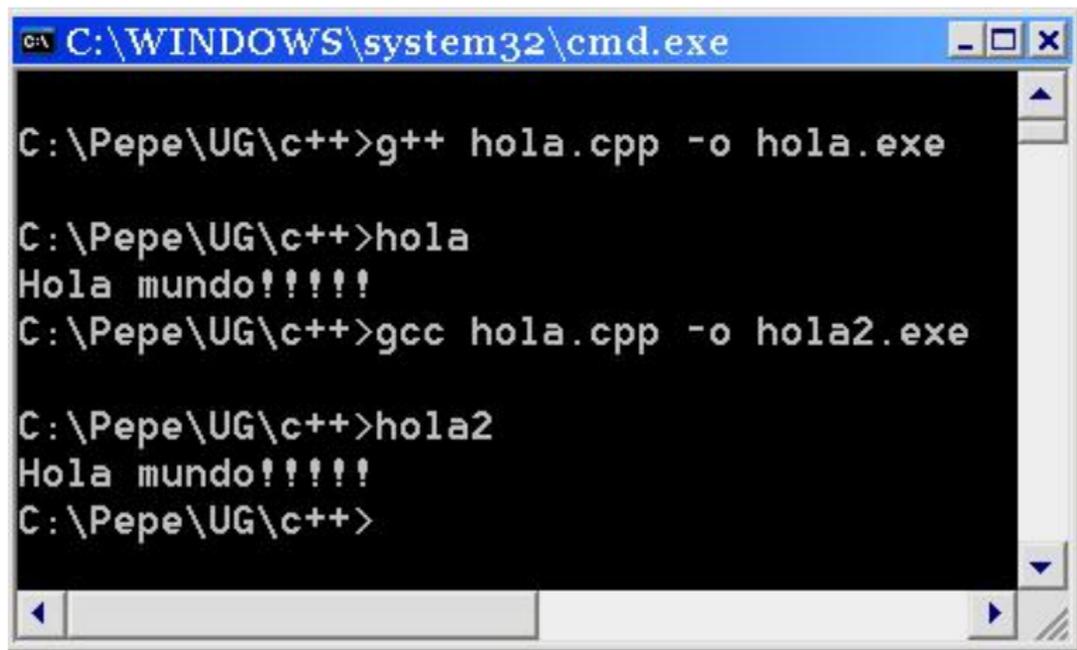
```
#include <iostream>

using namespace std;

int main(){
    cout << "Hola Mundo!!!!!" << endl;
    return 0;
}
```

Compilando con el comando `g++ (g++ hola.cpp -o hola3.exe)` produce un binario de 9793 bytes, y hace exactamente lo mismo que la version escrita en C.

La razón es que al compilar el código en C++ el compilador inserta más funciones y hace más decisiones durante la compilación que con el C puro.



```
C:\WINDOWS\system32\cmd.exe

C:\Pepe\UG\c++>g++ hola.cpp -o hola.exe

C:\Pepe\UG\c++>hola
Hola mundo!!!!

C:\Pepe\UG\c++>gcc hola.cpp -o hola2.exe

C:\Pepe\UG\c++>hola2
Hola mundo!!!!

C:\Pepe\UG\c++>
```

Figura: Compilando y ejecutando con ambos compiladores.

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr. La sigla GCC significa "GNU Compiler Collection". Originalmente significaba "GNU C Compiler"; todavía se usa GCC para designar una compilación en C. G++ refiere a una compilación en C++.

Sintaxis.

```
gcc [ opción | archivo ] ...
```

```
g++ [ opción | archivo ] ...
```

Las opciones van precedidas de un guión, como es habitual en UNIX, pero las opciones en sí pueden tener varias letras; no pueden agruparse varias opciones tras un mismo guión. Algunas opciones requieren después un nombre de archivo o directorio, otras no. Finalmente, pueden darse varios nombres de archivo a incluir en el proceso de compilación.

## Ejemplos

### **gcc hola.c**

compila el programa en C hola.c, genera un archivo ejecutable a.out.

**gcc -o hola hola.c** compila el programa en C hola.c, genera un archivo ejecutable hola.

**g++ -o hola hola.cpp** compila el programa en C++ hola.c, genera un archivo ejecutable hola.

**gcc -c hola.c** no genera el ejecutable, sino el código objeto, en el archivo hola.o. Si no se indica un nombre para el archivo objeto, usa el nombre del archivo en C y le cambia la extensión por .o.

## Ejemplos

**gcc -c -o objeto.o hola.c**

genera el código objeto indicando el nombre de archivo.

**g++ -c hola.cpp**

igual para un programa en C++.

**g++ -o /bin/hola hola.cpp**

genera el ejecutable hola en el subdirectorio bin del directorio propio del usuario.

**g++ -L/lib -L/usr/lib hola.cpp**

indica dos directorios donde han de buscarse bibliotecas. La opción -L debe repetirse para cada directorio de búsqueda de bibliotecas.

**g++ -I/usr/include hola.cpp**

indica un directorio para buscar archivos de encabezado (de extensión .h).

## Sufijos en nombres de archivo

Son habituales las siguientes extensiones o sufijos de los nombres de archivo:

**.c** fuente en C

**.C .cc .cpp .c++ .cp .cxx** fuente en C++; se recomienda **.cpp**

**.m** fuente en Objective-C

**.i** C preprocesado

**.ii** C++ preprocesado

**.s** fuente en lenguaje ensamblador

**.o** código objeto

**.h** archivo para preprocesador (encabezados), no suele figurar en la línea de comando de gcc

## Opciones en la compilación

### - **C**

realiza preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.

### - **E**

realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.

### -**o archivo**

indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.

### -**I ruta**

especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, así: `-I/usr/include`

## Opciones en la compilación

**-L**

especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, así: `-L/usr/lib`

**-Wall**

muestra todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.

**-g**

incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (GNU Debugger).

**-v**

muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

## Etapas de compilación

El proceso de compilación involucra cuatro etapas sucesivas: **preprocesamiento**, **compilación**, **ensamblado** y **enlazado**. Para pasar de un programa fuente escrito por un humano a un archivo ejecutable es necesario realizar estas cuatro etapas en forma sucesiva. Los comandos `gcc` y `g++` son capaces de realizar todo el proceso de una sola vez.

## Preprocesado

En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con `#define` son sustituidas en el código por su valor en todos los lugares donde aparece su nombre.

Usaremos como ejemplo este sencillo programa de prueba, `circulo.c`:

```
/* Circulo.c: calcula el área de un círculo.  
   Ejemplo para mostrar etapas de compilación.  
*/  
#define PI 3.1416  
  
main()  
{  
    float area, radio;  
  
    radio = 10;  
    area = PI * (radio * radio);  
    printf("Circulo.\n");  
    printf("%s%f\n\n", "Area de circulo radio 10: ", area);  
}
```

El preprocesado puede pedirse con cualquiera de los siguientes comandos; cpp alude específicamente al preprocesador.

```
$ gcc -E circulo.c > circulo.pp
```

```
$ cpp circulo.c > circulo.pp
```

Examinando circulo.pp

```
$ more circulo.pp
```

puede verse que la variable PI ha sido sustituida por su valor, 3.1416, tal como había sido fijado en la sentencia #define.

## 2. Compilación

La compilación transforma el código C en el lenguaje ensamblador propio del procesador de nuestra máquina.

```
$ gcc -S circulo.c
```

realiza las dos primeras etapas creando el archivo circulo.s;  
examinándolo con

```
$ more circulo.s
```

puede verse el programa en lenguaje ensamblador.

### 3. Ensamblado

El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.

El ensamblador se denomina así: `$ as -o circulo.o circulo.s` crea el archivo en código objeto `circulo.o` a partir del archivo en lenguaje ensamblador `circulo.s`. No es frecuente realizar sólo el ensamblado; lo usual es realizar todas las etapas anteriores hasta obtener el código objeto así:

```
$ gcc -c circulo.c
```

donde se crea el archivo `circulo.o` a partir de `circulo.c`. Puede verificarse el tipo de archivo usando el comando

```
$ file circulo.o
```

```
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

## Nota: sobre proyectos grandes

En los programas extensos, donde se escriben muchos archivos fuente en código C, es muy frecuente usar gcc o g++ con la opción -c para compilar cada archivo fuente por separado, y luego enlazar todos los módulos objeto creados. Estas operaciones se automatizan colocándolas en un archivo llamado makefile, interpretable por el comando make, quien se ocupa de realizar las actualizaciones mínimas necesarias toda vez que se modifica alguna porción de código en cualquiera de los archivos fuente.

## 4. Enlazado

Las funciones de C/C++ incluidas en nuestro código, tal como `printf()` en el ejemplo, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones a nuestro ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.

El enlazador se denomina `ld`. El comando para enlazar

```
$ ld -o circulo circulo.o -lc
```

`ld: warning: cannot find entry symbol _start; defaulting to 08048184` da este error por falta de referencias. Es necesario escribir algo como

```
$ ld -o circulo /usr/lib/gcc - lib/i386 - linux/2,95,2/collect2 -m  
elf_i386 -dynamic-linker /lib/ld - linux.so,2 -o circulo  
/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc - lib/i386 -  
linux/2,95,2/crtbegin.o -L/usr/lib/gcc - lib/i386 - linux/2,95,2
```

## 4. Enlazado

El uso directo del enlazador ld es muy poco frecuente. En su lugar suele proveerse a gcc los códigos objeto directamente:

```
$ gcc -o circulo circulo.o
```

crea el ejecutable circulo, que invocado por su nombre

```
$ ./circulo
```

Circulo. Area de circulo radio 10: 314.160004 da el resultado mostrado. Todo en un solo paso. En programa con un único archivo fuente todo el proceso anterior puede hacerse en un solo paso:

```
$ gcc -o circulo circulo.c
```

No se crea el archivo circulo.o; el código objeto intermedio se crea y destruye sin verlo el operador, pero el programa ejecutable aparece allí y funciona.

Es instructivo usar la opción -v de gcc para obtener un informe detallado de todos los pasos de compilación:

```
$ gcc -v -o circulo circulo.c
```



Programming Principles and Practice Using C++, Bjarne Stroustrup.



<http://www.codeblocks.org>



<http://www.wxwidgets.org>