



Pablo Alcain
pabloalcain@gmail.com

Compilación y Linkeo con
Diferentes Lenguajes

¿Qué hacemos cuando “compilamos”?

Consideremos un mínimo “Hola mundo” en C...

```
#include <stdio.h>
int main(int argc, char **argv)
{
    puts("Hello, world!\n");
    return 0;
}
```

... y “compilémoslo”

```
$ gcc hello_world.c -o hello_world
```

Más datos: “compilamos” con la opción **-v**

¿Qué hacemos cuando “compilamos”?

El ejecutable se crea en una cadena de 4 eslabones:

1. Pre-procesador
2. Compilación I: de C a assembler
3. Compilación II: de assembler a lenguaje de máquina
4. Linkeo: de lenguaje de máquina a ejecutable

¿Qué hacemos cuando “compilamos”?

El ejecutable se crea en una cadena de 4 eslabones:

1. Pre-procesador
2. Compilación I: de C a assembler
3. Compilación II: de assembler a lenguaje de máquina
4. Linkeo: de lenguaje de máquina a ejecutable

gcc es un “wrapper” que se encarga de llamar a estos 4 pasos en el orden adecuado.

Los “flags” que le pasamos a **gcc** a su vez van a ir a cada una de estas etapas, de acuerdo a quién le pertenezca

Pre-procesador

Obligatorio en C/C++

Se encarga de las directivas **#** como los includes y las macros.
En este caso, incluye `<stdio.h>` y, a su vez, todos los que éste incluya

```
#include <stdio.h>
int main(int argc, char **argv)
{
    puts("Hello, world!\n");
    return 0;
}
```

Para detener a **gcc** en esta etapa usamos el flag **-E**

Pre-procesador

```
$ gcc -E hello_world.c -o hello_world.pp.c
```

```
# 1 "hello_world.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello_world.c"
# 1 "/usr/include/stdio.h" 1 3 4
.....
extern int puts (const char *__s);
.....
# 942 "/usr/include/stdio.h" 3 4

# 2 "hello_world.c" 2
int main(int argc, char **argv) {
    puts("Hello, world!\n");
    return 0;
}
```

Evitamos tener que definir funciones y macros “usuales”

Compilación I: de C a assembler

Aquí está **el compilador**, el trabajo más pesado de los desarrolladores. Cuando hablan de diferencias entre compilar con intel o con gnu, están en esta etapa. Lo podemos separar en 3 subsecciones:

- a. El **front end**: Análisis sintáctico, semántico y léxico. Genera una *intermediate representation* (IR)
- b. El **middle end**: Toma la IR del front end y la optimiza -vamos a ver en otras clases cómo-.
- c. El **back end**: Genera el código de assembler y realiza optimizaciones de hardware (por ejemplo, elige qué registros se utilizan para las variables)

Para detener a **gcc** en esta etapa usamos el flag **-S**

Compilación I: de C a assembler

```
$ gcc -S hello_world.c -o hello_world.asm
```

```
.file "hello_world.c"
.section .rodata
.LC0:
.string "Hello, world!\n"
.text
.globl main
.type main, @function
main:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov DWORD PTR [rbp-4], edi
    mov QWORD PTR [rbp-16], rsi
    mov edi, OFFSET FLAT:.LC0
    call puts
    mov eax, 0
    leave
    ret
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 4.9.2-10ubuntu13) 4.9.2"
.section .note.GNU-stack,"",@progbits
```

Todavía podemos leerlo, cada vez más y más difícil

Compilación II: de assembler a máquina

Assembler está muy cerca del código de máquina. El paso es a través de un compilador, que genera los objetos en binario.

Para detener a **gcc** en esta etapa usamos el flag **-c**

Compilación II: de assembler a máquina

Assembler está muy cerca del código de máquina. El paso es a través de un compilador, que genera los objetos en binario.

Para detener a **gcc** en esta etapa usamos el flag **-c**

```
$ gcc -c hello_world.c -o hello_world.o
```

Ahora no podemos ver el archivo, pero sí podemos averiguar algunas cosas

```
$ nm hello_world.o
00000000000000000000 T main          descriptor
U puts             entrada
```

El objeto es binario, pero todavía no puede ejecutarse; para eso viene el linker

```
$ gcc -v hello_world.o -o hello_world.e verbose  
.....  
/usr/lib/gcc/x86_64-linux-gnu/4.9/collect2 -plugin /usr/lib/gcc/x86_64-linux-  
gnu/4.9/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/4.9/lto-wrapper  
-plugin-opt=-fresolution=/tmp/ccEeIqCi.res -plugin-opt=-pass-through=-lgcc -plugin-  
opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-  
lgcc -plugin-opt=-pass-through=-lgcc_s --sysroot=/ --build-id --eh-frame-hdr -m  
elf_x86_64 --hash-style=gnu --dynamic-linker /lib64/ld-linux-x86-64.so.2 -o  
hello_world.e /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crt1.o  
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crti.o  
/usr/lib/gcc/x86_64-linux-gnu/4.9/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.9  
-L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-  
linux-gnu/4.9/../../../../..../lib -L/lib/x86_64-linux-gnu -L/lib/./lib -L/usr/lib/x86_64-  
linux-gnu -L/usr/lib/./lib -L/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../.. hello_world.o  
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-  
needed /usr/lib/gcc/x86_64-linux-gnu/4.9/crtend.o /usr/lib/gcc/x86_64-linux-  
gnu/4.9/../../../../x86_64-linux-gnu/crtn.o
```

El objeto es binario, pero todavía no puede ejecutarse; para eso viene el linker

```
$ gcc -v hello_world.o -o hello_world.e verbose  
.....  
collect2 -dynamic-linker /lib64/ld-linux-x86-  
64.so.2 -o hello_world.e crt1.o crti.o crtbegin.o  
hello_world.o -lgcc -lc crtend.o crtn.o  
lo que pusimos... y más
```

El objeto es binario, pero todavía no puede ejecutarse; para eso viene el linker

```
$ gcc -v hello_world.o -o hello_world.e verbose  
.....  
collect2 -dynamic-linker /lib64/ld-linux-x86-  
64.so.2 -o hello_world.e crt1.o crti.o crtbegin.o  
hello_world.o -lgcc -lc crtend.o crtn.o  
lo que pusimos... y más
```

¡Tenemos un ejecutable!

```
$ ./hello_world.e  
Hello, world!
```

Descriptores del objeto: ¿Dicen algo?

```
/* file: visibilty.c */
#include <stdio.h>

static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;

static int add_abs(const int v1, const int v2) {
    return abs(v1)+abs(v2);
}

int main(int argc, char **argv) {
    int val5 = 20;
    printf("%d / %d / %d\n", add_abs(val1,val2),
          add_abs(val3,val4), add_abs(val1,val5));
    return 0;
}
```

```
$ nm visibility.o
0000 t add_abs
002a T main
      U printf
0000 r val1
0004 R val2
0000 d val3
0004 D val4
```

descriptores

¿Esto qué quiere decir?

Un programa con dos archivos chiquitos:

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ nm exp.o  
0000 T exp
```

```
$ gcc run.o exp.o -o run.e
```

El linker resuelve esas dependencias

¿Esto qué quiere decir?

Un programa con dos archivos chiquitos:

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ nm exp.o  
0000 T exp
```

```
$ gcc run.o exp.o -o run.e
```

El linker resuelve esas dependencias

Si modifico **exp.c** los objetos que dependían de exp pueden permanecer, y sólo compilo **exp.o**

¿Esto qué quiere decir?

Un programa con dos archivos chiquitos:

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ nm exp.o  
0000 T exp
```

```
$ gcc run.o exp.o -o run.e
```

El linker resuelve esas dependencias

Si modifico **exp.c** los objetos que dependían de exp pueden permanecer, y sólo compilo **exp.o**

Sólo compilo lo que modifico

Librerías estáticas: muchos objetos juntos

¿Tenemos que hacer nuestro propio objeto? No siempre...

Supongamos que tenemos la librería **mymath**.

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ gcc run.o -o run.e
```

Librerías estáticas: muchos objetos juntos

¿Tenemos que hacer nuestro propio objeto? No siempre...

Supongamos que tenemos la librería **mymath**.

```
$ nm run.o
0000 T main
      U exp
```

```
$ gcc run.o -o run.e
run.o: In function `main':
run.c:(.text+0x18): undefined reference to `exp'
collect2: error: ld returned 1 exit status
```

Librerías estáticas: muchos objetos juntos

¿Tenemos que hacer nuestro propio objeto? No siempre...

Supongamos que tenemos la librería `mymath`.

```
$ nm run.o
0000 T main
      U exp
```

```
$ gcc run.o -o run.e
run.o: In function `main':
run.c:(.text+0x18): undefined reference to `exp'
collect2: error: ld returned 1 exit status
```

Librerías estáticas: muchos objetos juntos

¿Tenemos que hacer nuestro propio objeto? No siempre...

Supongamos que tenemos la librería **mymath**.

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ gcc -lmymath run.o -o run_static.e
```

Librerías estáticas: muchos objetos juntos

¿Tenemos que hacer nuestro propio objeto? No siempre...

Supongamos que tenemos la librería **mymath**.

```
$ nm run.o  
0000 T main  
      U exp
```

```
$ gcc -lmymath run.o -o run_static.e  
$ nm run_static.e  
4004e8 T exp  
400594 T _fini
```

Efectivamente se “pega” el código del objeto en el ejecutable, el linker se encarga de resolver los saltos en las posiciones de memoria de las demás instrucciones.

Librerías dinámicas: mucho más que objetos

Es incómodo (¡y poco eficiente!) tener que pegar el código de todos los símbolos. **gcc** por defecto linkea *dinámicamente*

```
$ gcc -lm run.o -o run_shared.e
```

Una librería dinámica es su **soname**

El linker

Librerías dinámicas: mucho más que objetos

Es incómodo (¡y poco eficiente!) tener que pegar el código de todos los símbolos. **gcc** por defecto linkea *dinámicamente*

```
$ gcc -lm run.o -o run_dyn.e
$ nm run_dyn.e
         U exp@@GLIBC_2.2.5          soname
400674 T _fini
```

Efectivamente no está definido. Es porque se va a buscar y definir en tiempo de ejecución.

(En realidad se ejecuta primero el linker, y éste llama al ejecutable.)

El linker sabe a quién llamar gracias al **soname**.

Algunas diferencias

Las librerías estáticas se resuelven enteras, de izquierda a derecha.
Dependencias circulares: varias veces la misma librería o agruparlas:

-Wl, --start-group (. . .) -Wl, --end-group

(opción para el linker)

NUNCA linkear estáticamente a GNU libc

Linkeo dinámico requiere compatibilidad entre ejecución y compilación (por ejemplo, si lo van a llevar a otra computadora)

Se puede hacer un linkeo híbrido, algunas librerías estáticamente:

-Wl, -Bstatic, (. . .) , -Wl, -Bdynamic

Construyendo librerías

Librerías estáticas: **ar**

```
$ ar cr libmymath.a exp.o trig.o
```

Construyendo librerías

Librerías estáticas: **ar**

```
$ ar cr libmymath.a exp.o trig.o
$ nm -n libmymath.a ordena por memoria
exp.o:
0000 T exp
trig.o:
0000 T sin
0016 T cos
0046 T tan
```

Construyendo librerías

Librerías estáticas: **ar**

```
$ ar cr libmymath.a exp.o trig.o
$ nm -n libmymath.a          ordena por memoria
exp.o:
0000 T exp
trig.o:
0000 T sin
0016 T cos
0046 T tan
```

Agrupar varias librerías

```
$ cat libmymodel.a
GROUP (-lmymath -lmychem -lmyphys)
```

Construyendo librerías

Librerías dinámicas: **gcc** (es un poco más complicado)

```
$ gcc -fPIC -c exp.c  
$ gcc -fPIC -c trig.c
```

Position Independent Code

Construyendo librerías

Librerías dinámicas: **gcc** (es un poco más complicado)

```
$ gcc -fPIC -c exp.c
```

```
$ gcc -fPIC -c trig.c
```

```
$ gcc -shared exp.o trig.o -o libmymath.so
```

Position Independent Code

Construyendo librerías

Librerías dinámicas: **gcc** (es un poco más complicado)

```
$ gcc -fPIC -c exp.c
```

```
$ gcc -fPIC -c trig.c
```

Position Independent Code

```
$ gcc -shared exp.o trig.o -o libmymath.so
```

```
$ nm -n libmymath.so
```

```
...
```

```
06c0 T exp
```

```
06f6 T sin
```

```
070c T cos
```

```
073c T tan
```

Construyendo librerías

Librerías dinámicas: **gcc** (es un poco más complicado)

```
$ gcc -fPIC -c exp.c
$ gcc -fPIC -c trig.c
$ gcc -shared exp.o trig.o -o libmymath.so
$ nm -n libmymath.so

. . .
06c0 T exp
06f6 T sin
070c T cos
073c T tan
```

Position Independent Code

La variable de entorno **\$LD_PRELOAD** antepone alguna librería

```
$ LD_PRELOAD=./fastmath.so ./myprog
```


¿Qué pasa en FORTRAN?

El preprocessing no es obligatorio, y se habilita implícitamente con la extensión: file.F, file.FOR, file.F90

```
! file: hello_world.f90  
subroutine greet  
  print*, 'Hello, world!'  
end subroutine greet  
  
program hello  
  call greet  
end program
```

```
$ nm hello_world.o  
U _gfortran_set_args  
U _gfortran_set_options  
U _gfortran_st_write  
U _gfortran_st_write_done  
U _gfortran_transfer_...  
0000 T greet_  
0078 T main  
006d t MAIN_  
0020 r options.1.2323
```

fortran runtime library

decoradores

program

Comunicando FORTRAN y C

source

run.c

sum.f90

exp.c

trig.c

Comunicando FORTRAN y C

source

run.c

sum.f90

exp.c

trig.c

preproc

run.pp.c

exp.pp.c

trig.pp.c



Comunicando FORTRAN y C

source

run.c

sum.f90

exp.c

trig.c

preproc

run.pp.c

exp.pp.c

trig.pp.c

assembler

run.asm

sum.asm

exp.asm

trig.asm

Comunicando FORTRAN y C

source

run.c

sum.f90

exp.c

trig.c

preproc

run.pp.c

exp.pp.c

trig.pp.c

assembler

run.asm

sum.asm

exp.asm

trig.asm

objeto

run.o

sum.o

exp.o

trig.o

Comunicando FORTRAN y C

source

run.c

sum.f90

exp.c

trig.c

preproc

run.pp.c

exp.pp.c

trig.pp.c

assembler

run.asm

sum.asm

exp.asm

trig.asm

objeto

run.o

sum.o

exp.o

trig.o

ejecutable

calc.e

Comunicando FORTRAN y C

En general, hay que mirar dos cosas:

1. **Decoradores**: No están definidos por el estándar, pero hay convenciones usuales [el `_` en fortran (GNU); más difícil en C++]
2. **Memory Management**: Por defecto, en FORTRAN todos son punteros. Es mejor usar siempre arrays 1D en C.

Si recuerdan el memory management y observan los símbolos en cada objeto, es (dentro de todo) fácil

Comunicando FORTRAN y C

En general, hay que mirar dos cosas:

1. **Decoradores**: No están definidos por el estándar, pero hay convenciones usuales [el `_` en fortran (GNU); más difícil en C++]
2. **Memory Management**: Por defecto, en FORTRAN todos son punteros. Es mejor usar siempre arrays 1D en C.

Si recuerdan el memory management y observan los símbolos en cada objeto, es (dentro de todo) fácil

(ojo con los **strings**, se pasan como arrays en FORTRAN)

Comunicando FORTRAN y C: Detalle

```
/* file: run.c */  
  
extern void dot_(float *x, float *y,  
                int *n, float *res);  
  
int main (int argc, char **argv) {  
    float x[3], y[3];  
    float z;  
    int n;  
    n = 3;  
    x[0] = 1.0; x[1] = 2.0; x[2] = 3.0;  
    y[0] = 0.0; y[1] = 5.0; y[2] = 9.0;  
    dot_(x, y, &n, &z);  
    return 0;  
}
```

```
$ nm run.o  
          U dot_  
0000 T main
```

```
! file: dot.f90  
subroutine dot(x, y, n, res)  
    real      :: x(n), y(n)  
    integer   :: n  
    real      :: res  
    integer   :: i  
    real      :: prod  
  
    prod = 0  
    do i = 1, n  
        prod = prod + x(i) * y(i)  
    end do  
    res = prod  
end subroutine dot
```

```
$ nm dot.o  
0000 T dot_
```

Comunicando FORTRAN y C: Detalle

```
/* file: run.c */  
  
extern void dot_(float *x, float *y,  
                int *n, float *res);  
  
int main (int argc, char **argv) {  
    float x[3], y[3];  
    float z;  
    int n;  
    n = 3;  
    x[0] = 1.0; x[1] = 2.0; x[2] = 3.0;  
    y[0] = 0.0; y[1] = 5.0; y[2] = 9.0;  
    dot_(x, y, &n, &z);  
    return 0;  
}
```

```
$ nm run.o  
U dot_  
0000 T main
```

```
! file: dot.f90  
subroutine dot(x, y, n, res)  
    real      :: x(n), y(n)  
    integer   :: n  
    real      :: res  
    integer   :: i  
    real      :: prod  
  
    prod = 0  
    do i = 1, n  
        prod = prod + x(i) * y(i)  
    end do  
    res = prod  
end subroutine dot
```

```
$ nm dot.o  
00000 T dot_
```

Comunicando FORTRAN y C: Detalle

```
/* file: run.c */  
  
extern void dot_(float *x, float *y,  
                decoradores int *n, float *res);  
                                memoria  
  
int main (int argc, char **argv) {  
    float x[3], y[3];  
    float z;  
    int n;  
    n = 3;  
    x[0] = 1.0; x[1] = 2.0; x[2] = 3.0;  
    y[0] = 0.0; y[1] = 5.0; y[2] = 9.0;  
    dot_(x, y, &n, &z);  
    return 0;  
}
```

```
$ nm run.o  
U dot_  
0000 T main
```

```
! file: dot.f90  
subroutine dot(x, y, n, res)  
    real      :: x(n), y(n)  
    integer   :: n  
    real      :: res  
    integer   :: i  
    real      :: prod  
  
    prod = 0  
    do i = 1, n  
        prod = prod + x(i) * y(i)  
    end do  
    res = prod  
end subroutine dot
```

```
$ nm dot.o  
00000 T dot_
```

Algunas conclusiones

En C y en FORTRAN los símbolos sólo tienen el nombre de la función.
No podemos sobrecargar funciones:

```
/* file: divby2.c */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.c
```

Algunas conclusiones

En C y en FORTRAN los símbolos sólo tienen el nombre de la función.
No podemos sobrecargar funciones:

```
/* file: divby2.c */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.c  
divby2.c:5:7: error: conflicting types for 'divby2'  
    float divby2(float x) {  
        ^  
divby2.c:1:5: note: previous definition of 'divby2' was here  
    int divby2(int a) {  
        ^
```

Algunas conclusiones

Son importantes los headers para compilar con la misma función que vamos a linkear

```
/* file: run.c */
#include "exp.h"
int main(int argc, char **argv){
    float y;
    y = myexp(2.0);
    return 0;
}
```

```
/* file: exp.c */
#include "exp.h"

float myexp(float x) {
    return 1 + x + x*x/2;
}
```

```
/* file: exp.h */
#ifndef EXP_H
#define EXP_H
float myexp(float x);
#endif
```

Algunas conclusiones

Son importantes los headers para compilar con la misma función que vamos a linkear

```
/* file: run.c */
#include "exp.h"
int main(int argc, char **argv){
    float y;
    y = myexp(2.0);
    return 0;
}
```

```
/* file: exp.c */
#include "exp.h"

float myexp(float x) {
    return 1 + x + x*x/2;
}
```

```
/* file: exp.h */
#ifndef EXP_H
#define EXP_H
float myexp(float x);
#endif
```

include guards

C++ a C

La clave es ver los objetos

```
/* file: divby2.cpp */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.cpp
```


C++ a C

La clave es ver los objetos

```
/* file: divby2.cpp */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.cpp  
$ nm divby2.o  
0000 T _Z6divby2i  
0015 T _Z6divby2f
```

C++ a C

La clave es ver los objetos

```
/* file: divby2.cpp */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.cpp  
$ nm divby2.o  
0000 T Z6divby2i  
0015 T Z6divby2f
```

identificador
longitud
nombre
argumentos

C++ a C

La clave es ver los objetos

```
/* file: divby2.cpp */  
int divby2(int a) {  
    return a/2;  
}  
  
float divby2(float x) {  
    return x/2;  
}
```

```
$ gcc -c divby2.cpp  
$ nm divby2.o  
0000 T Z6divby2i  
0015 T Z6divby2f
```

identificador
longitud
nombre
argumentos

El name mangling no está estandarizado

C++ a C

La clave es ver los objetos

```
/* file: divby2_c.cpp
*/
extern "C" {
    int divby2(int a);
}

int divby2(int a) {
    return a/2;
}
```

```
$ gcc -c divby2_c.cpp
```

C++ a C

La clave es ver los objetos

```
/* file: divby2_c.cpp
*/
extern "C" {
    int divby2(int a);
}

int divby2(int a) {
    return a/2;
}
```

```
$ gcc -c divby2_c.cpp
$ nm divby2_c.o
0000 T divby2
```

C++ a C

Y los namespaces...

```
/* file: calculator.cpp */  
namespace Calculator{  
    int divby2(int a);  
};  
  
int Calculator::divby2(int a) {  
    return a/2;  
}
```

```
$ gcc -c calculator.cpp
```

C++ a C

Y los namespaces...

```
/* file: calculator.cpp */  
namespace Calculator{  
    int divby2(int a);  
};  
  
int Calculator::divby2(int a) {  
    return a/2;  
}
```

```
$ gcc -c calculator.cpp  
$ nm calculator.o  
0000 T _ZN10Calculator6divby2Ei
```

C++ a C

extern al rescate!

```
/* file: calculator_c.cpp */  
namespace Calculator{  
    extern "C" {  
        int divby2(int a);  
    }  
};  
  
int Calculator::divby2(int a) {  
    return a/2;  
}
```

```
$ gcc -c calculator_c.cpp
```


C++ a C

extern al rescate!

```
/* file: calculator_c.cpp */  
namespace Calculator{  
    extern "C" {  
        int divby2(int a);  
    }  
};  
  
int Calculator::divby2(int a) {  
    return a/2;  
}
```

```
$ gcc -c calculator_c.cpp  
$ nm calculator_c.o  
0000 T divby2
```

FORTRAN 2003

Estandarizada (al fin!) la interfaz con C

El estándar está dado por el módulo **ISO_C_BINDING**

Las subrutinas se pueden declarar con **BIND(C)**

Definen tipos **C_INT**, **C_FLOAT**, **C_SIGNED_CHAR**

Los strings siguen siendo complicados



Pablo Alcain
pabloalcain@gmail.com

Compilación y Linkeo con
Diferentes Lenguajes