



<http://sites.google.com/site/utnfrbactalks/>

Oradores

Matías Dumrauf
Facundo Viale

Talk II

Tools and More Stuff

- C para Avanzados
- GCC & Makefile
- Development Tools
- Valgrind

C para Avanzados

¿ Que es un puntero a una Función ?

Un puntero a una función, es muy similar a un puntero tradicional. Básicamente es una variable que referencia una región de memoria; en otras palabras es una variable cuyo valor es una dirección de memoria, pero esta región de memoria es correspondiente a código ejecutable.

El código ejecutable, al igual que las variables se encuentra en la memoria ram por lo cual este se encuentra entre un rango de dirección que solo tiene acceso de lectura y ejecución, a diferencia de las variables normales que tienen acceso de lectura y escritura

```
15 void sayHello(char *name){  
16     printf("Hello, %s\n", name);  
17 }  
18  
19 int main(int argc, char **argv) {  
20     void (*func)(char*);  
21  
22     func = sayHello;  
23  
24     func("John");  
25  
26     return EXIT_SUCCESS;  
27 }  
28
```

¿ Porque usar punteros a funciones ?

- Mayor modularidad
- Mayor delegación de responsabilidades
- Introducir conceptos de lenguajes funciones como por Ej:
Funciones de Orden Superior.
- Al ser una herramienta que se maneja en tiempo de ejecución, nos ofrece mas dinamismo en el código.
- Mayor reutilización de funciones

Veamos un Ejemplo con Listas

Supongamos un TAD `t_list` que sirve para el manejo de listas enlazadas, nosotros queremos recorrer cada elemento de la lista pero abstrayéndonos de este proceso. Es decir, que el mismo TAD nos provea de una función que sepa recorrer la lista.

```
35 void      collection_list_iterator( t_list *list, void (*closure)(void*) );
18 void persona_print(t_persona *persona){
19     printf("%s, %s\n", persona->apellido, persona->nombre);
20 }
23 int main(int argc, char **argv) {
24     t_list *list = collection_list_create();
25
26     {
27         t_persona *p = malloc( sizeof(t_persona) );
28
29         p->nombre = "John";
30         p->apellido = "Doe";
31
32         collection_list_add(list, p);
33     }
34
35     collection_list_iterator(list, persona_print);
36
37     return EXIT_SUCCESS;
38 }
```

En este caso, la función `collection_list_iterator` sabe recorrer todos los elementos de la lista y por cada elemento que encuentre llama a la función `closure` pasándole como argumento el elemento actual.

Veamos otro Ejemplo con Listas

Tomando como ejemplo el TAD anterior, ahora es nuestra necesidad que el TAD no exponga una función capaz de eliminar a todos los elementos de la lista (Junto con esto queremos liberar la memoria de estos elementos). Ya que nuestro `t_list` es un TAD genérico que funciona con cualquier tipo de dato ya que lo único que sabe es que guarda `void*`, no tiene una forma genérica de saber eliminar los datos que contiene. *Pero usando punteros a funciones esto se soluciona!*

```
46 void collection_list_destroy( t_list *list, void (*data_destroyer)(void*) );
47
22 void persona_destroy(t_persona *persona){
23     free(persona->nombre);
24     free(persona->apellido);
25     free(persona);
26 }
--
42 collection_list_clean(list, persona_destroy);
43
```


Que mas?

Implementando un buen wrapper podríamos encapsular el código de un select de esta manera:

```
101 void sockets_select(t_list* servers,  
102                    t_list *clients,  
103                    int usec_timeout,  
Es 104                    t_socket_client *(*onAcceptClosure)(t_socket_server*),  
105                    int (*onRecvClosure)(t_socket_client*) );  
106
```

- Una lista de servidores que esten a la espera de nuevas conexiones
- Una lista de clientes que están a la espera de recibir información
- El timeout
- Un puntero a una función que es llamada cada vez que sockets_select detecta internamente que un nuevo cliente se tiene que conectar. La función recibir el servidor de donde proviene la nueva conexión e implementa la lógica que decide aceptarla no. Si es que si, retorna la nueva conexión la cual sockets_select agrega a la lista de clientes de lo contrario retorna NULL.
- Un puntero a una función que es llamada cada vez que un cliente recibe información, recibiendo como argumento dicho cliente.

En este caso, logramos delegar responsabilidad y generalizar la función select haciendo que el que la use se abstraiga lo mas posible de la implementación de este.

Inner Function o Nested Functions

Dentro de las extensión propias que trae GCC, se nos da la posibilidad de declarar una función dentro de otra función. Esto puede resultar particularmente útil cuando necesitamos declarar funciones auxiliares que no hacen a la lógica de nuestro programa

```
28 int main(int argc, char **argv) {
29     t_list *list = collection_list_create();
30
31     {
32         t_persona *p = malloc( sizeof(t_persona) );
33
34         p->nombre = "John";
35         p->apellido = "Doe";
36
37         collection_list_add(list, p);
38     }
39
40     void print_list(t_persona *p){ printf("%s, %s\n", p->apellido, p->nombre); }
41
42     collection_list_iterator(list, print_list);
43
44     return EXIT_SUCCESS;
45 }
```

Scope en las Nested Functions

Este tipo de función nos permite manejar el scope de manera especial, ya que pueden acceder a las variable declaradas en la función en la cual están contenida sin tener que recibirla por argumento.

```
15 int sumar(int a, int b){  
16  
17     int _sum(int value){  
18         return a + value;  
19     }  
20  
21     return _sum(b);  
22 }  
23  
24 int main(int argc, char **argv) {  
25  
26     printf("%d", sumar(1,1) );  
27  
28     return EXIT_SUCCESS;  
29 }
```

Esto ocurre porque las variable declaradas en sumar son variables globales a todo el bloque y a todo lo contenido en el bloque incluyendo la nested function.

True Lambda Function

Otra de las extensiones propias del GCC, nos permite hacer funciones Lambda. Este tipo de funciones se utilizan de manera auxiliar y no requieren de un nombre para ser identificadas. En ciertas ocasiones resultan útiles cuando se necesita reutilizar un pedazo de código y resulta poco práctico crear una función puntual para ello.

```
t_list *nombres = collection_list_create();  
  
collection_list_add(nombres, "Juan");  
collection_list_add(nombres, "Pedro");  
collection_list_add(nombres, "Pablo");  
  
collection_list_iterator(nombres, ({ void $(char* str){ puts(str);} $; }));
```

La sentencia `({ void $(char* str){ puts(str);} $; })` nos devuelve un puntero a una función que no tiene retorno y que tiene como parámetro un `char*`. Obviamente nosotros podemos optar por indicar un retorno o no poner argumentos. Incluso esta sentencia puede ser asignado a una variable.

La sintaxis requiere que la sentencia se encuentre encerrada entre paréntesis y llaves, y además colocar `$` donde iría el nombre de la función y `$;` como última sentencia dentro del bloque de la función.

Argumentos Variables

Es una herramienta que nos brinda el lenguaje a través del cual podemos especificar que una función recibe N cantidad de argumentos, donde N es ≥ 0 . Es una herramienta presente en la mayoría de los lenguajes. Ej mas conocido:

```
14 extern int printf (const char *format, ...);
```

Para manejar los argumentos variables C nos da la librería `#include <stdarg.h>`, donde `va_list` es un tipo de dato que contiene los argumentos recibidos. La limitación que tiene C es que, una función no puede recibir solo argumentos variables sino que al menos debe tener un argumento fijo.

Las funciones de `#include <stdarg.h>` son:

`void va_start(va_list ap, argN)`: Recibe un `va_list`, el cual es inicializado, y un `argN` que es el argumento anterior a nuestros argumentos variables.

`void va_copy(va_list dest, va_list src)`: Genera una copia de un `va_list`. **(Solo ANSI C99)**

`type va_arg(va_list ap, type)`: Nos devuelve el siguiente argumento en un determinado tipo.

`void va_end(va_list ap)`: Limpia la estructura `va_list`.

Utilidades?

El caso mas común es si quisiéramos implementar nuestro propio TAD de logs, en el cual llamamos a la función de logeo como un printf pero que internamente agregue mas información como el PID o el ThreadID.

```
4 #ifndef LOG_H_
5 #define LOG_H_
6
7     #include <stdarg.h>
8
9     #define MAX_LOGMESSAGE_LENGTH 1024
10
11     typedef enum {
12         LOG_ERROR    = 1,
13         LOG_WARNING  = 2,
14         LOG_INFO     = 3
15     } e_log_level;
16
17     typedef enum {
18         LOG_OUTPUT_NONE           = 1,
19         LOG_OUTPUT_CONSOLE       = 2,
20         LOG_OUTPUT_FILE          = 3,
21         LOG_OUTPUT_CONSOLEANDFILE = 4
22     } e_log_output;
23
24     typedef struct{
25         char *path;
26         FILE* file;
27         int program_pid;
28         char *program_name;
29         e_log_output mode;
30     } t_log;
31
32     t_log* log_create( t_log*, char *logfile_path, char* program_name, e_log_output mode );
33     int log_info( t_log *, char *thread_name, char *format, ... );
34     int log_warning( t_log *, char *thread_name, char *format, ... );
35     int log_error( t_log *, char *thread_name, char *format, ... );
36     void log_destroyer( t_log* );
37
38
39 #endif /*LOG_H_*/
```

Implementación

```
14 static int log_write ( t_log *, char *thread_name , e_log_level level , const char* format, va_list args_list );
15
16 int log_info(t_log *self, char *thread_name, char *format, ... ){
17     va_list args_list;
18     va_start(args_list, format);
19     return log_write(self, thread_name, LOG_INFO, format, args_list);
20 }
21
22 static int log_write( t_log *self, char *thread_name , e_log_level level , const char* format, va_list args_list ){
23     unsigned int thread_id = pthread_self();
24     char *str_type = log_levelAsString(level);
25     char logbuff[MAX_LOGMESSAGE_LENGTH+100];
26     char msgbuff[MAX_LOGMESSAGE_LENGTH];
27
28     if( self->mode == LOG_OUTPUT_NONE ) return 1;
29
30     vsprintf(msgbuff, format, args_list);
31
32     sprintf( logbuff, " %s/%d %s/%u %s: %s\n", self->programName , self->programPID , thread_name , thread_id , str_type , msgbuff );
33
34     if( self->mode == LOG_OUTPUT_FILE || self->mode == LOG_OUTPUT_CONSOLEANDFILE ){
35         fprintf( self->file , "%s", logbuff );
36         fflush( self->file ) ;
37     }
38     if( self->mode == LOG_OUTPUT_CONSOLE || self->mode == LOG_OUTPUT_CONSOLEANDFILE ){
39         printf("%s", logbuff);
40     }
41
42     return 1;
43 }
```

Otro Ejemplo

```
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <stdarg.h>
14 #include <string.h>
15
16 char* concat(int num, ...){
17     int length = 0, index;
18     char * ret = NULL;
19     va_list ap;
20
21     va_start(ap, num);
22     for(index=0; index<num; index++)
23         length += strlen( va_arg(ap, char*) );
24     va_end(ap);
25
26     if( length > 0 ){
27         ret = malloc(length + 1);
28         memset(ret, 0, length);
29         va_start(ap, num);
30         for(index=0; index<num; index++)
31             strcat(ret, va_arg(ap, char*));
32         va_end(ap);
33     }
34
35     return ret;
36 }
37
38 int main(int argc, char **argv) {
39     printf("%s\n", concat(3, "Hola", " ", "Mundo!") );
40     return EXIT_SUCCESS;
41 }
```


Consideraciones

El mínimo tipo de dato que manejan los argumentos variables es el `int`. Por lo que si pasamos un `char` dentro de un argumento variable, cuando lo leamos vamos a tener que hacer *`va_arg(ap, int)`* en vez de *`va_arg(ap, char)`*.

Internamente el `char` que nosotros pasamos es colocado en una variable `int`.

Thread-Local

Thread-local storage (TLS) es un mecanismo mediante el cual podemos declarar una variable cuyo scope sea el del thread donde se encuentra. Esto quiere decir que cada thread tiene una copia propia de esa variable.

Esto ocurre porque cuando un thread es creado este tiene su propio stack, es por ello que el compilador prepara el código para que la variable en vez de ser creada en la memoria global del proceso, sea creada en el stack de los threads.

Esto resulta muy practico en casos donde necesitamos que una variable sea global a un thread y que no existan posibles problemas de concurrencia. Como por ejemplo el caso de la macro *errno*.

Para declarar que una variable es Thread-Local simplemente usamos la directiva `__thread`:

```
__thread int i;
```

Manejo de Errores

El lenguaje de programación C nos brinda algunas herramientas para el manejo de errores entre ellas el header `errno.h` y la macro llamada `errno`.

La macro `errno`, es un l-value de tipo `int` que contiene el código de error perteneciente a la última función que falló. Esta se encuentra definida en `errno.h` como *`extern int errno;`*.

Ej:

Por ejemplo si usamos un `send` o un `recv` y esta falla. La función devuelve `-1` indicando un error. ¿¿Cuál fue el error??

El código de este se encuentra guardado en `errno`.

Algunos Errores

Dentro de `errno.h` se encuentra una lista casi interminables de defines con el nombre del error y su código. Según el sistema operativo y la plataforma estos pueden variar ampliamente haciendo que no sea 100% portable. Algunos ej:

EACCES: Permission denied (POSIX.1)

EADDRINUSE: Address already in use (POSIX.1)

EADDRNOTAVAIL: Address not available (POSIX.1)

ECONNABORTED: Connection aborted (POSIX.1)

ECONNREFUSED: Connection refused (POSIX.1)

ECONNRESET: Connection reset (POSIX.1)

EIO: Input/output error (POSIX.1)

ELIBACC: Cannot access a needed shared library

ENAMETOOLONG: Filename too long (POSIX.1)

Errores En Entorno Multi-Thread

Cita del estándar Posix

In POSIX.1, errno is defined as an external global variable. But this definition is unacceptable in a multithreaded environment, because its use can result in nondeterministic results. The problem is that two or more threads can encounter errors, all causing the same errno to be set. Under these circumstances, a thread might end up checking errno after it has already been updated by another thread.

Es por ello que

errno is defined by the ISO C standard to be a modifiable lvalue of type int, and must not be explicitly declared; errno may be a macro. errno is **thread-local**; setting it in one thread does not affect its value in any other thread.

Esto indica que existe una variable errno por cada thread en ejecución, es por ello que errno es thread-safety

True Way Of Read & Write Streams

La lectura y escritura sobre streams es una de las operaciones mas típicas que los programadores suelen realizar. Para esto Posix, en la librería ***unistd.h*** no ofrece las funciones:

```
ssize_t read (int fd, void *buff, size_t len);
```

```
ssize_t write(int fd, const void *buff, size_t count);
```

Uno de los problemas mas típicos es es la falta de conciencia a la hora de uso de las funciones y es por ello que se omiten chequeos.

size_t y ssize_t son tipos de datos definidos en el estándar de Posix como wrapper del tipo int. Se diferencian en que size_t no tiene signo y ssize_t es sigando.

Leer con read()

Como se hace un read se deben validar los siguientes casos:

- La función read retorna un valor igual a len, es decir, la lectura fue satisfactoria y en buff se encuentra la información seteada.
- La función read retorna un valor menor a len y los bytes leídos son seteados en buff. Esto indica que un error ocurrió en el medio, que se alcanzo el EOF antes de leer la cantidad indicada en len o que una señal interrumpió el read. Llamar nuevamente a read con el valor de len actualizado y con el buff actualizado terminara de leer los bytes restantes o retornara el error correspondiente.
- La función read retorna 0, esto infica EOF.
- La función read retorna -1 y en errno se setea EINTR, esto indica que una signal fue recibida antes de que se leea cualquier byte
- La función read retorna -1 y en errno se setea EAGAIN, esto indica que no hay nada para leer y debería llamarse la funcion mas adelante. Esto solo es valido cuando se usa un read no bloqueante.
- La función read retorna -1 y en errno se setea un valor distinto de EINTR y EAGAIN lo cual indica un error mas serio.

Leer con read()

```
18     ssize_t ret;
19
20     while( len != 0 && (ret = read(fd, buff, len)) != 0 ) {
21         if (ret == -1) {
22             if (errno == EINTR)
23                 continue;
24             perro ("read");
25             break;
26         }
27
28         len -= ret;
29         buff += ret;
30     }
```


Escribir con write()

La función write es muy similar a la de read, por lo que las consideraciones que se deben tomar son las mismas. Pero existe una salvedad y es que en el write no existe el EOF, la función garantiza que se ejecuta completamente o no se ejecuta. *PERO ESTO NO SIGNIFICA QUE LA ESCRITURA SE HAYA REALIZADO YA QUE EL WRITE NO ESCRIBE EN EL DISPOSITIVO SINO EN LOS BUFFERS INTERNOS DEL KERNEL Y LUEGO ESTE OPTA EL MOMENTO OPORTUNO PARA ESCRIBIR EN DISPOSITIVO.*

```
18     ssize_t ret, nr;
19
20     while ( len != 0 && (ret = write (fd, buf, len)) != 0){
21         if (ret == -1) {
22             if (errno == EINTR)
23                 continue;
24             perror("write");
25             break;
26         }
27
28         len -= ret;
29         buf += ret;
30     }
```

GCC & Makefile

GCC

[From Wikipedia]

GNU Compiler Collection es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y lo distribuye la FSF bajo la licencia GPL. Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX, de código abierto o también de propietarios, como Mac OS X. GCC requiere el conjunto de aplicaciones conocido como binutils para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos o crear listas, enlazarlos, o quitarles símbolos innecesarios.

Originalmente GCC significaba GNU C Compiler (compilador GNU para C), porque sólo compilaba el lenguaje C. Posteriormente se extendió para compilar C++, Objective-C, Fortran, Ada y otros.

Casi todo GCC está escrito en C, a fines de mayo del 2010 se anunció que se comenzará a utilizar C++ en el desarrollo de GCC.

En la versión 4.2.3 se incluye soporte para: ARC, ARM, Blackfin, CRIS, CRX, Darwin, DEC Alpha, DEC Alpha/VMS, FRV, GNU/Linux, H8/300, H8/500, HPPA, IA-64 "Itanium", M32C, M32R/D, MIPS, MMIX, MN10300, Morpho MT, Motorola M680x0, Motorola M68hc1x, Motorola 88000, PA-RISC, PDP-11, PowerPC, RS/6000, Score, SPARC, SuperH, System/370, System 390, System V, TMS320C3x/C4x, V850, VAX, x86, x86-64, Xstormy16, Xtensa, zSeries, A29K, Atmel AVR, C4x, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, MN10200, NS32K y ROMP.

Etapas de compilación en GCC:

- Preprocessing
- Compilation Proper
- Assembly
- Linking

Extensiones que maneja GCC:

Extension	Description
.c	C source code which must be preprocessed.
.i	C source code which should not be preprocessed.
.ii	C++ source code which should not be preprocessed.
.cc, .cp, .cxx, .cpp, .CPP, .c++, .C	C++ source code which must be preprocessed.
.h	C, C++, Objective-C or Objective-C++ header file to be turned into a precompiled header (default).
.hh, .H, .hp, .hxx, .hpp, .HPP, .h++, .tcc	C++ header file to be turned into a precompiled header.
.S, .s	Assembly language source code
.o	Compiled object code
.a, .so	Compiled library code

GCC & Makefile

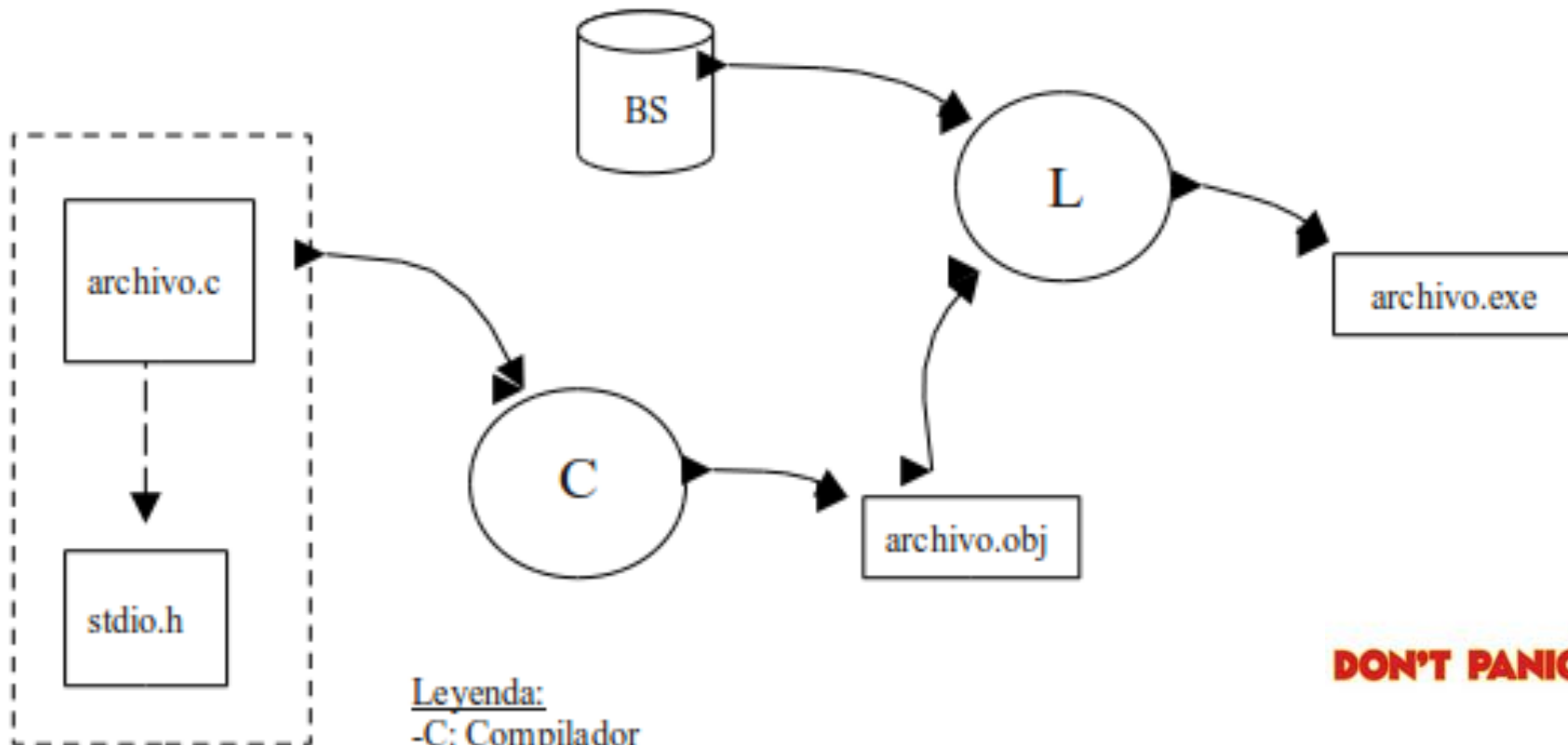
Compilación:

```
facundoviale@ARBADCL051:~/Development/C/Learning$ ls
main.c
facundoviale@ARBADCL051:~/Development/C/Learning$ gcc main.c -o test
facundoviale@ARBADCL051:~/Development/C/Learning$ ls
main.c test
facundoviale@ARBADCL051:~/Development/C/Learning$ █
```

Parámetros de Compilación:

Options	Description
-o FILE	Specify the output filename; not necessary when compiling to object code. If FILE is not specified, the default name is a.out.
-c	Compile without linking.
-DFOO=BAR	Define a preprocessor macro named FOO with a value of BAR on the command-line.
-lFOO	Link against libFOO.
-g	Include standard debugging information in the binary.
-ggdb	Include lots of debugging information in the binary that only the GNU debugger, gdb, can understand.
-ON	Specify an optimization level N, 0<=N<= 3.
-ansi	Support the ANSI/ISO C standard, turning off GNU extensions that conflict with the standard
-pedantic	Emit all warnings required by the ANSI/ISO C standard.
-pedantic-errors	Emit all errors required by the ANSI/ISO C standard.
-w	Suppress all warning messages. Bad Idea!
-Wall	Emit all generally useful warnings that gcc can provide.
-Werror	Convert all warnings into errors, which will stop the compilation
-v	Show the commands used in each step of compilation.

Compilación y Linkeo de un Programa



Leyenda:

- C: Compilador
- L: Linkeo
- BS: Biblioteca Estándar

DON'T PANIC



Make:

[From Wikipedia]

La herramienta make se usa para las labores de creación de fichero ejecutable o programa, para su instalación, la limpieza de los archivos temporales en la creación del fichero, etc ..., todo ello especificando unos parámetros iniciales (que deben estar en el makefile) al ejecutarlo. Es muy usada en los sistemas operativos tipo Unix/Linux. Por defecto lee las instrucciones para generar el programa u otra acción del fichero makefile. Las instrucciones escritas en este fichero se llaman dependencias.

Los Makefiles son los ficheros de texto que utiliza make para llevar la gestión de la compilación de programas. Todos los Makefiles están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto.

Este además de las reglas, nos permite definir variable, colocar comentarios y llamar a comandos. Es decir que a grandes rasgos puede ser visto como un script Shell, BASH o BAT pero mas simplificado y especializado para la tarea de construcción de programas.

Reglas de los Makefile:

```
target : dependency dependency [...]  
    command  
    command  
    [...]
```

target: Es una etiqueta que define lo que queremos crear, actúa como nombre de la regla.

dependency: Es la lista de cosas requeridas para poder realizar esta regla. La lista puede estar compuesta por:

- uno o varios archivos fuente a compilar.
- una o varias librerías (ya como código objeto) que son necesarias para compilar nuestro código.
- una o varias reglas que tienen que realizar antes de llevar acabo esta.

command: Es la secuencia ordenada de comando necesarios para realizar la "compilación".

Variables en los Makefile:

```
BASEDIR = /home/vialef/myproject  
SRCDIR = $(BASEDIR)/src
```

GOOD

```
CC := gcc -o  
CC += -O2
```

BAD

```
CC = gcc  
CC = $(CC) -o  
*** Recursive variable `CC' references itself (eventually). Stop.
```


GCC & Makefile

Reglas Implícitas:

```
CC= gcc
CFLAGS= -Wall -ansi -D_XOPEN_SOURCE=500
```

```
OBJS = editor.o screen.o keyboard.o
HDRS = editor.h screen.h keyboard.h
```

```
# Editor
editor : $(OBJS)
    $(CC) $(CFLAGS) -o editor $(OBJS)
```

```
editor.o : editor.c $(HDRS)
    $(CC) -c $(CFLAGS) editor.c
```

```
screen.o : screen.c screen.h
    $(CC) -c $(CFLAGS) screen.c
```

```
keyboard.o : keyboard.c keyboard.h
    $(CC) -c $(CFLAGS) keyboard.c
```

```
# ALL
all: editor
```

```
# Clean
clean :
    rm editor $(OBJS)
```

Regla por Patrones:

```
src/%.o : src/%.c
    $(CC) -c $(CFLAGS) $< -o $@
```

\$< = *The name of the first dependency in a rule.*
\$@ = The filename of a rule's target

Development Tools

Development Tools

- Qué es un IDE y para que sirve?
- Eclipse
 - Features
 - Configuración del Proyecto
 - Debugging
 - Integracion con SVN: Subclipse
- Visual Studio 2005
 - Features
 - Configuración del Proyecto
 - Debugging
 - Integracion con SVN: Subclipse

Development Tools

¿Que es un IDE y para que sirve?

(From Wikipedia) An integrated development environment (IDE) also known as integrated design environment or integrated debugging environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of:

- a source code editor
- a compiler and/or an interpreter
- build automation tools
- a debugger

Algunos ejemplos son:

- **Eclipse** - Java, PHP, C/C++, COBOL, Perl, Ruby, Python, JavaScript, CSS, HTML
- **Netbeans** - Java, Scala, Clojure, PHP, C/C++, Ruby, Python, JavaScript, CSS, HTML
- **Vim** - Java, PHP, C/C++, Ruby, Python, Lisp, Perl, Pascal, JavaScript, CSS, HTML
- **Emac** - Java, PHP, C/C++, Ruby, Python, Lisp, Perl, Pascal, JavaScript, CSS, HTML
- **Visual Studio** - C#, F#, C/C++, ASP .Net, VB .Net, IronRuby, IronPython, JavaScript, CSS, HTML

Development Tools

Features:

- Syntax highlighting
- Include Assistant
- Code validations in coding time
- Autocomplete
- Integration with SCM (SVN, CVS, Git, Mercurial, etc..)
- Advance Source Navigation
- Advance Refactoring
- Automatic Makefile Generation

Project Configuration:

- Build Configuration
- Editor Configuration

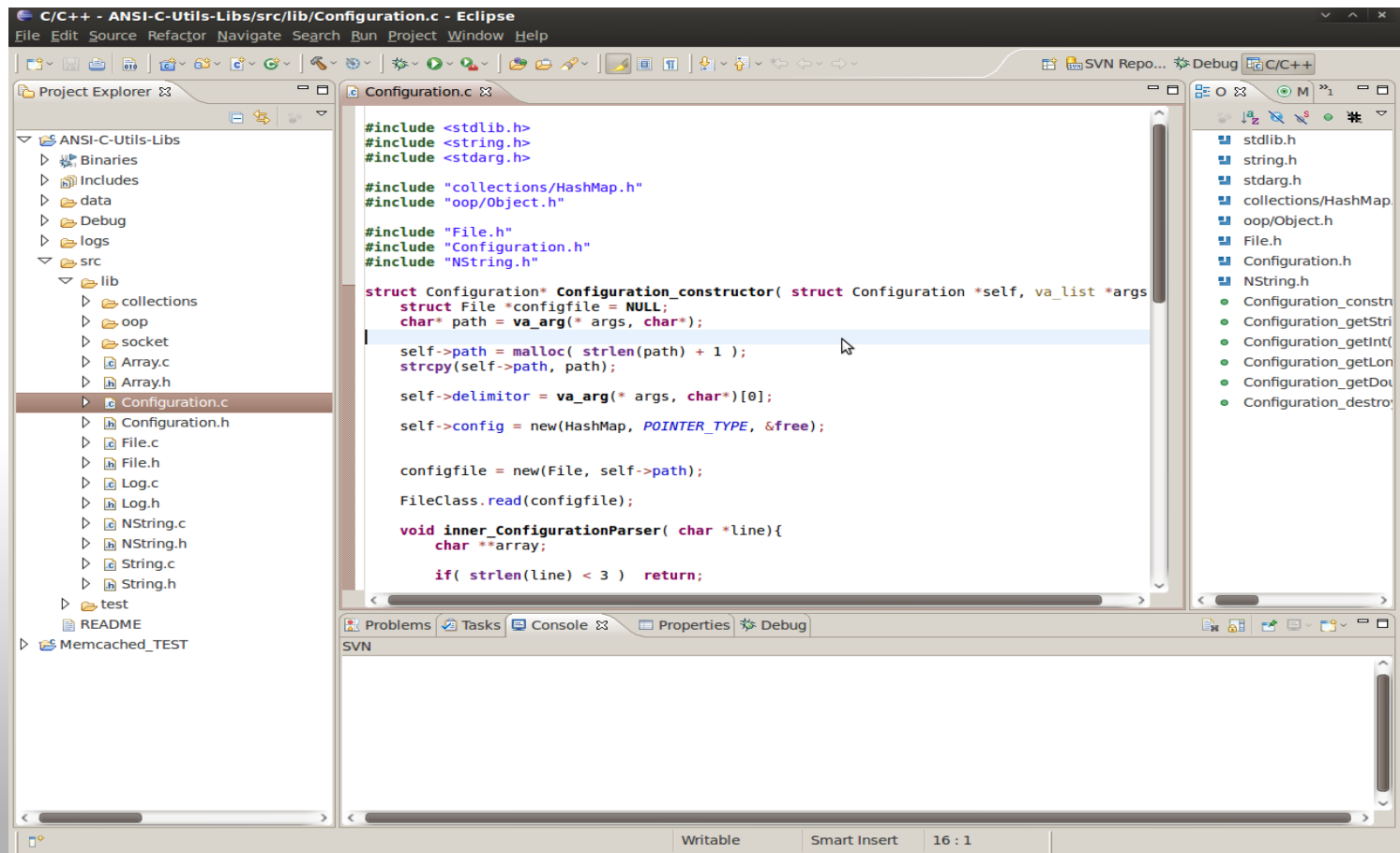
Debugger:

- Step by Step
- Breakpoints
- Variable View
- Watch Expresion
- Casting Type

Development Tools

Eclipse

Es un entorno de desarrollo integrado (IDE) de código abierto multiplataforma para el desarrollo en varios lenguajes. Esta plataforma, típicamente ha sido usada para desarrollar en Java, lenguaje con el que esta hecho, pero con el paso del tiempo se le han integrados mas lenguajes. Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.



Visual Studio






Development Tools

Development Tools

Features:

- Solución y Proyectos

Administrator\My Documents\Visual Studio 2005\Projects\C-Talks

Name ▲	Size	Type
 Console		File Folder
 HttpServer		File Folder
 C-Talks.ncb	11 KB	VC++ Intellisense Database
 C-Talks.sln	1 KB	Microsoft Visual Studio Solution
 C-Talks.suo	7 KB	Visual Studio Solution User Options

- Agregar Nuevos Elementos

Recomendación Personal: utilizar una estructura de directorios

- Project Configuration
- Release Mode vs Debug Mode
- Debugger:
 - Step by Step
 - Breakpoints
 - Watches
- "Autocompletar"
- Source Navigation - "Filters"
 - Directory ("virtual")
 - Files ("físico")
- Advance Refactoring
- MS Build transparente al programador

Development Tools

Views:

- Code Definition Windows
- Error List
- Output

Development Tools

Custom Features:

- Window -> New Windows & Split Windows (con cuidado!)
- Sintax Coloring
- Font's size & color
- Import & Export Settings

Qué extrañamos del Eclipse?

- Un autocompletar como la gente.
- Code Templates
- Auto-agregado de Matching Braces
- Chequeo de errores de compilación en Tiempo Real

Development Tools

Emulating Visual Studio?

- Slickedit
 - Compile errors in real-time.
 - Auto-Complete Braces
 - Code Style
 - Code Templates
 - More syntax coloring
 -

Development Tools

Valgrind

Es un conjunto de herramientas diseñadas para el debugeo de memoria y detección de memory leaks. Entre algunas cosas nos permite detectar:

- Uso de memoria no inicializada.
- Lectura/escritura de memoria que ha sido previamente liberada.
- Lectura/escritura fuera de los límites de bloques de memoria dinámica.
- Memory Leaks
- Race Condition

Para lograr esto, Valgrind no es mas que una Virtual Machine que permite compilar y recompilar en tiempo de ejecución. Es por ello que al correr un programa con Valgrind este puede ir varias veces mas lento de lo normal y consumir mucha mas memoria.

Una de sus limitaciones, es que no puede detectar acceso fuera de los limites de un bloque de *memoria estática*. Ej:

```
char buff[5];  
buff[5] = 0; /* Error! el rango de buff es de 0 a 4 */
```

Development Tools

Ejemplo con Valgrind

Acceso a un puntero NULL

```
17 char *buff1 = NULL;  
18  
19 buff1[0] = 3;  
--
```

```
fviale@fviale-Vostro-3500:~/Workspace/Personal/C/Test/Debug$ valgrind ./Test  
==6845== Memcheck, a memory error detector  
==6845== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.  
==6845== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info  
==6845== Command: ./Test  
==6845==  
==6845== Invalid write of size 1  
==6845==    at 0x406894: main (Test.c:19)  
==6845==   Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Development Tools

Ejemplo con Valgrind

Memory Leak y Acceso Fuera de los Limites de un Bloque Dinámico

```
17     char *buff1 = malloc(5);
18
19     buff1[5] = 3;
20
```

```
fviaie@fviaie-Vostro-3500:~/Workspace/Personal/C/Test/Debug$ valgrind ./Test
==6976== Memcheck, a memory error detector
==6976== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==6976== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==6976== Command: ./Test
==6976==
==6976== Invalid write of size 1
==6976==    at 0x4068A2: main (Test.c:19)
==6976==   Address 0x53ce045 is 0 bytes after a block of size 5 alloc'd
==6976==    at 0x4C2815C: malloc (vg_replace_malloc.c:236)
==6976==   by 0x406895: main (Test.c:17)
==6976==
==6976==
==6976== HEAP SUMMARY:
==6976==    in use at exit: 5 bytes in 1 blocks
==6976==   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
==6976==
==6976== LEAK SUMMARY:
==6976==    definitely lost: 5 bytes in 1 blocks
==6976==    indirectly lost: 0 bytes in 0 blocks
==6976==    possibly lost: 0 bytes in 0 blocks
==6976==    still reachable: 0 bytes in 0 blocks
==6976==    suppressed: 0 bytes in 0 blocks
```

Bibliografía utilizada

- <http://es.wikipedia.org/wiki/Errno.h>
- <http://linux.die.net/man/3/errno>
- <http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html#C-Extensions>
- http://es.wikipedia.org/wiki/C%C3%A1lculo_lambda
- <http://es.wikipedia.org/wiki/Thread-Safety>
- <http://www.unix.org/whitepapers/reentrant.html>
- <http://valgrind.org/docs/manual/manual.html>
- <http://en.wikipedia.org/wiki/Valgrind>