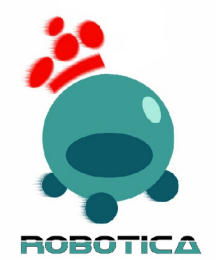
Herramientas de programación C/C++

Julio M. Vega y José M. Cañas

http://jde.gsyc.es



Grupo de Robótica, 21 Mayo 2009

Contenidos

- Introducción
- Construcción de una aplicación
  - Compilación y enlazado
  - Bibliotecas dinámicas y estáticas
  - Tablas de símbolos
  - Make y makefiles
- Depuración
  - GNU Debugger (GDB)
  - Valgrind

Introducción

- Robótica tiene mucho de programación
- La inteligencia de un robot está en su software
- Depurar nuestro software es importante y necesario
- A veces encontrar errores es difícil
- Aplicaciones como *gdb* o *valgrind* ayudan a encontrar errores
- Errores de programación o de funcionalidad

Construcción de una aplicación

Compilación y enlazado

- Código fuente (miprograma.c)
- Compilar
- Código objeto (miprograma.o)
- Enlazar
- Ejecutable, *main*
- a.out, elf (miprograma)

- `gcc -c file1.c file2.c file3.c`
- `gcc -o outputfile file1.o file2.o file3.o`
- `gcc -o outputfile file1.c file2.c file3.c`

Tablas de símbolos

- Funciones y variables tienen su dirección
- gcc
- Opciones de compilación `-Wall`
- ¿Dónde buscar las cabeceras? -I
- `nm`

Bibliotecas

- Cabeceras
- Bibliotecas estáticas y dinámicas
- Enlazador dinámico `ld.so`
  - LD_LIBRARY_PATH
  - `/etc/ld.conf.so` y `ldconfig`
- Opciones de enlazado
  - ¿Dónde buscar las bibliotecas? -L
  - ¿Qué bibliotecas buscar? -l
- Dependencias `ldd`

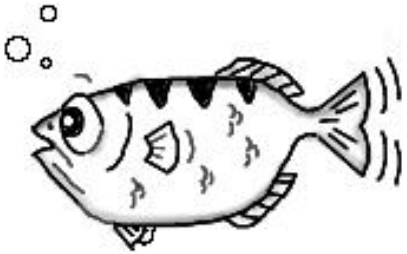
Make y Makefile

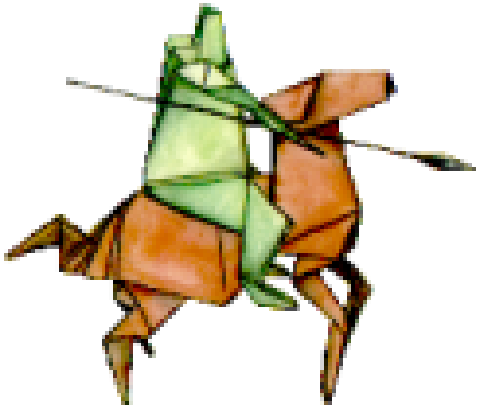
- Automatizar todo el proceso de construcción de la aplicación
- Objetivos, requisitos y reglas
- TAB
- Variables
- `pkg-config`
- No sólo para construir una aplicación, también otras cosas

```
JDEDIR =
INC-DIR = -I. -I/usr/include/opencv -I$(JDEDIR)/include/jderobot `pkg-config --cflags libglade-2.0 gtkglext-1.0 gdkglext-1.0 gthread-2
LIB-DIR = -L. -L/usr/lib/jderobot `pkg-config --libs libglade-2.0 gtkglext-1.0 gdkglext-1.0 gthread-2.0`
GCC = gcc
CFLAGS = -g -pedantic -Wall
LIBS = -lm -lX11 -lcv -lcxcore -lhighgui -lcvaux -lcolorspaces
COMPONENT= opencvdemo2

$(COMPONENT): $(COMPONENT).o
        $(GCC) -shared -Wl,-soname,$(COMPONENT).so $(LIB-DIR) -o $(COMPONENT).so $(COMPONENT).o $(LIBS)

$(COMPONENT).o: $(COMPONENT).c
        $(GCC) $(CFLAGS) $(INC-DIR) -fPIC -c $(COMPONENT).c

clean:
        rm -f *.o $(COMPONENT).so
```

Depuración

- Depurar es localizar y reparar errores del código fuente
- Proceso necesario en la creación de nuevo software
- Las aplicaciones de depuración nos ayudan a identificar los errores





GNU Debugger (GDB)

- La mayoría de las distribuciones Linux vienen con este *debugger*
- GDB nos permite:
- Ver la estructura interna de un programa
- Imprimir valores de variables
- Establecer puntos de ruptura (*breakpoints*)
- Avanzar paso a paso en el código

Compilación

- Para que nuestra aplicación pueda ser depurada con GDB necesitamos indicárselo
- A la hora de compilar, lo haremos con *gcc* o *g++*
- Añadimos una opción extra -g y la compilación incluirá información de depuración

Ejecución

- GDB se lanza desde consola con el comando gdb
- Y le pasamos el nombre de nuestra aplicación como parámetro, por ejemplo gdb app
- O ya dentro de la propia consola *GDB*, se lo indicamos con el comando file app
- Y para empezar la ejecución de nuestro programa, con el comando run o r

Ejemplo de ejecución

- Si no hay ningún fallo, la aplicación se ejecutará por completo
- Pero si hay algo mal, GDB interrumpirá la ejecución y tomará el control
- Así nos permite examinar el estado de todo y podremos encontrar porqué falla
- Veremos un códido de ejemplo... Comandos *run*, *list*, *continue*, *next*, *help*

Uso de breakpoints

- Para ver qué ocurre en un determinado punto, podemos indicar la línea en concreto
- GDB interrumpirá la ejecución cuando llegue a tal punto
- Sintaxis: break nombreFichero.c:numeroLínea
- Si establecemos condiciones: break nombreFichero.c:numeroLínea if condicion
- Podemos hacer también seguimiento: watch condicion

Pila de llamadas

- La pila de llamadas es un segmento de memoria que utiliza la estructura pila (*stack*)
- Donde almacena información sobre las llamadas a subrutinas actualmente en ejecución en el programa en proceso
- Cada vez que una nueva subrutina es llamada, se apila una nueva entrada con información sobre ésta
- Sintaxis: bt (*backtrace*)
- Se nos mostrará un listado de llamadas, por orden de antigüedad (marco o *frame*)
- Para ver las variables del marco actual: info locals
- Y podemos cambiar de marco: frame numFrame

Valgrind

- Conjunto de aplicaciones para detectar errores en el tratamiento de memoria y gestión de threads
- Valgrind nos permite:
- Detectar errores en memoria
- Detectar errores en manejo de threads
- Analizador de caché y predicción de ramificación/es
- Analizador de pila (*heap*)

Encontrar problemas de memoria

- Antes de nada, instalar Valgrind (viene en repositorio)
- Éstos son los peores problemas! Sólo se dan cuando hacemos algo fuera de límites
- En C/C++ no tenemos recolector de basura
- La liberación de memoria es un aspecto muy importante
- Nos centraremos en la utilidad memcheck
- Otras utilidades: *Cachegrind*, *Callgrind*, *Massif*, *Helgrind*, ...

Memcheck

- Nos permitirá comprobar el uso de memoria
- Obtener un listado de *free/malloc*: valgrind –tool=memcheck application
- Si tenemos algún escape de memoria, la relación *free/malloc* será incoherente
- Opción *leak-check*. Obtener un listado de *alloc/malloc/new* que no tiene un free emparejado
- Sintaxis: valgrind –tool=memcheck –leak-check=yes application
- Para un listado más exhaustivo añadimos opción –show-reachable=yes

Otros mensajes del memcheck

- *Invalid read/write of size X*. Escrituras/lecturas erróneas
- *Conditional jump or move depends on uninitialised value(s)*. Variable no inicializada en una regla condicional
- *Invalid free()*. Puntero ya liberado
- *Mismatched free() / delete / delete []*. Liberación de memoria incorrecta