ANSI C Modular

Desarrollando software extensible en C

Juan Pedro Bolívar Puente

granada.hacklabs.org

Diciembre 2008

Índice

1. Introducción
2. Bibliotecas reutilizables
3. Polimorfismo
4. Plugins

Índice

1. Introducción
2. Bibliotecas reutilizables
3. Polimorfismo
4. Plugins

¿Por qué C?

A veces no es sólo masoquismo

Desventajas

Muy bajo nivel, poca funcionalidad.

Ventajas

Lenguaje universal

- ANSI C es el lenguaje más **portable** que existe. (sí, más que Java)
- Fácil **interfaz** con todos los lenguajes de programación existentes

Lenguaje ligero

- Biblioteca estándar y código generado pequeño
- Compilación y ejecución "*determinista*"

Metodología básica

Orientando a objetos

¿Cómo conseguir modularidad?

- C estructurado
  Agrupando *funciones* en módulos.
- C *orientado a objetos*
  Agrupando funciones *y datos* en módulos
  ⇒ Tipos de Datos Abstractos

¿Orientación a objetos?

- C no es orientado a objetos...
- ...pero podemos aplicar una metodología
  ⇒ Tipos de Datos Abstractos

Metodología básica

Construyendo Tipos de Datos Abstractos

Todos los tipos de datos se implementan como una estructura con unos métodos asociados.

Nomenclatura consistente

1. `tipo_s` para la estructura.
2. `tipo_t` para la referencia.

   `tipo_t` puede ser la estructura para objetos básicos que queramos tener en pila
3. `tipo_metodo` para los métodos del tipo.
4. Métodos comunes:
   - `tipo_create` para el creador de instancia.
   - `tipo_create_with...` para los creadores con parámetros.
   - `tipo_destroy` para el reciclador.
   - `tipo_initialize` para el constructor.
   - `tipo_finalize` para el destructor.

Ejemplo de Tipo de Dato Abstracto

linea.h

```
struct linea_s {
  int x0;
  int y0;
  int x1;
  int y1;
};
typedef struct linea_s* linea_t;

status_t linea_initialize (struct linea_s*);
void linea_finalize (struct linea_s*);
linea_t linea_create ();
void linea_destroy (linea_t);
void linea_dibujar (linea_t);
contorno_t linea_contorno (linea_t);
```

Ejemplo de Tipo de Dato Abstracto

linea.c

```
status_t linea_initialize (struct linea_s* l) {
  memset (l, 0, sizeof (struct linea_s));
  return OK;
}

void linea_finalize (struct linea_s*) {}

linea_t linea_create () {
  linea_t l = malloc (sizeof (struct linea_s));
  return l && linea_initialize (l) ? l : NULL;
}

void linea_destroy (linea_t l) { free (l); }
...
```

Índice

1. Introducción
2. Bibliotecas reutilizables
3. Polimorfismo
4. Plugins

Bibliotecas reutilizables

El problema de la compatibilidad binaria

Un paso natural es agrupar distintos TDA's en un biblioteca reutilizable

Problema

Queremos que que no haya que cambiar o recompilar el código cliente cuando modificamos la biblioteca.

Solución

¡Compatibilidad Binaria!

Compatibilidad binaria

Construyendo una biblioteca reutilizable

Compatibilidad binaria *externa*

Los módulos compilados pueden enlazarse y ejecutarse en el SO

- API estable
  No cambiamos los nombres ni la signatura de los métodos
- ABI estable
  El compilador y el enlazador siguen un estándar para el formato binario (`elf`, `a.out`, `...`)

Compatibilidad binaria *interna*

Modificaciones en la biblioteca no imponen una nueva estructura de la pila en el cliente

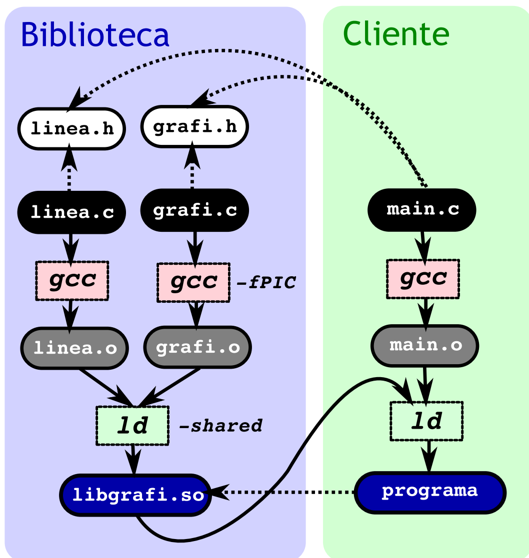


Figura: Flujo de construcción de una bilbioteca

Manifestación del problema

Código cliente

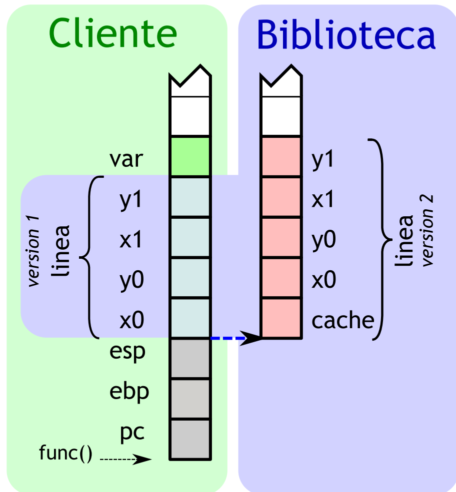
```
struct linea_s linea;
int var = 5;
linea_initialize (&linea);
linea.x1 = linea.y1 = 1;
linea_dibujar (&linea);
linea_finalize (&linea);
```

linea.h versión 2

```
struct linea_s {
  int cache;
  ...
};
```

Manifestación del problema

El secreto está en la pila



Estado Version 1

```
linea.x0 = 0
linea.y0 = 0
linea.x1 = 1
linea.y1 = 1
var      = 5
```

Estado Version 2

```
linea.cache = 0
linea.x0    = 0
linea.y0    = 1
linea.x1    = 1
linea.y1    = 0
var         = 0
```

Solución

Ocultación durante la compilación

El código compilado del cliente asume nada sobre los datos de la biblioteca.

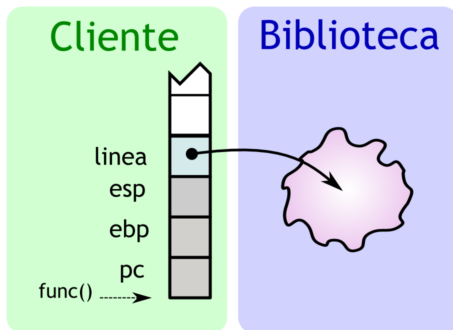
1. Encapsulación
   Se proveen accesores para todos los miembros públicos
2. Heap
   Se prohibe crear el objeto en la pila y *la biblioteca se encarga de crear la instancia*
3. Ocultación
   Se oculta la estructura mediante un *puntero opaco* (patrón *Pimpl*)

Coste de eficiencia ⇒ excepciones en tipos *simples* y *estables*

Solución

Ocultación durante la compilación

El código compilado del cliente asume nada sobre los datos de la biblioteca.



Aplicando la solución

linea.h

```
/* declaracion anticipada */
struct linea_s;
typedef struct linea_s* linea_t;

/* quitamos intialize y finilize */
linea_t linea_create ();
void linea_destroy (linea_t);

/* anadimos accesores */
int linea_get_x0 (linea_t);
void linea_set_x0 (linea_t, int);
...
void linea_dibujar (linea_t);
contorno_t linea_contorno (linea_t);
```

Aplicando la solución

linea.c

```
struct linea_s { /* declaracion real */
  int x0, y0, x1, y1;
};
linea_t linea_create () {
  /* el cliente no conoce sizeof (struct linea_s) */
  linea_t linea = malloc (sizeof (struct linea_s));
  if (!linea) return NULL;
  memset (linea, 0, sizeof (linea));
  return linea;
}
void linea_destroy (linea_t self) { free (linea); }
int linea_get_x0 (linea_t self) { return self->x0; }
void linea_set_x0 (linea_t self, int x0) {
  self->x0 = x0; }
...
```

Aplicando la solución

Cliente antiguo

```
struct linea_s linea;
int var = 5;
linea_initialize (&linea);
linea.x1 = linea.y1 = 1;
linea_dibujar (&linea);
linea_finalize (&linea);
```

Cliente nuevo

```
linea_t linea;
int var = 5;
linea = linea_create ();
linea_set_point_a (linea, 1, 1);
linea_dibujar (linea);
linea_destroy (linea);
```

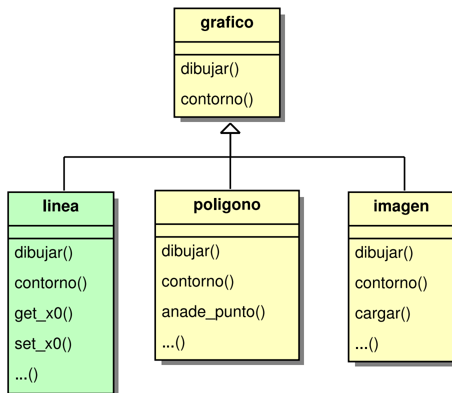
Índice

1. Introducción
2. Bibliotecas reutilizables
3. Polimorfismo
4. Plugins

Polimorfismo

Logrando genericidad

Una familia de tipos que tiene comportamiento especializado para una semántica común.



Polimorfismo

Despacho dinámico

Despacho dinámico

Al llamar un método, se decide el código a ejecutar en tiempo de ejecución

Punteros a funciones

Permiten guardar la dirección de una función en una variable para su posterior llamada.

Uso

```
typedef double (*funcion_unaria_t) (double);
funcion_unaria_t func;
func = sqrt;
val = func (3);
```

La interfaz y el comportamiento

Comportamient de un objeto

El comportamiento de un objeto se agrupa en una tabla de punteros a funciones que agrupa las funciones que implementan una interfaz.

Interfaz genérica de grafico_t

grafico.h

```
struct grafico_impl_s {
  void*       (*create) ();
  void        (*destroy) (void*);
  void        (*dibujar) (void*);
  contorno_t (*contorno) (void*);
};
```

Ajustando la implementación a la interfaz

linea.h

```
/* cambiamos el tipo por void* para generificar */
typedef void* linea_t;

/* exportamos la implementacion de la linea */
extern grafico_impl_s linea_impl;
```

linea.c

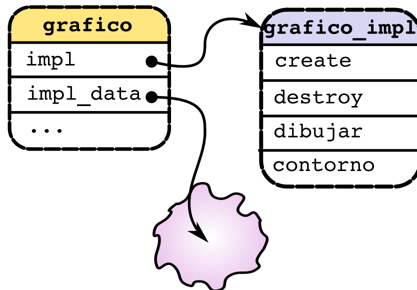
```
/* definimos la implementacion de la linea */
grafico_impl_s linea_impl = {
  .create = linea_create,
  .destroy = linea_destroy,
  .dibujar = linea_dibujar,
  .contorno = linea_contorno
};
```

Componiendo el objeto

¿Cómo se compone la jerarquía en un único objeto?

Primera aproximación

La superclase guarda y construye los datos de la subclase

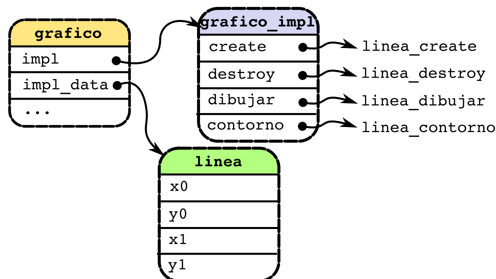


Componiendo el objeto

¿Cómo se compone la jerarquía en un único objeto?

Primera aproximación

La superclase guarda y construye los datos de la subclase



Componiendo el objeto

¿Cómo se compone la jerarquía en un único objeto?

Primera aproximación

La superclase guarda y construye los datos de la subclase

grafico.h

```
struct grafico_s;
typedef struct grafico_s grafico_t;
grafico_t grafico_create (struct grafico_impl_s*);
void grafico_destroy (grafico_t);
void grafico_dibujar (grafico_t);
contorno_t grafico_contorno (grafico_t);
```

grafico.c

```
struct grafico_s {
  grafico_impl_s* impl;
  void*           impl_data;
}
grafico_t grafico_create (struct grafico_impl_s* impl) {
  grafico_t g = malloc (sizeof (struct grafico_s));
  if (!g) return NULL;
  g->impl_data = impl->create ();
  if (!g->impl_data) { free (g); return NULL; }
  g->impl = impl;
  return g;
}
void grafico_destroy (grafico_t self) {
  self->impl->destroy (self->impl_data);
  free (self);
}
void grafico_dibujar (grafico_t self) {
  self->impl->dibujar (self->impl->data);
}
contorno_t grafico_contorno (grafico_t) {
  return self->impl->contorno (self->impl);
}
```

El punto de vista del cliente

Tratamiento homogéneo de familias de tipos

Podemos manejar lineas y poligonos de forma homogenea

```
void actualizar_grafico (contorno_t c, grafico_t g) {
  if (contorno_interseca (grafico_contorno (g), c))
    grafico_dibujar (g);
}
int main () {
  grafico_t g1, g2;
  contorno_t pantalla = {0, 0, 640, 480};
  g1 = grafico_create (g1, &linea_impl);
  g2 = grafico_create (g2, &poligono_impl);
  actualizar_grafico (pantalla, g1);
  actualizar_grafico (pantalla, g2);
  grafico_destroy (g1);
  grafico_destroy (g2);
}
```

Casting

Extendiendo la clase base

¡Pero y cómo consigo acceder a los métodos de las subclases!

Anadiendo accesores para los datos de la subclase

grafico.h

```
void* grafico_child (grafico_t);
```

Et voilá...

```
grafico_t g1 = grafico_create (&linea_impl);
linea_set_coords (grafico_child (g1),
0, 0, 640, 480);
```

Casting e información de tipos

¡Cuidado!

El compilador pierde la información de tipos

```
grafico_t g1 = grafico_create (&poligono_impl);
linea_set_coords (grafico_child (g1), /* Error !! */
                  0, 0, 640, 480);
```

Solución RTTI

Anadir otro accesor para la implementación

```
void visitante_grafico (grafico_t g1) {
  if (grafico_impl (g1) == &linea_impl)
    linea_set_coords (grafico_child (g1),
                      0, 0, 640, 480);
}
```

Balance

Desventajas

- Se puede hacer un *upcast* pero no un *downcast*
- Fragmentación de la memoria
- Jerarquía de varios niveles difíciles

Ventajas

- Mantiene compatibilidad binaria
- Puede almacenarse el subtipo para postponer su construcción
- Podría intercambiarse la implementación en tiempo de ejecución

Índice

1. Introducción
2. Bibliotecas reutilizables
3. Polimorfismo
4. Plugins

Sistema de Plugins

Esto sí que es extensibilidad

Objetivo

¡Añadir funcionalidad al programa sin recompilarlo!

Mecanismo

Cargar una subclase desde un fichero

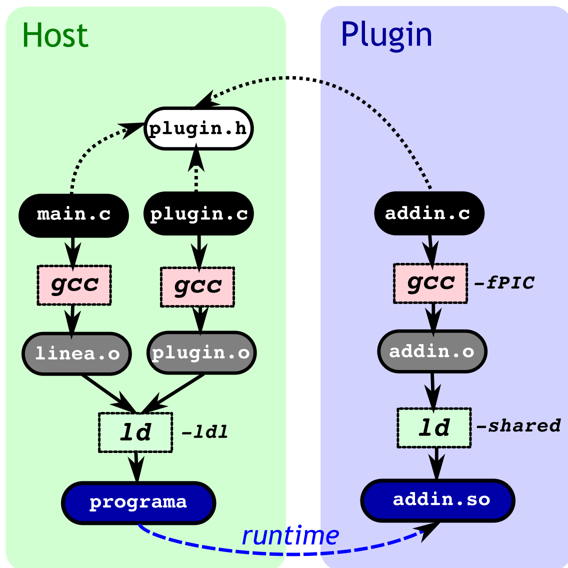


Figura: Flujo de compilación de un sistema de plugins

Carga dinámica

Carga dinámica

Abrimos un archivo *shared object* y buscamos símbolos (funciones y datos globales) en tiempo de ejecución.

POSIX libdl `<dlfcn.h>`

1. `void* dlopen (const char* fname, int flags)`
   Devuelve un *handle* a la biblioteca `fname`.
   `flags` debe incluir al menos:
   - `RTLD_LAZY`: Resolución perezosa de los símbolos de función
   - `RTLD_NOW`: Resuelve todos los símbolos directamente
2. `void* dlsym (void* lib, const char* symname)`
   Devuelve un puntero a un símbolo de la biblioteca
3. `int dlclose (void* lib)`
   Libera la referencia a la biblioteca

Ejemplo de carga dinámica

Ejemplo

```
void *handle;
double (*cosine)(double);
char *error;
handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
if (!handle) {
  fputs (dlerror(), stderr);
  exit(1);
}
cosine = dlsym(handle, "cos");
if ((error = dlerror()) != NULL)  {
  fputs(error, stderr);
  exit(1);
}
printf ("%f\n", (*cosine)(2.0));
dlclose(handle);
```

Retos del sistema de plugins

...no es tan fácil...

- Polimorfismo

  Un plugin es la implementación de una interfaz

- Vesiones

  Nuestro programa no debe colgarse al cargar plugins viejos

- Metadatos

  ¿Identificación, quién hizo el plugin, cuando, licencia?

- Multi-plugins

  Un mismo *shared-object* puede proporcionar varios plugins

- Homogeneidad

  Un mismo sistema de plugins para distintos tipos de plugin

Punto de vista del plugin

Solución

El plugin se encarga de instalarse a sí mismo

1. El plugin conoce la interfaz.
   Ej, `plugin.h`
2. El plugin sólo publica un método.
   Ej, `void init_plugin (plugin_mgr_t)`
3. En `init_plugin` se registra ofreciendo:
   - La versión del sistema definida en `plugin.h`
   - Los metadatos
   - Un puntero a la implementación. Ej, `struct grafico_impl_s*`
4. Puede registrar varias veces

Plugin de ejemplo

addin.c

```c
#include <plugin.h>

struct grafico_impl_s addin_impl = {
  .create   = addin_create,
  .destroy  = addin_destroy,
  .dibujar  = addin_dibujar,
  .contorno = addin_contorno
};

void init_plugin (plugin_mgr_t pmgr)
{
  plugin_mgr_register_grafico (pmgr,
      PLUGIN_SYSTEM_VERSION, "addin", &addin_impl);
}

/* implementacion del plugin ... */
```

Punto de vista del host

`plugin_mgr_t`, *administra los plugins*

- `int plugin_mgr_load (const char*)`
  Carga los plugins de un *shared object*
- `plugin_t plugin_mgr_find (const char*)`
  Busca y devuelve un plugin
- Constructor y destructor...
- Iteradores sobre los plugins...

`plugin_t`, *accede a un plugin*

- `struct grafico_impl_s* plugin_get_grafico_impl (plugin_t)`
  Obtiene la implementacion de un plugin grafico
- `const char* plugin_get_name (plugin_t)` Accesores para los metadatos
- Los constructores y destructores son privados

Ejemplo de host

Ejemplo

```
plugin_mgr_t pmgr;
plugin_t p;

pmgr = plugin_mgr_create ();
plugin_mgr_load ("addin.so");
p = plugin_mgr_find (pmgr, "addin");
if (p && plugin_get_grafico_impl (p)) {
  g = grafico_create (plugin_get_grafico_impl (p));
  if (g) {
    grafico_dibujar (g);
    grafico_destory (g);
  }
}
plugin_mgr_destroy (pmgr);
```

plugin.h

```
#define PLUGIN_SYSTEM_VERSION   1
enum plugin_type_e { PLUGIN_GRAFICO = 0 };
typedef enum plugin_type_e plugin_type_t;
typedef void (*init_plugin_func_t) (plugin_mgr_t);

struct plugin_s;
typedef struct plugin_s* plugin_t;
struct plugin_manager_s;
typedef struct plugin_mgr_s* plugin_mgr_t;

plugin_mgr_t plugin_mgr_create ();
void plugin_mgr_destroy (plugin_mgr_t);
void plugin_mgr_register_grafico (plugin_mgr_t self,
  int sys_version, const char* name, struct grafico_impl_s*);
void plugin_mgr_load (plugin_mgr_t self, const char* file);
plugin_t plugin_mgr_find (plugin_mgr_t self,
                          const char* name);

struct grafico_impl_s*
plugin_get_grafico_impl (plugin_t self);
const char* plugin_get_name (plugin_t self);
```

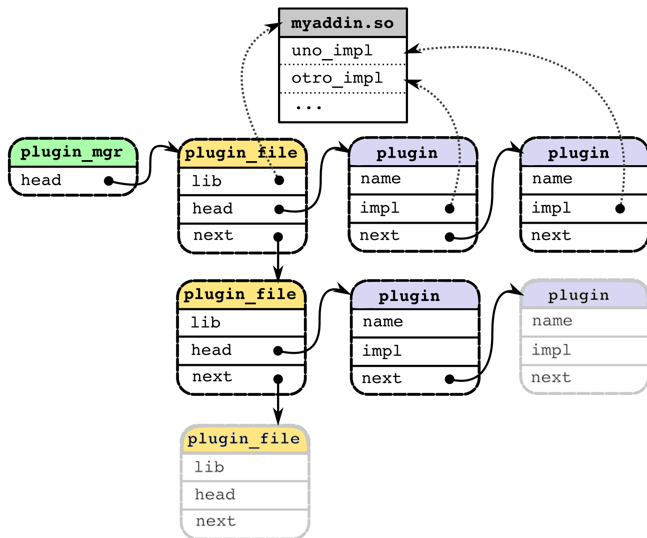


Figura: Diseño del sistema de plugins

plugin.c

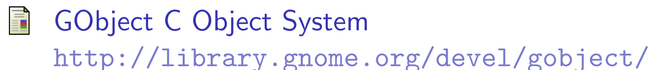
```
void plugin_mgr_load (plugin_mgr_t self, const char* file) {
  init_plugin_func_t init_func;

  self->current = plugin_file_create (file);
  if (self->current) {
    init_func = dlsym (self->current->lib, "init_plugin");
    if (init_func)
      init_func (self);
    else
      fprintf (stderr, "plugin␣sin␣inicializador\n", file);

    if (self->current->head) {
      self->current->next = self->head;
      self->head = self->current;
    } else {
      plugin_file_destroy (self->current);
      fprintf (stderr, "plugin␣vacio\n", file);
    }
    self->current = NULL;
  }
}
```

Resumen

1. Tipos de Datos Abstractos
   - Agrupamos métodos y datos
2. Bibliotecas reutilizables
   - Compatibilidad binaria necesaria
   - Utilizar punteros opacos
3. Polimorfismo
   - Permite usar tipos distintos de forma homogenea
   - Se usa una estructura de punteros a funciones
4. Plugins
   - Extensibilidad a posteriori
   - Se usa carga dinámica y registro delegado

Recursos adicionales

Object Orientation in ANSI C
Axel-Tobias Schreiner
1993, http://www.cs.rit.edu/~ats/books



GObject C Object System
http://library.gnome.org/devel/gobject/

Program Library HOWTO
David A. Wheeler
http://tldp.org/HOWTO/Program-Library-HOWTO

KDE Binary Compatibility Policy
http://techbase.kde.org/Policies/Binary_Compatibility_Issu

¿Preguntas?

Muchas gracias por su atención.