uc3m | Universidad Carlos III de Madrid

C Misceláneo: Cuestiones Avanzadas

Contenidos

Punteros a funciones

- Idea y Usos
- Funciones de retro-llamada (callbacks)
- Tablas de punteros a funciones

Librerías externas

- Símbolos y enlazado
- Enlazado dinámico frente a estático
- Enlazado de librerías externas
- Creación de librerías

Punteros a funciones

- En algunos lenguajes de programación, las funciones son entidades de primer orden
  - pueden pasarse a funciones, ser devueltas en funciones
- En el caso de C, la función no es una variable. Pero, es posible trabajar con punteros a funciones.
- Pregunta: En qué escenarios es interesante trabajar con punteros a funciones? Cuál es su utilidad?
- Ejemplos de declaración:
  - `int (*fp )( int )/* Fíjate en el () */`
  - `int (*fp )(void*,void*)`
- Los punteros a función pueden ser asignados, pasados a y desde funciones y almacenados en arrays.

Retro-llamadas (Callbacks)

- Definición: Una retro-llamada es un pieza de código que se pasa a una función. En C, se implementan con punteros a funciones.

  Ejemplo:

```
void qsort(void* arr, int num,int size,
                  int(*fp )(void* pa, void* pb))
```

- `qsort`() función de la librería estándar que puede ordenar un array de cualquier tipo de datos.

  - Pregunta: Cómo lo consigue? retro-llamadas.

- `qsort`() llama a una función de ordenación definida por el usuario.
  - La función recibe dos argumentos y retorna (`<0`,`0`,`>0`) dependiendo del orden relativo de los dos elementos.

Retro-llamada: el ejemplo `qsort`

```
/* definición de una tabla */
int arr[]={10,9,8,1,2,3,5};
/* función que va a ser retro-llamada */
int asc(void* pa, void* pb)
        { return(*(int *)pa - *(int *)pb);}
/* función que va a ser retro-llamada */
int desc(void* pa, void* pb)
         { return(*(int *)pb -*( int*)pa );}
 /* ordenación ascendente */
qsort(arr,sizeof(arr)/sizeof(int),sizeof(int),asc);
 /* ordenación descendente */
qsort(arr,sizeof(arr)/sizeof(int),sizeof(int),desc);
```

Aplicar función genérica (uso)

- Considerar la siguiente lista enlazada:

```
struct node {int data ;
             struct node* next ;  };
```

- Y una función 'apply' definida como sigue:

```
void apply(struct node* phead,
     void (* fp )(void* , void* ), void* arg)
   { struct node* p=phead ;
    while ( p !=NULL) {
     fp(p,arg);  /* Otra alternativa: (* fp )(p,arg)
*/
     p=p->next ; }
    }
```

Continuación

- Usando la función:

```
/* función retro-llamada */
void print(void* p, void* arg)
   {struct node* np =(struct node *)p;
    printf("%d ",np->data);
   }

struct node* phead ;
/* Inicialización de la lista */
apply(phead, print, NULL);
```

Tabla de punteros a funciones

- Ejemplo: Considera el siguiente caso donde se llaman diferentes funciones dependiendo de su tipo.

```
enum TYPE{SQUARE,RECT,CIRCILE ,POLYGON};
struct shape {float params[MAX];
              enum TYPE type ;};
void draw(struct shape* ps){
     switch (ps->type){
       case SQUARE: draw_square(ps);break ;
       case RECT: draw_rect (ps); break;
       }
 }
```

Tabla de punteros a funciones (2)

- Lo mismo se puede conseguir usando una tabla de punteros a funciones

```
void (*fp[4])(struct shape* ps)=
{&draw_square,&draw_rec,&draw_circle,&draw_poly};

typedef void(*fp)(struct shape* ps) drawfn;

drawfn fp[4]
={&draw_square,&draw_rec,&draw_circle,&draw_poly};

void draw(struct shape* ps){
 (* fp[ps->type])(ps); /* invoca la función correcta */}
```

Símbolos y bibliotecas

- Hay muchas bibliotecas externas al código que ofrecen muchas funcionalidades

  Por ejemplo: C standard library

- Los programas acceden a esas funciones y variables a través de símbolos
- Las cabeceras de archivo contienen declaraciones y prototipos que se asocian a símbolos en tiempo de compilación
- Los símbolos se enlazan a definiciones en bibliotecas externas durante el enlazado

Funciones y variables como símbolos

- Considera el "`holamundo`" que aparece abajo:

```
#include <stdio.h>
const char msg[] = "Hello, world." ;
 int main (void){
     puts(msg);
     return 0;
   }
```

- ¿Qué variables y funciones están declaradas globalmente? `msg`, `main`(), `puts`(), y otras en `stdio.h`

Funciones y variables como símbolos

- Compilemos, pero no enlacemos, el fichero `hello.c` para crear `hello.o`:

```
athena% gcc -Wall -c hello.c -o hello.o
```

- `-c`: compila, pero no enlaza `hello.c` ; se compila el código en instrucciones máquina pero hace que el programa sea ejecutable
- No se asignan ni las direcciones a cada línea de código, variables estáticas ni globales
- El enlazado resuelve los símbolos definidos (como en C standard lib) y hace que el código sea ejecutable
- Se necesita hacer un enlazado sobre `hello.o`, (con `gcc` o `ld`) para asignar memoria a cada símbolo

Funciones y variables como símbolos

- Veamos los símbolos incluidos en hello.o:

```
athena% nm hello.o
```

- Salida:

```
0000000000000000 T main
0000000000000000 R msg U puts
```

- ’T’: (text) code; ’R’: read-only memory; ’U’: undefined symbol
- Los ceros antes de los símbolos muestran que no se han colocado aún las direcciones de memoria
- Los símbolos sin definir se definen externamente, y se resuelven durante el enlazado.

Funciones y variables como símbolos

- ¿Porqué no hay símbolos para otras declaraciones en `stdio.h`?
- El compilador no se molesta creando símbolos para funciones que no van a ser usadas (ahorra espacio)
- ¿Qué pasa cuando enlazamos?

```
athena% gcc -Wall hello.o -o hello
```

- Se reserva memoria para los símbolos definidos
- Quedan por definir los símbolos que se encuentran en librerías externas (como libc de C standard library)

Funciones y variables como símbolos

- Veamos entonces los nuevos símbolos:

```
athena% nm hello
```

- Salida: (otros símbolos por defecto)

```
0000000000400524 T main
000000000040062c R msg
                 U puts@@GLIBC_2.2.5
```

- Aparecen las direcciones para símbolos estáticos (definidos en tiempo de compilación).
- El símbolo `puts` queda localizado en la GLIBC_2.2.5 (GNU C standard library) librería compatida
- El símbolo `puts` se asignará en tiempo de ejecución

Enlazado estático y dinámico

- Funciones y variables globales deben de ser asignadas antes de ser usadas
- Pueden asignarse en tiempo de compilación (`static`) o en tiempo de ejecución (`shared`)
- Ventajas/desventajas para ámbos
- Símbolos en el mismo archivo, otros archivos `.o`, o bibliotecas estáticas (carpetas, y ficheros .a)
  – enlazado estático
- Símbolos en bibliotecas compartidas (ficheros `.so`)
  – enlazado dinámico
- `gcc` enlaza bibliotecas contra librerías compartidas (por defecto). También es posible hacer contra librerías estáticas con el flag -static

Enlazado estático

- ¿Que pasa si se enlaza estáticamente contra la biblioteca?

```
athena% gcc -Wall -static hello.o -o hello
```

- Ahora los símbolos muestran:

```
0000000000004014c0 W puts
0000000000400304 T main
000000000046cd04 R msg
```

- 'W': linked to another defined symbol

Enlazado estático

- En tiempo de enlazado, los símbolos estáticamente enlazados se añaden al ejecutable
- También crece mucho más el tamaño del ejecutable

  (static – 688K, dynamic – 10K)
- El ejecutable resultante no depende de localizar archivos de bibliotecas externas en tiempo de ejecución.
- Para usar una nueva versión de una librería, hay que recompilar

Enlazado dinámico

- El enlazado dinámico sucede en tiempo de ejecución
- Durante la compilación, el enlazador busca los símbolos en las bibliotecas externas compartidas
- Símbolos compartidos son cargados como parte del arranque (antes de `main`())
- Requiere una biblioteca externa para definir los símbolos exactamente como se espera la declaración
- Al cambiar una función en la biblioteca compartida puede romper un programa
- Se puede usar la información de versión para minimizar ese problema
- Esa es la razón por la cual muchas librerías como `libc` raramente se modifican o eliminan funciones, incluso cosas rotas como `gets`()

Enlazando bibliotecas externas

- Programs linked against C standard library by default
- To link against library `libnamespec.so` or `libnamespec.a`, use compiler flag -lnamespec to link against library
- Library must be in library path (standard library directories + directories specified using `-L` directory compiler flag)
- Use `-static` for force static linkage
- This is enough for static linkage; library code will be added to resulting executable

Cargando bibliotecas compartidas

- Bibliotecas compartidas durante el tiempo de enlazado, pero que necesitan ser localizada en tiempo de ejecución (e.g. plugin)
- Bibliotecas compartidas localizadas en tiempo de ejecución usando la librería de enlazado `ld.so`
- Cuando hay cambios en las librerías compartidas, se debe de ejecutar `ldconfig` para actualizar la vista de `ld.so`
- Durante la carga, los símbolos en las bibliotecas dinámicas se les asigna memoria y se cargan desde el fichero de la biblioteca dinámica

Cargando bibliotecas compartidas bajo demanda

- En Linux, puedes cargar símbolos desde las bibliotecas compartidas usando `dlfcn.h`
- Abrir una librería compartida para ser cargada:

```
void * dlopen(const char *file, int mode);
```

Modes:

```
RTLD_LAZY(lazy loading of library),
RTLD_NOW(load now), RTLD_LOCAL,
RTLD_GLOBAL
```

Cargando bibliotecas compartidas bajo demanda

- Conseguir la dirección de un símbolo cargado desde la librería

```
void* dlsym(void * handle,
            const char* symbol_name);
```

`handle` desde una llamada a `dlopen`;
retorna una dirección a una variable o a una función identificada por `symbol_name`

- Función de cierre de una biblioteca compartida

```
int dlclose(void * handle);
```

- Estas funciones no son parte de C-standard-library; necesitas enlazar contra `libdl: con -ldl` (opción de compilación del gcc).

Creando bibliotecas

- Contexto: Bibliotecas que contienen código C como otro programa
- Librerías estáticas o compartidas compiladas desde ficheros objeto usando `gcc`
- Pasos para compilar una librería estática
  - Compilar sin enlazar:

```
athena% gcc -g -Wall -c infile.c -o outfile.o
```

  - Agrupar los ficheros en la biblioteca:

```
athena% ar -rcs libname.a outfile1.o outfile2.o
```

Creando bibliotecas dinámicas

- Paso 1: Compilar sin enlazar usando `gcc`

```
athena% gcc -g -Wall -fPIC -c infile.c -o
outfile.o
```

  Opción `-fPIC`: crear código independiente de la posición, dado que todo el código se reposicionará.

- Enlazar los ficheros usando `ld` para crear el fichero compartido (.so)

```
athena% ld -shared -soname libname.so -o
libname.so.version -lc outfile1.o outfile2.o
```

- Añadir el directorio a la variable `LD_LIBRARY_PATH` , para que `ld.so` puede encontrar el fichero.
- Configurar `ld.so` para nuevas bibliotecas

```
athena% ldconfig -v
```

uc3m | Universidad Carlos III de Madrid

Gracias por su atención