Lenguajes de Programación I

Nombres y Asociación

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright © 2007-2016

¿Qué son Nombres y Asociaciones?

. . . en el contexto de lenguajes de programación

Nombre
(*Name*)

- Cadena alfanumérica mnemónica que denota un elemento de interés.
- Logran la *Abstracción* – objetivo de los lenguajes de programación.

Asociación
(*Binding*)

- Relación entre un nombre y el elemento que denota.
- Creadas y destruidas según las decisiones tomadas por el diseñador.

¿Cuándo se asocia un nombre a "algo"?

Binding Time

Momento de Asociación
(*Binding Time*)

- Instante en que comienza la relación entre el nombre y lo denotado.
- Momento preciso en que se establece e intervalo de duración dependen de la decisión de diseño.
- Asociación temprana – mejorará la eficiencia de ejecución.
- Asociación tardía – mejorará la flexibilidad de uso.

Las asociaciones más tempranas

Cuando el lenguaje sólo es una idea

Al momento de Diseñar el Lenguaje
(*Language Design Time*)

- Palabras reservadas.
- Tipos primitivos y sus constructores.

Las asociaciones más tempranas

Cuando se construye la implantación

Al momento de Implantar el Lenguaje
(*Language Implementation Time*)

- Precisión para representar elementos.
- Disposición de elementos almacenados en memoria.
- Acoplamiento con el subsistema de I/O.

Las asociaciones del usuario

Mientras se escribe el programa

Al momento de Escribir el Programa
(*Program Writing Time*)

- El Programador es libre de escoger nombres para:
  - Tipos y sus constructores.
  - Variables.
  - Funciones o Procedimientos.
  - Módulos.
- Sin entrar en conflicto con lo decidido por Diseñador e Implantador.

Las asociaciones de transformación

Mientras se produce el programa ejecutable

Al momento de Compilar el Programa
(*Compile Time*)

- Asociar construcciones de Alto Nivel con equivalentes de Bajo Nivel.
  - ¿Cuáles instrucciones de máquina para cada fragmento de programa?
  - ¿Cuáles ubicaciones de máquina para datos estáticos?
- Generación de nombres internos.
  - Para puntos de salto en el lenguaje de máquina.
  - Para identificar datos constantes.
  - Para agregar información de *debugging* o *profiling*.

Las asociaciones de construcción

Mientras se completa el programa ejecutable

Al momento de Enlazar el Programa
(*Link Time*)

- Resolver referencias entre módulos y librerías.
- Decidir la distribución final de memoria.

Las asociaciones de arranque

Ready... set...

Al momento de Cargar el Programa
(*Load Time*)

- Asignación de memoria virtual.
- Asignación de recursos adicionales.

Mientras se ejecuta el programa

...go!

Al momento de Ejecutar el Programa
(*Run Time*)

- Asociar valores a variables.
- Aparecen, cambian, desaparecen y reaparecen – dependen del flujo de ejecución y anidamiento.
- Ejecutar funciones y procedimientos.

Estático vs. Dinámico

Más temprano que tarde, *ma non troppo*

Asociación Temprana

- También llamada **Asociación Estática** (*Static Binding*).
- Apunta hacia la *eficiencia*.
- Típica de los lenguajes compilados.

Asociación Tardía

- También llamada **Asociación Dinámica** (*Dynamic Binding*).
- Apunta hacia la *flexibilidad*.
- Típica de los lenguajes interpretados.

El panorama de las cosas

Momentos de
Asociación Temprana

- Diseño.
- Implantación.
- Escritura.
- Compilación.
- Enlace.

Momentos de
Asociación Tardía

- Enlace.
- Carga.
- Ejecución.

La frontera entre lo estático y dinámico
es bastante difusa en algunos lenguajes.

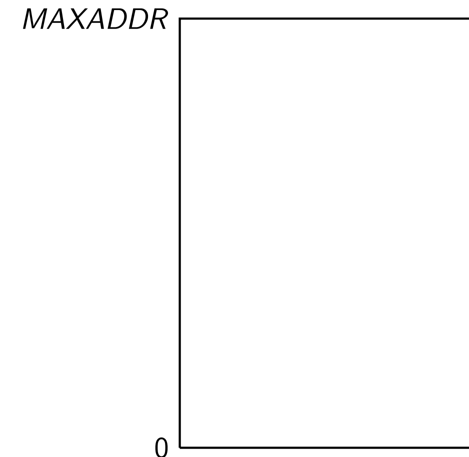
Diferentes tiempos, diferentes espacios

- Creación de Objetos –
  variables, tipos, procedimientos. . . "cosas" del lenguaje.
- Creación de Asociaciones entre Nombres y Objetos.
- Uso de los Objetos a través de esas Asociaciones.
- Activación y reactivación de Asociaciones suspendidas temporalmente.
- Destrucción de Asociaciones entre Nombres y Objetos.
- Destrucción de Objetos.

El tiempo que pasa entre la creación y destrucción de una Asociación se denomina **Tiempo de Vida** (*lifetime*).

¿Dónde se almacenan los objetos?

Memoria virtual asignada al programa en ejecución



- Sistema de Operación asigna espacio virtual al **proceso**.
- `MAXADDR` según CPU – 4Gb en 32 bits puro, 64Gb en 32 bits PAE, 16Eb en 64 bits puro.
- **Programa** sólo puede usar lo **asociado**.

Mecanismos de Asignación de Almacenamiento

El Tiempo de Vida establece el Espacio

- **Objetos Estáticos**
  - Ubicación fija absoluta.
  - Establecida a tiempo de compilación o carga.
  - Mantenida durante la ejecución del programa.
- **Objetos en Pila**
  - Almacenados y removidos en la *Pila de Ejecución*.
  - Modelan la invocación de subrutinas o bloques anidados.
- **Objetos en *Heap***
  - Almacenados y removidos arbitrariamente.
  - Implícita o explícitamente en conexión con variables dinámicas.

Guía para la existencia

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

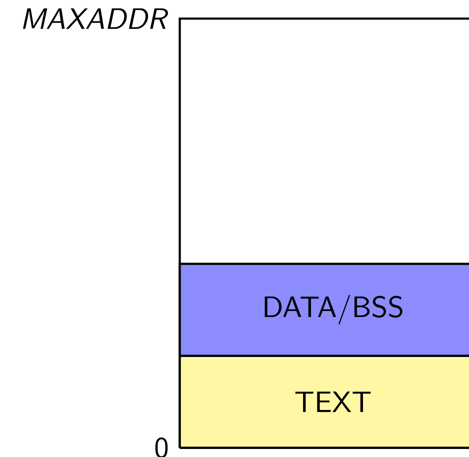
Almacenamiento Estático

¿Cuáles son los objetos estáticos?

- Código ejecutable – en general, no puede modificarse.
- Variables globales.
- Constantes – explícitas e implícitas.
- Variables locales que preservan su valor entre invocaciones.
- Tablas de Símbolos para *debugging* y *profiling*.
- En lenguajes sin recursión, el espacio para subrutinas.

Almacenamiento Estático

El código al principio, las globales después



- Tamaño fijo – calculado al compilar.
- `TEXT` sólo puede ejecutarse.
- `DATA/BSS` puede tener datos sin inicializar.

Guía para la existencia – Objetos Estáticos

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

Almacenamiento en Pila

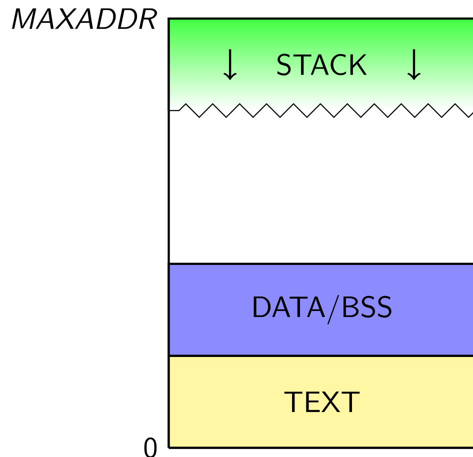
¿Cuáles objetos aparecen y desaparecen según el flujo?

- En lenguajes con recursión, cada subrutina cuenta con su espacio de trabajo denominado **Registro de Activación** para almacenar
  - Argumentos.
  - Variables locales.
  - Direcciones y valores de retorno.
- El programa *ejecutable* debe mantener la pila:
  - El **llamador** sigue la *secuencia de invocación* antes de saltar.
  - El **llamado** debe comenzar con un *prólogo* y finalizar con un *epílogo* para el mantenimiento.
- Una pila modela la aparición de llamadas a rutinas activas – no sabemos cuántas, ni en cuál orden, hasta correr el programa. . .

Los detalles (¡muchos!) los estudiaremos más adelante, ahora solamente interesa la idea general.

Almacenamiento en Pila

Pila de Ejecución



- Pila crece hacia las direcciones bajas.
- Tamaño limitado por el sistema de operación.
- Usar instrucciones especiales del CPU (`PUSH`, `POP`) siempre que sea posible.

Guía para la existencia – Objetos en Pila

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

Registro de Activación – *Stack Frame*

Separación de preocupaciones

El código generado para la rutina. . .

- Establece un apuntador base a la pila (*frame* o *base pointer*).
- Accede a los objetos en la pila usando desplazamientos relativos (*offset*) en relación al *frame pointer*.
- No importa dónde quede el registro de activación en la pila, pues los desplazamientos son relativos al "centro" de cada invocación.

Registro de Activación – *Stack Frame*

Separación de preocupaciones

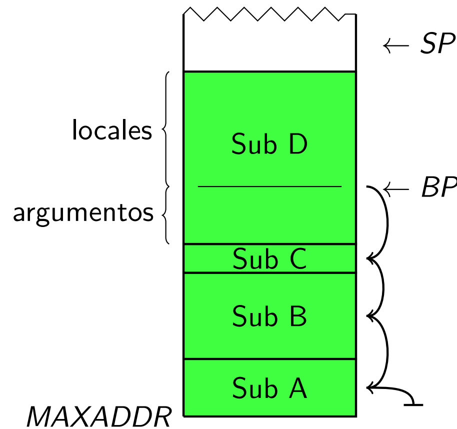
El código generado para la llamada. . .

- Comienza por empilar parámetros y saltar a la rutina – esta es la secuencia de llamada en el llamador.
- El prólogo dentro del procedimiento llamado
  - Conserva el *frame pointer* previo.
  - Establece valor concreto para *frame pointer*.
  - Mueve tope de la pila para reservar espacio local.
- El procedimiento llamado ejecuta.
- El epílogo dentro del procedimiento llamado
  - Regresa tope de pila al *frame pointer*.
  - Restablece el *frame pointer* anterior.
  - Retorna al llamador.

Todo el código es generado *estáticamente*,
para controlar llamadas que ocurrirán *dinámicamente*.

Registro de Activación

...el rastro de la ejecución



- SP (*Stack Pointer*) – tope de la pila.
- BP (*Base Pointer*) – base de *stack frame*.
- Locales – tienen *offset* **negativo**.
- Argumentos – tienen *offset* **positivo**.
- **Cadena dinámica** – ¿quién llamó a quién?

Guía para la existencia – Las llamadas

Llamando a baz(2)

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

- *BP* → main.
- main empila el 2
- main llama a baz – CPU guarda la dirección en pila.
- baz empila *BP*.
- baz copia *SP* a *BP*, apunta al “centro”.
- baz mueve *SP* haciendo espacio para bar.

Guía para la existencia – Las llamadas

En el cuerpo de la subrutina baz(2)

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

- *BP* más *offset* **positivo** para qux.
- *BP* más *offset* **negativo** para bar.
- El valor de retorno podria tener *offset* **positivo** o estar en un registro.
- Proceso similar para llamar a malloc.

Guía para la existencia – Las llamadas

Regresando de la subrutina baz(2)

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main() {
  printf("The answer is %d\n", baz(2));
}
```

- baz mueve *SP* hasta *BP* liberando el espacio de bar.
- baz desempila el *BP* anterior.
- baz retorna a main.
- main recupera el valor de retorno del tope de la pila o de un registro.

Almacenamiento en *Heap*

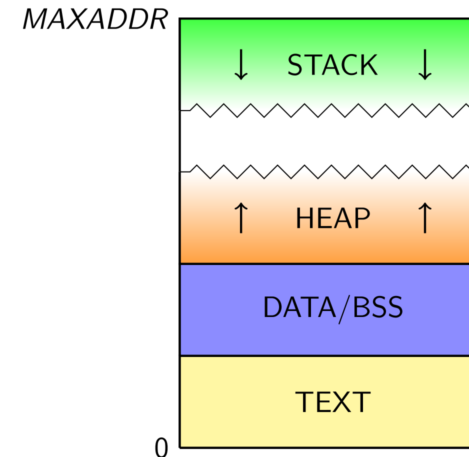
¿Cuáles objetos aparecen y desaparecen arbitrariamente?

- **Heap** – región de almacenamiento en la cual pueden reservarse y liberarse bloques arbitrariamente.
- Primitivas o funciones del lenguaje para reservar y liberar bloques.
- El *ejecutable* debe contar con código *adicional* que implante la estrategia de manejo de memoria:
  - Lista de Bloques Libres (*Free List*).
  - Política para reservar bloques:
    - First Fit.
    - Best Fit.

Escribir un manejador de memoria es un ejercicio complicado que se estudia en Lenguajes de Programación III

Almacenamiento en *Heap*

Completa



- Crece hacia las direcciones altas.
- Crece en bloques grandes – se amplía con `sbrk` en Unix.
- Nunca se reduce – necesario reciclar.

Guía para la existencia – Almacenamiento en Heap

. . . no lo puedo colorear ☹

```
#include <stdio.h>
#include <stdlib.h>

int foo = 21;

int baz(int qux) {
  int *bar;
  bar = (int *) malloc(sizeof(int));
  *bar = foo * qux;
  return *bar;
}

main () {
  printf("The answer is %d\n", baz(2));
}
```

- `malloc` reserva espacio en el *heap* para un `int`.
- `bar` apunta a ese espacio.
- El *heap* tendrá **bastante** más espacio reservado, disponible para otros `malloc`.

Fragmentación del *Heap*

Memoria libre que no puedes usar

Fragmentación Interna

- El objeto contenido no ocupa todo el bloque asignado.
- El espacio sobrante no puede utilizarse.
- Aparece cuando el tamaño de los bloques es fijo o no hay disponible ningún bloque que ajuste mejor.

Fragmentación del *Heap*

Memoria libre que no puedes usar

Fragmentación Externa

- El objeto es más grande que cualquiera de los bloques disponibles.
- Hay muchos bloques pequeños que no son continuos, o bien son continuos pero no se consolidaron.

¿Cuánto cuesta almacenar en el Heap?

Eficiencia en espacio y tiempo

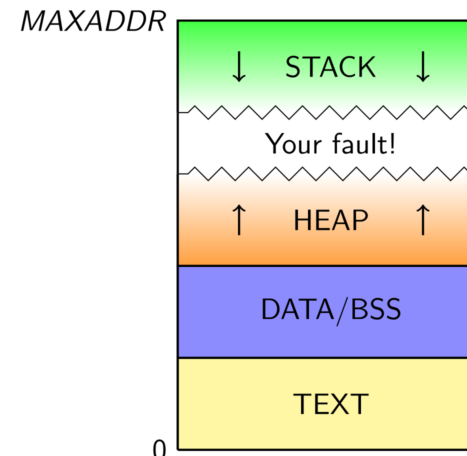
- Recorrer la Lista Libre es proporcional a su longitud.
- Varias Listas Libres cada una con bloques de tamaño diferente.
  - Sistema de “panitas” (*Buddy System*).
  - Sistema Fibonacci.
  - Combinación de contigüos en ambos casos.
- Compactar el *heap* a intervalos regulares puede reducir la fragmentación externa.

Liberación de Espacio en el *Heap*

- Explícita.
  - Simple de implantar con alguna instrucción primitiva en el lenguaje.
  - Es el método más eficiente.
  - Requiere que el programador identifique **correctamente** el momento en que el objeto debe destruirse...
  - ...pero suelen equivocarse y ocurren errores por *referencias colgantes* o *pérdida de memoria*.
- Implícita.
  - Conocida como **recolección de basura (*garbage collection*)**.
  - Complejo de implantar para minimizar el impacto sobre el tiempo de ejecución y los requerimientos de memoria disponible.
  - El programador no necesita hacer nada.
  - Prácticamente obligatorio en lenguajes interpretados y muy frecuente en lenguajes con Máquina Virtual.

Disposición de almacenamiento

Fallas accediendo a memoria



Segmentation fault

El programa accede a una dirección dentro de su **espacio**, pero sin **asociación**.

Bibliografía

- [*Scott*]
  - Secciones 3.1 y 3.2.
  - Ejercicios y Exploraciones
- Página de Wikipedia sobre Memory Management
- Página de Wikipedia sobre Segmentation Fault
- Compile el programa *sólo* hasta lenguaje ensamblable y fíjese en la asignación de espacio hecha por el compilador, así como en la secuencia de llamada.