Estructuras de Datos Dinámicas en C (Pt. 2)

Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur

Copyright

- Copyright © **2011-2024** A. G. Stankevicius
- Se asegura la libertad para copiar, distribuir y modificar este documento de acuerdo a los términos de la **GNU** Free Documentation License, Versión 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera
- Una copia de esta licencia está siempre disponible en la página **http://www.gnu.org/copyleft/fdl.html**

- La versión transparente de este documento puede ser obtenida de la siguiente dirección:

http://cs.uns.edu.ar/~ags/teaching

Contenidos

- Concepto de puntero
- Pasaje por valor y por referencia
- Tipos de datos estructurados
- Gestión de la memoria dinámica
- Estructuras de datos dinámicas
- Concepto de posición directa e indirecta
- Fases de la compilación
- Parámetros en la línea de comandos

Memoria dinámica

- La declaración de toda variable reserva un espacio de tamaño previamente conocido en el registro de activación en curso
  - El espacio reservado para las variables de tipos de datos elementales coincide con lo retornado por la función `sizeof()` al ser aplicada a ese tipo
  - El espacio reservado para las variables de tipos de datos estructurados coincide con la suma del espacio reservado para cada uno de sus componentes
- Por otra parte, también es posible gestionar la reserva y liberación de memoria dinámica

Heap vs. Stack

A ayudame a entender la diferencia entre las regiones de memoria denominadas heap y stack

Por supuesto, te ayudaré a entender las diferencias entre el heap y el stack. Estas son dos áreas de memoria utilizadas por los programas para almacenar datos, pero tienen características y usos muy diferentes.

Stack (Pila):

1. Estructura: Es una estructura de datos LIFO (Last In, First Out).
2. Asignación: La memoria se asigna y libera automáticamente.
3. Tamaño: Tiene un tamaño fijo y limitado (varía según el sistema operativo y la configuración).
4. Velocidad: Las operaciones en el stack son muy rápidas.
5. Uso: Se usa para:
   - Variables locales
   - Parámetros de funciones
   - Dirección de retorno de funciones
6. Control: Gestionado por el compilador.
7. Tiempo de vida: Las variables existen solo mientras la función que las creó está en ejecución.

Heap (Montículo):

1. Estructura: Es un área de memoria de propósito general.
2. Asignación: La memoria se asigna y libera manualmente por el programador.
3. Tamaño: Mucho más grande que el stack, limitado solo por la memoria física del sistema.
4. Velocidad: Las operaciones son más lentas que en el stack.
5. Uso: Se usa para:
   - Objetos y estructuras de datos de tamaño variable
   - Datos que necesitan existir más allá del ámbito de una función
6. Control: Gestionado por el programador (en lenguajes como C) o por el recolector de basura (en lenguajes como Java).
7. Tiempo de vida: Las variables existen hasta que se liberen explícitamente o hasta que el programa termine.

Ejemplo en C:

c Copy

```c
#include <stdlib.h>

void funcion() {
    int x = 5;               // Variable en el stack
    int *y = malloc(sizeof(int));  // Puntero en el stack, pero apunta a memo
    *y = 10;

    // Uso de x e y

    free(y);  // Liberar la memoria del heap
} // x se libera automáticamente, y (el puntero) también, pero no la memoria
```

En este ejemplo, `x` se almacena en el stack y se libera automáticamente cuando la función termina. El puntero `y` está en el stack, pero la memoria a la que apunta está en el heap. Esta memoria del heap debe liberarse manualmente con `free()`.

Memoria dinámica

- **C** cuenta con las siguientes funciones de librería para la gestionar la asignación de memoria dinámica:
  - **`void* malloc(size_t)`**: esta función intenta reservar la cantidad indicada de memoria dinámica
  - **`free(void*)`**: esta función libera la porción de memoria dinámica que comienza donde se indica
  - **`void* realloc(void*, size_t)`**: esta función reajusta al tamaño solicitado el espacio de memoria dinámica que comienza donde se indica

Memoria dinámica

- Al terminar de hacer uso de grandes estructuras dinámicas enlazadas se debe tener cuidado al liberar el espacio que ocupaban
- La invocación a **`free()`** sólo libera el espacio reservado por el **`malloc()`** que generó el puntero pasado como argumento
  - Si una estructura se armó mediante múltiples invocaciones a **`malloc()`**, su espacio deberá ser retornado mediante múltiples invocaciones a **`free()`**

¡IMPORTANTE!

Memoria dinámica

```
int *i;
char *c;
struct persona *p;
i = (int *) malloc(sizeof(int));
c = (char *) malloc(sizeof(char));
p = (struct persona *)
      malloc(sizeof(struct persona));
free(i);
c = (char *) realloc(c, sizeof(char) * 9);
```

Java vs. C

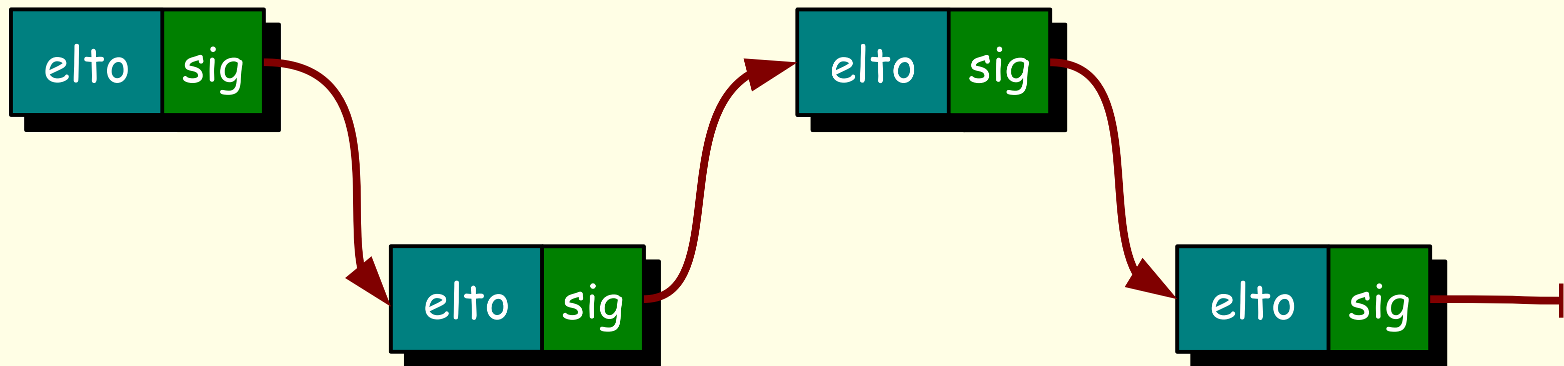
- Hasta ahora los lenguajes de programación **Java** y **C** han compartido bastantes similitudes, sobre todo a nivel de sintaxis
- En este punto aparece una diferencia que estamos obligados a tener siempre en cuenta:
  - **Java** se hace cargo por completo de la recuperación de la memoria dinámica asignada a un objeto cuyo uso finalizó
  - **C**, en contraste, deja esa tarea por completo en manos del programador

Listas

- Recordemos las principales características de la estructura de datos dinámica “lista”:
  - Es una estructura de datos dinámica
  - El espacio ocupado por sus elementos es asignado a medida que se necesite
  - Cada elemento apunta al siguiente, determinando una relación lineal
  - Puede ser simple o doblemente enlazada
  - Permite implementar pilas y colas

Listas

```
struct celda {
  tipo elemento;
  struct celda *sig;
};
```



Listas

- Esta estructura de datos cierta particularidades:
  - Los elementos usualmente son todos del mismo tipo
  - Por lo general, cada celda es creada por separado invocando repetidamente a la función **`malloc()`**
  - Cada celda apunta a la siguiente
  - La última celda apunta a **`NULL`**
  - La lista completa se representa mediante un puntero al primer elemento

Posición directa vs. indirecta

- La posición de un elemento denota su ubicación dentro de la lista
- Existen principalmente dos variantes para el concepto de posición:
  - Posición directa: la posición se denota mediante un puntero a la celda conteniendo el elemento deseado
  - Posición indirecta: la posición se denota mediante un puntero a una celda conteniendo un puntero a la celda conteniendo el elemento deseado

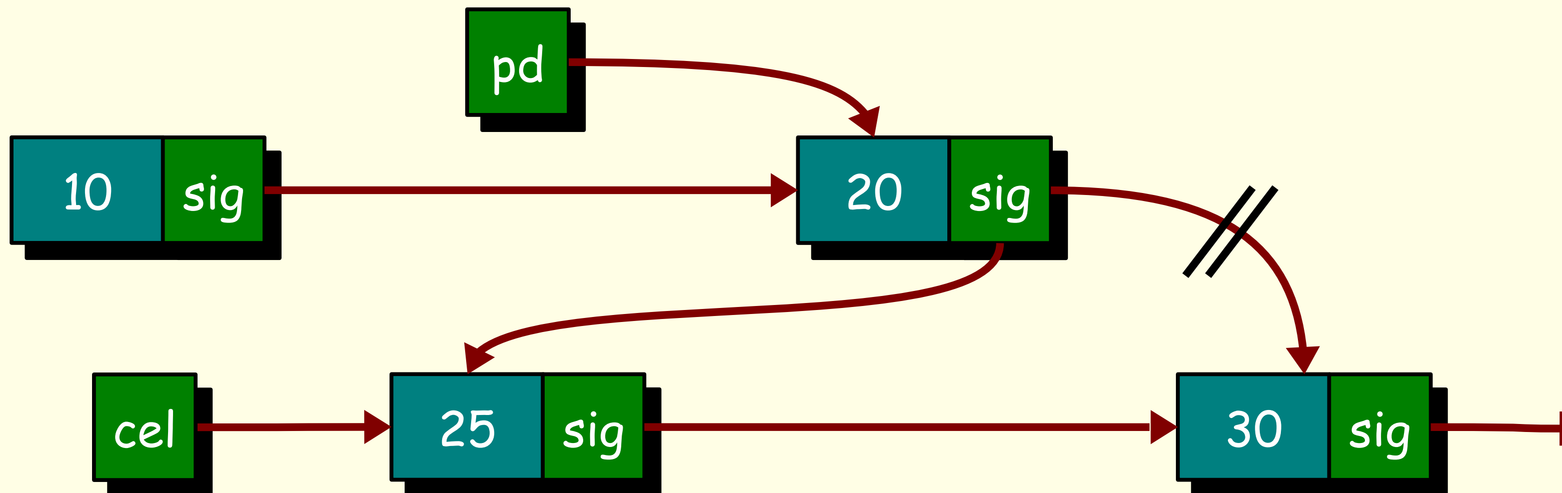
Posición directa vs. indirecta

- Considerando el elemento **`elto`**, el puntero **`pd`** representa su posición directa y **`pi`** su posición indirecta:

Inserción en listas

- Veamos cómo agregar una nueva celda ***cel** a continuación del elemento en la posición ***pd** (usando el concepto de posición directa):

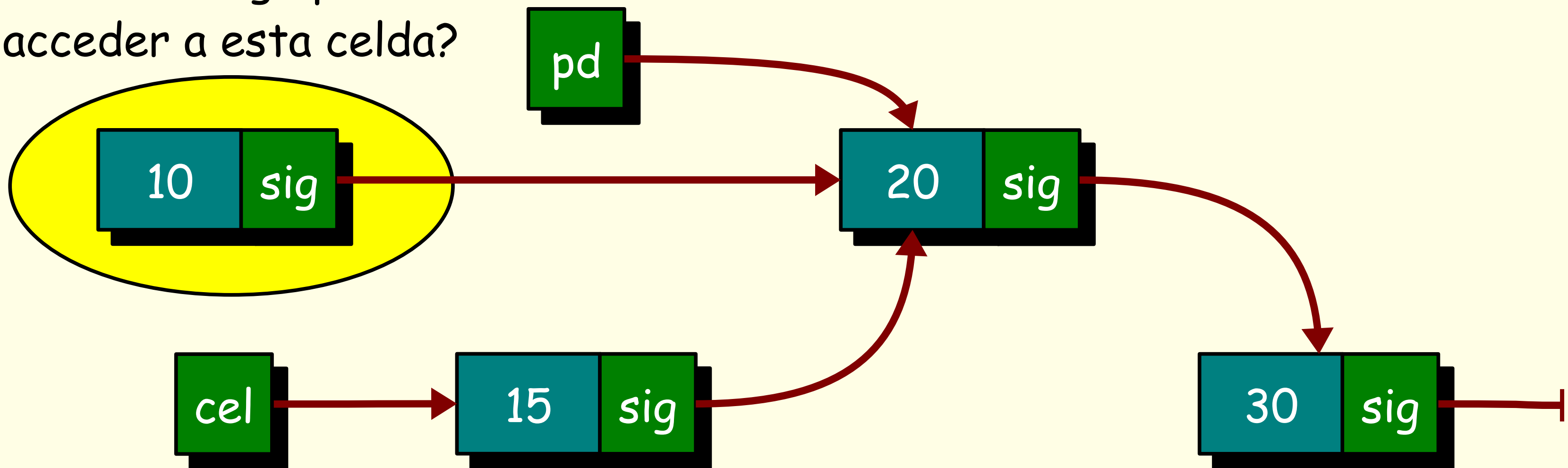
```
struct celda *cel, *pd;
```



Inserción en listas

- Veamos cómo agregar una nueva celda ***cel** antes que el elemento en la posición ***pd** (usando el concepto de posición directa):

```
struct celda *cel, *pd;
```



Inserción en listas

- Para agregar un elemento a una lista antes de otro hace falta poder acceder al elemento anterior en la lista
- Esto se puede resolver de varias formas, siempre pagando algún costo:
  - Recorriendo la lista desde el principio
  - Haciendo uso de posición indirecta en vez de directa
  - Implementando una lista doblemente enlazada
- ¿Qué costo pago en cada caso?

Fases de la compilación

- Compilación (traducción):
  - Preprocesado
  - Generación de código assembler
  - Generación de código objeto
- Enlazado o vinculación:
  - Enlazado con otros archivos objeto
  - Enlazado con librerías estándar
  - Generación del ejecutable

Fases de la compilación

.c

preprocesado

.c

generación de código assembler

.s

compilación

.o

ensamblado

.exe

enlazado

.o

.a

librerías estándares y del usuario

Directrices al preprocesador

- Las directrices al preprocesador son interpretadas antes de la compilación:
  - **#define**: define una nueva constante o macro del preprocesador
  - **#include**: indica que se debe incluir el contenido de otro archivo en el punto indicado
  - **#ifdef**, **#ifndef**: señala el comienzo de un bloque de preprocesamiento condicionado
  - **#endif**: marca el fin de un bloque de preprocesamiento condicionado

Constantes y macros

- El preprocesador permite asociar valores constantes a ciertos identificadores que serán expandidos antes de la compilación propiamente dicha:

```
#define variable valor-cte
```

- Análogamente, las macros son funciones que han de ser expandidas antes de la compilación:

```
#define macro(args, ...) def-función
```

Constantes y macros

```
#define PI        3.14
#define CANT      5
#define AREA(rad) PI * rad * rad
#define MAX(a, b) (a > b ? a : b)
int main() {
  int i; float vector[CANT];
  for(i = 0; i < CANT; i++)
    vector[i] = MAX(i * 5.2, AREA(i));
}
```

Archivos de encabezamiento

- Los archivos de encabezamientos incluidos mediante la directiva **`#include`** suelen tener la extensión "**`.h`**"
- Las funciones declaradas en los archivos de encabezamiento no incluyen sus implementaciones
- Las variables que allí aparezcan están declaradas como **`extern`**, ya que su definición ha de figurar en otro archivo fuente (el "**`.c`**" asociado a ese "**`.h`**")

Preprocesado condicional

- Para incluir código cuya compilación dependa de ciertas circunstancias, se puede hacer uso de las siguientes directivas:

```
#ifdef variable
  <bloque de sentencias>
#endif
#ifndef variable
  <bloque de sentencias>
#endif
```

Preprocesado condicional

```
#define DEBUG
int main() {
  int i, acc;
  for (i = 0; i < 10; i++)
    acc = i * i - 1;
  #ifdef DEBUG
    printf(“fin del bucle: %d”, acc);
  #endif
}
```

Conversión de tipos

- **C** cuenta con diversas funciones de librería para asistir al programador en la conversión entre tipos de datos:
  - **`int atoi(char*)`**: traduce de strings a enteros
  - **`long atol(char*)`**: traduce de strings a enteros largos
  - **`double atof(char*)`**: traduce de strings a reales
  - **`char* itoa(char*, int)`**: traduce de enteros a strings

Argumentos en línea

- En **C** torna simple acceder a los argumentos suministrados en la línea de comandos:

```
void main(int argc, char *argv[]) {
  int i;
  printf("%d argumentos", argc);
  for (i = 1; i < argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }
}
```

¿Servirá de algo?

sirve de algo en pleno siglo XXI saber procesar los argumentos recibidos por un program en la línea de comandos?

Show drafts

¡Absolutamente! Saber procesar argumentos de línea de comandos en pleno siglo XXI sigue siendo una habilidad muy valiosa para programadores y usuarios avanzados por varias razones:

1. Personalización y Flexibilidad:

- **Adaptación a diferentes escenarios:** Permite ejecutar un programa con diferentes configuraciones sin necesidad de modificar el código fuente. Por ejemplo, puedes especificar la ruta de un archivo de entrada, el nivel de detalle de un reporte, o activar o desactivar ciertas funcionalidades.
- **Automatización de tareas:** Facilita la creación de scripts y automatizaciones que interactúan con otros programas o sistemas.
- **Creación de herramientas versátiles:** Permite desarrollar herramientas que puedan ser utilizadas por otros usuarios con diferentes necesidades.

2. Comprensión Profunda de los Programas:

- **Mayor control:** Al entender cómo se pasan argumentos a un programa, tienes un control más preciso sobre su comportamiento.
- **Depuración y resolución de problemas:** Facilita la identificación y solución de errores al poder ejecutar el programa con diferentes opciones y ver cómo responde.

3. Integración con otras Herramientas:

- **Pipelines:** Los argumentos de línea de comandos son fundamentales en la creación de pipelines de procesamiento de datos, donde la salida de un programa se convierte en la entrada del siguiente.
- **Scripts de construcción:** Herramientas como Make o Gradle utilizan argumentos de línea de comandos para personalizar el proceso de compilación y empaquetado de software.

4. Habilidad Esencial para Desarrolladores:

- **Desarrollo de software:** Es una habilidad básica que todo programador debería dominar, ya que muchos lenguajes de programación y herramientas lo soportan.
- **Entornos de desarrollo:** La mayoría de los IDEs y entornos de línea de comandos ofrecen funcionalidades para trabajar con argumentos de manera eficiente.

En resumen, aunque las interfaces gráficas son cada vez más comunes, la línea de comandos sigue siendo una herramienta poderosa y flexible. Saber procesar argumentos de línea de comandos te permite:

- **Aumentar tu productividad:** Automatizando tareas y creando herramientas personalizadas.
- **Profundizar tus conocimientos:** Entendiendo mejor cómo funcionan los programas y cómo interactúan entre sí.
- **Ser un programador más completo:** Dominando una habilidad esencial en el desarrollo de software.

Argumentos en línea

- Cabe acotar que la convención es que el primer argumento es el nombre del programa que se está ejecutando

```
printf("Invocado como: %s",argv[0]);
```

- Es decir, los argumentos en las las restantes posiciones (de **1** a **argc-1**), son los argumentos en la línea de comando efectivos
  - Nótese que los argumentos se reciben como cadenas de caracteres, incluso al tratarse de números

¿Preguntas?