Manuel Arturo Chinchilla Sánchez - 2013009344

Resumen #5

Pointers and Memory

Pointers: Los punteros resuelven dos problemas de software comunes. Permiten a diferentes secciones de código compartir información fácilmente. Permiten crear estructuras complejas de datos enlazados. Los punteros almacenan referencias de memoria a otros valores.

Pointer Dereference: La operación de "desreferencia" consiste en seguir la referencia de un puntero para obtener su valor. La única restricción es que el puntero debe de tener una referencia asignada para la eliminación del acceso a la misma. La mayor parte de los errores en el código de los punteros se debe a esta restricción, que para que funcione la operación de desreferencia primero debe existir la misma.

NULL Pointer: La constante NULL es un valor especial para un puntero, que encierra la idea de "un puntero apuntando a nada". Normalmente usamos este en la inicialización de los punteros, para evitar errores en tiempo de ejecución.

Pointer Assignment: La operación de asignación (=) entre dos punteros, hace que los dos apunten a la misma cosa.

Make A Drawing: Hacer dibujos de la memoria pueden ser herramienta muy útil para pensar sobre el código de un puntero. Nos puede ayudar a comprender como se va a usar la memoria en tiempo de ejecución.

Sharing: Dos punteros que apuntan a un mismo valor, se dice que están "compartiendo".

Shallow and Deep Copying: En términos generales, al realizar una "Shallow copy" se está aplicando el "sharing", un segundo puntero hace referencia a la misma dirección de memoria que el primero, sin embargo con la "Deep copy" se crea una copia independiente del valor referenciado en una nueva posición de la memoria y el puntero hará referencia a esa nueva ubicación.

Bad Pointers: Cuando se crea un puntero y no es inicializado, se dice que está mal, ya que hace referencia a una posición de memoria que puede estar ocupada por datos llamados "basura".

Local Memory: Las variables representan espacio de almacenamiento en la memoria de la computadora.

Usar parámetros locales tiene sus ventajas:

- Son convenientes, satisfacen las necesidades prácticas, algunas funciones necesitan la memoria temporalmente, y las variables locales facilitan esto.
- Eficientes, Son relativamente eficientes asignando y desasignando memoria.

Aunque también tenemos desventajas:

- Tiempo de vida corto, a veces nuestros programas necesitan que la memoria continúe asignada después de que la función que asigno la memoria a la variable e cerró.
- Comunicación restringida, dado que los parámetros locales son copias de los parámetros de llamada, no proporcionan un medio de comunicación.

Heap Memory: también conocida como memoria “dinámica”, y es la alternativa al “Stack”, a diferencia de la memoria local que es muy automática, en el Heap solicita explícitamente la asignación de un bloque de memoria de un tamaño determinado, y esta memoria sigue siendo asignada hasta que el programador pida explícitamente en el código que sea liberada. Como nada sucede automáticamente, el programador tiene un mayor control de la memoria, pero como dicen por ahí, “cuanto más poder tenemos, mayor es nuestra responsabilidad”. Las ventajas del heap son:

- Tiempo de vida, el programador controla exactamente cuando la memoria es asignada y liberada, además es posible crear estructuras de datos.
- Tamaño, el tamaño de la memoria asignada puede ser controlada con más detalles.

Desventajas:

- Más trabajo, la asignación y liberación de memoria en el heap necesita estar explícitamente en el código.
- Más bugs, como tenemos más control sobre la memoria, esto nos puede llevar a tener más errores en nuestro código.