Punteros en C/C++: problemas a evitar

Introducción

Un puntero es una variable que contiene la dirección de memoria de otra variable.
Los punteros permiten código más compacto y eficiente; utilizándolos en forma ordenada dan gran flexibilidad a la programación.

La dirección de memoria de una variable se obtiene con el operador unario `&`.
El operador unario `*` permite la desreferencia de un variable puntero; es decir, permite el acceso a lo apuntado por un puntero.

Dado el ejemplo

```
int x = 1, y = 2;
int *ip;   /* ip es un puntero a int */

ip = &x;       /* ip apunta a x */
y = *ip;       /* a y se le asigna lo apuntado por ip */
*ip = *ip + 3; /* incrementa lo apuntado por ip, x    */
ip = NULL;     /* ip apunta a direcc. especial (nada) */
```

La sintaxis de la declaración de un puntero imita a las expresiones en que la variable puede utilizarse; cada puntero apunta a un tipo específico de datos (con la excepción del puntero genérico `void`).
La dirección especial `NULL` (o cero) indica que un puntero apunta a "nada" y es usada como centinela para establecer el fin de estructuras autoreferenciadas. Además, esta es retornada por la función de asignación de memoria, `malloc`, en el caso de no haber suficiente memoria. El operador `new`, en cambio, aborta el programa cuando no tiene más memoria que dar.

Problemas a evitar en el uso de punteros

El uso descuidado de punteros lleva a programas incomprensibles y particularmente a problemas difíciles de encontrar y reparar.

Ausencia de inicialización de punteros

Un problema común es el uso de punteros no inicializados. Mientras las variables externas (a funciones) o estáticas son inicializadas en cero por omisión, las variables locales (pertenecientes a funciones) no son inicializadas automáticamente. Por lo tanto punteros definidos dentro de funciones que no sean inicializados pueden contener cualquier valor ("basura").

Es un error desreferenciar un puntero con valor NULL,

```
int *ip = NULL;

cout << *ip << endl;
```

provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa.

También es erróneo usar un puntero no inicializado,

```
int *ip;

cout << *ip << endl;
```

los compiladores en general informan (warning) que se esta usando una variable no inicializada. [*Nota: Para que g++ lo haga debe llamarse así: g++ -O1 -W -Wall -c src.cpp*]
Su ejecución tiene un comportamiento aleatorio, dependiendo del valor que casualmente exista en el área de memoria asignada a la variable puntero. Puede llegar a funcionar incorrectamente si la información que hay en memoria casualmente es un entero válido; pero probablemente provocará la detención del programa. En estructuras de datos dinámicas la ejecución del programa en estas condiciones es errática, luego de ejecutarse incorrectamente por cierto tiempo se detiene ("cuelga") el programa.

Potencialmente mas grave es modificar un área de memoria que no fue definida explícitamente,

```
int *ip;

*ip = 10;
```

no solo modifica un espacio de memoria que no le fue asignado; sino que, al no estar inicializado el puntero, se modifica el espacio de memoria al que casualmente direccione el puntero. La declaración `int *ip` solo asigna memoria para la variable puntero, no para los datos apuntados por éste. En **general** (lo cual no quiere decir que ocurra siempre) provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa.

Una forma adecuada de hacerlo es

```
int *ip;

ip = new int; //(1)
*ip = 10;
```

donde la función `new` establece memoria para el entero y la asignación inicializa el puntero. En caso de querer utilizar estrictamente el lenguaje **C** (el operador `new` es propio de **C++**), la sentencia (1) pasaría a ser la siguiente:

```
ip = (int*) malloc (sizeof (int)); //(1)
```

La asignación de punteros se puede realizar adecuadamente al

- establecer su valor inicial en NULL, y luego asignar memoria previo a su desreferenciamiento,
- asignar un puntero a otro que a sido adecuadamente inicializado, o
- usar las funciones `new` o `malloc` para reservar memoria.

Problemas de alias

La flexibilidad de tener más de un puntero para una misma área de memoria tiene ciertos riesgos.

El manejo de memoria dinámica (heap) es restringido a las funciones `new (o malloc)` y `delete (o free)`, en

```
int *ip;
int x = 1;

ip = &x;
delete ip;
```

es erróneo que `delete` desasigne memoria que no fue asignada con `new`.
En general provoca la excepción de violación de acceso de memoria (segmentation fault) y detiene el programa.

El manejo de alias puede dejar punteros colgados (dangling), en

```
int *ip, *jp;

ip = new int;
*ip = 1;
jp = ip;
delete ip;
cout << *jp << endl;
```

`jp` queda apuntando a un área de memoria que conceptualmente no esta disponible, más aun el área de memoria a la cual `jp` quedó apuntando puede ser reasignada por ejecuciones subsiguientes de la función `new`. Notar el error en que se podría incurrir si luego de reasignar el área de memoria en cuestión para otro uso (nueva invocación a `new`), se desasigna el área de memoria a la que apunta `jp` (`delete jp`).

Otro problema potencial es la pérdida de memoria asignada, en el ejemplo

```
int *ip = new int;
int *jp = new int;

*ip = 1;
*jp = 2;
jp = ip;
```

el área de memoria asignada a `jp` se perdió, no puede ser retornada al heap para su posterior reasignación. Errores similares son reasignar memoria a un puntero que ya

tiene memoria asignada o asignarle el valor NULL, sin previamente desasignar la memoria inicial. Esto es un mal manejo de la memoria y puede llevar al agotamiento prematuro de la misma.