Resumen sobre el uso de punteros y variables dinámicas

Operación	Código	Notas
Definición y declaración	`typedef int* pEntero; pEntero p;`	Poniendo un * antes de la variable el typedef es para facilitar el tipo de variables a las que apunta
Inicialización	`p= NULL;` L> es una posición de memoria	Es conveniente igualar a NULL al inicio del programa (a menos que hagamos `p=malloc(sizeof int))`)
Creación de la variable dinámica	`p=malloc(sizeof (int))`	No se puede utilizar la variable `*p` sin haber hecho el `malloc()`.
Destrucción de esta variable	`free(p)` borra la posición de memoria a la que apunta p.	Siempre se debe realizar cuando no necesitemos la variable. Después se debe hacer `p=NULL`
Acceso al contenido de la variable dinámica	`*p= valor;` `vble= *p;`	
Asignación de valores a punteros	`p= NULL;` `p= q;`	Son las dos únicas sentencias de asignación permitidas.
Comparación de punteros	`p==q;` `p!=q;` `p== NULL;` `p != NULL;`	

NOTACIÓN	
`typedef int* pEntero; pEntero p;`	p = puntero *p= contenido de la variable apuntada por p

Errores más comunes en el manejo de punteros

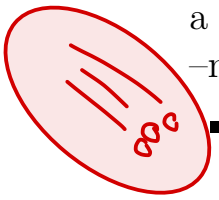
1. Avisos de compilación más frecuentes.

   a) Las variables puntero sólo pueden apuntar a datos de un tipo particular. Por lo tanto, para que los punteros puedan compararse o asignarse entre sí tienen que ser del mismo tipo.

   b) Confundir el puntero (p) con la variable a la que apunta (`*p`).

2. Errores de ejecución más frecuentes.

   a) La variable referenciada por un puntero sólo existe cuando se inicia el apuntador mediante la asignación a una variable ya existente o mediante `malloc()`. Un error muy frecuente es intentar acceder a la variable referenciada cuando no existe. En este caso, estaremos intentando acceder a una dirección de memoria no válida y causará un error de ejecución –normalmente un *Segmentation Fault*.

   - Incorrecto:

```
typedef int* tPos;
tPos p;
...
*p=...
```

   → hay que asignarle memoria antes de acceder a su contenido.

   - Correcto.

```
typedef int* tPos;
tPos p, q;
...
p = malloc(sizeof(int)); *p=... ó
q = malloc(sizeof(int)); p= q; *p=...
```

   Hay que tener en cuenta que los punteros acceden directamente a la memoria del ordenador y, por lo tanto, al acceder a posiciones de memoria no reservadas y escribir en ellas pueden ocurrir errores inesperados como escribir en el propio código del programa.

   b) Para evitar este problema haremos que el puntero contenga el valor `NULL` siempre que no apunte a una variable. Así, podremos reconocer cuándo el puntero apunta o no a una variable con sólo preguntar por su valor.

c) Hay que tener cuidado cuando tenemos varios punteros que apuntan a la misma variable, ya que la modificación de la variable por parte de uno de ellos implicará que también cambiará el contenido para los demás.

d) Además, si uno de ellos libera la variable los demás quedarán desreferenciados (referencias perdidas). Si uno hace free(p), los que apuntan a p, ya no apuntan a p, apuntan a otro sitio

e) La memoria de un ordenador es grande pero no ilimitada, y puede acabarse si constantemente creamos nuevas variables sin liberar el espacio de las que ya no necesitemos. En este sentido, otro tipo de errores que no causan error de ejecución están relacionados con el `free`:

1) Dejar variables a las que ya no apuntan ningún puntero, sin haber hecho un `free`. Supone una pérdida de capacidad de memoria para ese programa.
2) Hacer `free(p)` y no preocuparnos de que `p` apunte a alguna dirección válida o a `NULL`.
3) Hacer un `free(p)` y acceder posteriormente a `p`.

Por cada malloc tiene que haber un free.

Paso de punteros como parámetros

En C, por defecto, el paso de parámetros se hace por valor. C no tiene parámetros por referencia. Se emula mediante el paso de la dirección de una variable, utilizando punteros en los argumentos de la función.

- **Ejemplo: Paso de punteros por valor.** Diseñar una rutina *EsNulo* que dado un puntero de tipo *tPEntero* devuelva 1 o 0 en función de si el contenido del puntero es el valor NULO.

```
int EsNulo(tPEntero p){
   if (p != NULL) then
      return 1
   else
      return 0;
}
```

Aunque un código más correcto sería:

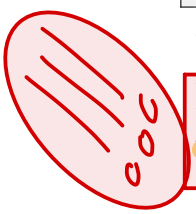
```
int EsNulo(tPEntero p){
    return (p == NULL);
}
```

- **Ejemplo: Paso de punteros por referencia.**

Escribir una rutina *Intercambiar* que, dados dos punteros de tipo *tPEntero*, intercambie sus contenidos sólo en el caso de que no sean punteros nulos. En este caso, pasaremos los punteros por *referencia*.

```
void Intercambiar (tPEntero* p,tPEntero* q){
    tPEntero t;

    if ((!EsNulo(p)) && (!(EsNulo(q))) {
        t = *p;
        *p = *q;
        *q = t;
    }
}
```



Un error común es no pasar el puntero por referencia cuando hace falta. Por ejemplo, supongamos que queremos crear un procedimiento que dado un puntero cree la variable dinámica asociada a él:

```
void CrearVariable (tPEntero* p){
    *p = malloc(sizeof(int));
}
```

¿Cómo hay que pasar el parámetro?¿Qué ocurriría si lo pasamos por valor y luego hacemos *p = 3? Daría un error de ejecución.

Paso de variables dinámicas como parámetros

- **Ejemplo: Paso de variables por valor.** Diseñar una rutina *imprimir* que imprima el contenido de una variable dinámica de tipo *int* apuntada por una variable *int* *. Existen dos opciones:

```
int *p;

int imprimir1(int m){
   printf(" %d \n",m);
}
void main () {
   ...
   imprimir1(*p);
   ...
}
```

O bien

```
int *p;

int imprimir2(int* m){
   printf(" %d \n",*m);
}
void main () {
   ...
   imprimir2(p);
   ...
}
```

- **Ejemplo: Paso de variables por referencia.**

  Escribir una rutina *Intercambiar* que permita rellenar los valores de una variable dinámica.

```
int * p;
int a;

void main (){
    ...
    CrearVariable(&p);
    leerEntero(P);
    leerEntero(&a);
    ...
}

void leerEntero(int* m){
    printf(" Dame un valor de tipo entero: \n");
    scanf(" %d ",m);
}
```