TEMA 8: Gestión dinámica de memoria

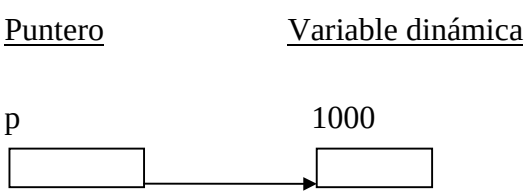
8.1.-Tipo de datos puntero

Hasta ahora, los tipos de datos que hemos visto (a excepción de strings y ficheros) eran estructuras de datos estáticas, es decir, estructuras que no pueden cambiar de tamaño a lo largo de todo el programa. Pero en los lenguajes dinámicos se pueden crear *estructuras de datos dinámicas*, estructuras que si que pueden variar su tamaño durante la ejecución del programa. Además es posible crearlas y destruirlas en cualquier punto del programa, de manera que las reglas de ámbito no las afectan directamente.

Este tipo de variables tienen sin embargo una diferencia muy importante con las variables estáticas y es que en general no tienen nombre que las identifique. Por eso necesitamos utilizar unas variables especiales para referenciar las variables dinámicas: los *punteros*.

Un puntero es una variable que contiene una posición de memoria, y por tanto se dice que apunta a esa posición de memoria.

Ejemplo:



Declaración de un puntero en C++:

```
typedef tipo * NombreTipo;
NombreTipo variable;
```

O directamente:

```
tipo * variable;
```

Ejemplo:

Declaración de un puntero a entero en C++:

```
typedef int * Ptr_int;
Ptr_int p_i;
```

O directamente:

```
int * p_i
```

El tipo de dato al que apunta el puntero (en el ejemplo int) se denomina tipo base del puntero.

Se debe tener en cuenta que cuando declaramos una variable de tipo puntero sólo se reserva memoria para el puntero, **NO** para la variable a la que apunta.

Para obtener el contenido de una variable dinámica a la que apunta el puntero, utilizaremos el operador de indirección `*` seguido del nombre del puntero.

Ejemplo:

```
*p_i = 5;
cout << *p_i;
```

8.2.-Asignación y liberación de memoria

Como ya hemos dicho, la declaración de un puntero no crea ninguna variable dinámica. Para crear una variable dinámica lo que se debe hacer es reservarle memoria. Esto se realiza mediante el operador *new*.

Igualmente, cuando una variable dinámica deja de ser útil, debe ser eliminada. Para ello se utiliza el operador *delete*.

- `new TipoBase`: Crea una variable dinámica del tipo TipoBase. Este operador devuelve como resultado de la operación la dirección de memoria donde empieza la variable. Esta dirección se deberá asignar a un puntero.

- `delete nombrevariable`: Destruye la variable a la que apunta , es decir, libera la memoria que había reservado.

Ejemplo:

```
p_i = new int;
*p_i = 1;
....
delete p_i;
```

Es muy importante que antes de utilizar una variable dinámica, ésta tenga memoria reservada, si no los resultados pueden ser impredecibles.

También es muy importante que toda memoria reservada sea posteriormente liberada, cuando ya ha dejado de ser útil, pues si no se podría llegar a ocupar toda la memoria del ordenador.

En algunos lenguajes modernos (Ada, Java, ...), la liberación de la memoria se realiza de forma automática. En este caso se dice que el lenguaje tiene un *recolector automático de basura* (garbagge collector). C++ no posee este recolector.

8.3.-Operaciones con punteros

8.3.1.-Indirección:

Consiste en obtener el contenido de la variable a la que apunta el puntero. Se realiza, como ya hemos visto, con el operador `*`.

8.3.2.-Asignación:

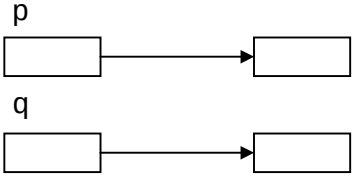
A un puntero sólo se le puede asignar otro puntero, una dirección de memoria o la constante `NULL` (dirección nula). Esta constante se utiliza normalmente para inicializar los punteros.

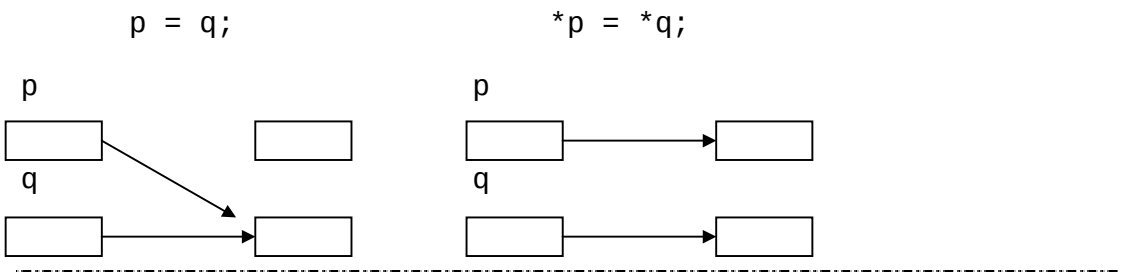
Es importante distinguir entre la asignación de punteros y la asignación entre contenidos de las variables dinámicas.

Ejemplo:

```
int *p, *q;

p = new int;
*p = 2;
q = new int;
*q = 5;
```





8.3.3.-Comparación:

Sobre los punteros también están definidos los operadores de comparación, aunque los únicos que se deben utilizar son los operadores de igualdad (==) o desigualdad (<>) de punteros, ya que el resto de comparaciones no tienen sentido.

8.4. Paso de parámetros por referencia mediante punteros

Mediante el uso de punteros es posible realizar el paso de parámetros por referencia (mecanismo que utilizan lenguajes como el C).

Para ello lo que se hace es pasar un puntero que apunte a la variable que queremos modificar, y dentro del subprograma realizar las modificaciones sobre la variable a través del puntero.

Para obtener la dirección de una variable, se puede utilizar en C++ el operador &.

Ejemplo:

```
typedef int * ptr_int;

int main (void);
void inc (ptr_int)

int main (void)
{
    int i;

    i = 2;
    inc(&i);

    cout << i;   →    3

    return 0;
}

void inc (ptr_int p_i)
{
    *p_i = *p_i + 1;
```

```
}
```

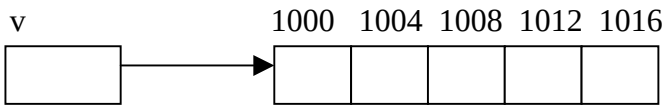
Un puntero puede apuntar a cualquier variable del programa, pero conviene evitar que apunte a variables locales, pues dejan de existir cuando la función acaba.

8.5. Relación entre punteros y vectores

Punteros y vectores en C++ están íntimamente ligados. De hecho, el nombre de un vector es realmente un puntero. Esto quiere decir que cuando se declara un vector como:

```
typedef int Vector[5];
Vector v = {0,10,20,30,40};
```

La variable v realmente es un puntero a entero, es decir, un puntero al tipo base del vector. Este puntero contiene la dirección donde empieza el vector.



Y por tanto, si mostramos el contenido de v obtendremos el elemento en la primera posición del vector.

```
cout << *v;        →    0
```

Esto significa que en C++ la indexación de vectores es realmente una operación sobre punteros y por tanto, *v y v[0] son la misma operación.

Cuando indexamos un puntero, realmente estamos haciendo dos operaciones diferentes. Por ejemplo, en la operación:

```
cout << v[2];      →    20
```

Lo **primero** es averiguar la posición del tercer elemento suponiendo que los elementos son de tipo entero (puesto que el puntero es de tipo entero). Si v vale 1000, el tercer elemento estará en 1008.

Lo **segundo** es, una vez ya tenemos la dirección, tomar el contenido de esa posición. En nuestro ejemplo sería 20.

8.6. Vectores dinámicos

Al igual que se pueden crear variables dinámicas, también se pueden crear vectores dinámicos. Esto se puede hacer gracias a la estrecha relación que existe entre vectores y punteros.

Para crear un vector dinámico:

```
tipo *variable // Definimos un puntero
...
variable = new tipo[tamaño]; // Reservamos memoria para 10 enteros
```

Ejemplo: *Definir un vector dinámico de 10 enteros*

```
int *p // Definimos un puntero a entero
...
p = new int[10];  // Reservamos memoria para 10 enteros
```

De esta forma reservamos memoria para 10 enteros y asignamos la posición del primer entero al puntero p.

Este vector ya se puede manejar como si fuera un vector estático:

```
p[3] = 0;
cout << p[5];
```

La memoria reservada para el vector se libera de la siguiente forma:

```
delete [] variable;
```

Ejemplo

En el ejemplo anterior:

```
delete [] p;
```

8.7. Punteros a estructuras

También es posible crear variables dinámicas que sean estructuras:

Ejemplo

```
struct Complejo
{
    float re;
    float im;
};

typedef Complejo * Comp_ptr;

int main()
```

```
{
    Comp_ptr p;

    p = new Complejo;
    (*p).re = 0;
    (*p).im = 0;
...
}
```

Para simplificar el uso de los punteros a estructuras existe un operador para realizar el acceso a los campos de la estructura, de manera que las dos líneas últimas se pueden escribir también de la siguiente manera:

```
    p -> re = 0;
    p -> im = 0;
```

Un caso especial de estructuras dinámicas son las denominadas *estructuras dinámicas recursivas*, son estructuras que contienen algún campo de tipo puntero a la estructura:

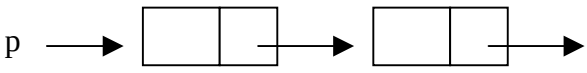
```
struct Elemento;

typedef struct Elemento Ptr;

struct Elemento       // Estructura de datos recursiva
{
    int numero;
    Ptr sig;     // Puntero a la estructura Elemento
};

typedef Ptr Lista;
```

Con estas estructuras se pueden crear tipos de datos muy complejos. Por ejemplo con la anterior estructura se pueden crear listas:



Una función para mostrar los elementos de la lista por pantalla podría ser la siguiente:

```
void MostrarLista (Lista l)
{
    Ptr p;

    p = l;
    while (p != NULL)
    {
        cout << p -> numero;
        p = p -> sig;
    }

    return;
}
```

Y una función para insertar elementos en esta lista podría ser la siguiente:

```
void InsertarDelante (Lista & l, int dato)
{
    Ptr aux;

    aux = new Elemento;
    aux -> sig = l;
    aux -> numero = dato;
    l = aux;

    return;
}
```

8.8.-Errores frecuentes

Por último, hay que tener en cuenta que cuando se utilizan punteros es muy fácil provocar errores pero muy difícil encontrarlos, ya que normalmente no se generan mensajes de error y el programa simplemente se comporta de manera impredecible.

Para minimizar los errores conviene seguir los siguientes consejos:

1. Hay que evitar utilizar punteros que apunten a variables estáticas y en especial a las variables locales.

2. Todo puntero ha de apuntar a una dirección de memoria reservada, o si no, asignarle el valor `NULL`. Es una buena práctica declarar el puntero con el valor `NULL`.

3. No olvidar reservar memoria (cuando sea necesario) antes de utilizar el puntero.

4. No confundir la asignación de punteros con la asignación de contenidos (`p = q;` con `*p = *q`).

```
/*****************************************************************/
/* Programa para la suma de vectores de enteros de tamanyo variable */
/* usando memoria dinamica                                       */
/*****************************************************************/

#include<iostream.h>

typedef int * Vector;

void MostrarVector(const Vector v, int tam);
void LeerVector(Vector v, int tam);
void SumarVector(const Vector v1, const Vector v2, Vector vr,int tam);

int main()
{
    Vector v1, v2, vr;
    int tam;            // Tamanyo real del vector

    cout << "Introduce el tamaño del vector:";
    cin >> tam;

    // Reservamos memoria para los vectores
    v1 = new int[tam];
    v2 = new int[tam];
    vr = new int[tam];

    cout << "Introduce el primer vector:" << endl;
    LeerVector(v1, tam);
    cout << "Introduce el segundo vector:" << endl;
    LeerVector(v2, tam);

    SumarVector(v1, v2, vr, tam);

    cout << "El vector suma es:" << endl;
    MostrarVector(vr, tam);

    // Liberamos la memoria
    delete [] v1;
    delete [] v2;
    delete [] vr;

    return 0;
}

void MostrarVector(const Vector v, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        cout << v[i] << " ";
    cout << endl;
    return;
}

void LeerVector(Vector v, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        cin >> v[i];
    return;
}
```

```
void SumarVector(const Vector v1, const Vector v2, Vector vr, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        vr[i] = v1[i] + v2[i];
    return;
}
```