Programación Orientada a Objetos - Apuntadores

Pedro Fernando Flores Palmeros
ESIME - ZACATENCO

Abril 2020

1. Introducción a Apuntadores

Una herramienta que hace que C y C++ sean muy versátiles, rápidos y muy usados, es que brindan la capacidad de poder manipular la memoria directamente mediante el uso de apuntadores.

Un apuntador es una variable que guarda una dirección de memoria, así como hay variables que guardan enteros, flotantes o caracteres, el puntero o apuntador guarda la dirección de memoria de una variable.

Suponga que en un programa se crea un variable de tipo entero `int miVariable`, esta variable tendrá un tamaño de 4 bytes, en memoria se vería gráficamente algo así

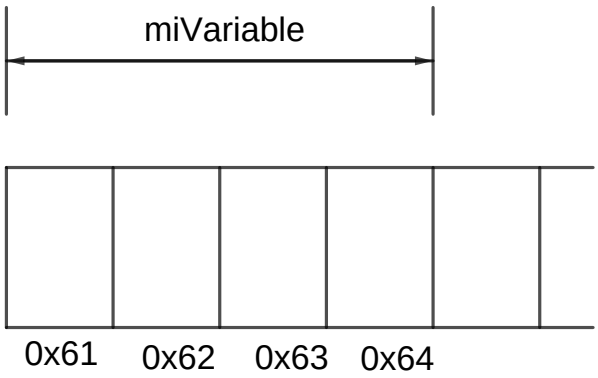

Figura 1: Representación gráfica de la memoria

Donde los números `0x61, 0x62, 0x63, 0x64` son direcciones de memoria, la numeración varía entre computadoras. Por lo general lor programadores no necesitan saber la dirección de memoria tal cual, sin embargo está disponible si se utiliza el operador `&`, tal como se muestra en el siguiente código

```cpp
#include <iostream>

using namespace std;

int main(){
   int miVariable = 8;
   float residuo = 69.7;
   double parametro = -6585.74;

   cout << "El valor de miVariable es: " << miVariable << endl;
   cout << "La direccion de miVariable es: " << &miVariable << endl;
   cout << "-------------" << endl << endl;

   cout << "El valor de residuo es: " << residuo << endl;
   cout << "La direccion de residuo es: " << &residuo << endl;
   cout << "-------------" << endl << endl;

   cout << "El valor de parametro es: " << parametro << endl;
   cout << "La direccion de parametro es: " << &parametro << endl;
   cout << "-------------" << endl << endl;
}
```

Código 1: Obtención de la dirección de una variable

Y la salida en la terminal (que puede variar en la ejecución y en la computadora) sería

```
El valor de miVariable es: 8
La direccion de miVariable es: 0x7ffe21733ad8
-------------

El valor de residuo es: 69.7
La direccion de residuo es: 0x7ffe21733adc
-------------

El valor de parametro es: -6585.74
La direccion de parametro es: 0x7ffe21733ae0
-------------
```

Código 2: Obtención de la dirección de una variable - Salida en terminal

En realidad el programador no necestia conocer el valor numérico de la dirección de las variables. Lo que realmente es importante es saber que cada variable tiene una dirección y que cada variable ocupa cierta cantidad de bytes dentro del sistems. Al declarar una variable no importanto el tipo, el compilador ya sabe cuánta memoria reservar y le asigna automáticamente una dirección.

El apuntador debe de tener su propia memoria, sin embargo, no importa el tipo de dato al que apunte, el apuntador o puntero mantiene el tamaño.

2. Definición y operadores

Los punteros constituyen un nuevo tipo de variables, y su función principal es almacenar información. Sólo en comparación con las otras variables la naturaleza de los datos que almacena es distinta, los punteros guardan o almacenan direcciones de memoria, es dechir, hacen referencia a otra zona de memoria donde se encontrarán los verdaderos datos.

2.1. Operador de dirección

La dirección de memoria de una variable está representada por el nombre de la variable, para conocer la dirección de memoria de una variable concreta se debe de utilizar el **operador dirección** denotado por `&`. Tal como se vió en primer código de este documento.

La sintaxis de la definición de una variable de tipo puntero es la siguiente:

```
tipo_dato * nombre_puntero;
```

donde `tipo_dato` puede ser cualquier dato definido en C o C++, inclusive puede ser una clase o una estructura y `nombre_puntero` es el nombre de la variable tipo puntero.

En el siguiente bloque de código se muestra un ejemplo de declaración de un puntero `miPtr` y este puntero será de naturaleza entero

```
int * miPtr;
```

hasta ahorita sólo se ha creado el puntero, pero este puntero no está apuntando a ninguna variable, y eso puede ser peligroso. **Es una buena práctica de programación inicializar un puntero desde la declaración**, en caso de que aún no se tenga una variable que se le pueda asignar al apuntador, lo recomendable es declarar un **puntero nulo** tal como se muestra en el siguiente código

```
int * miPtr = NULL;
```

De esta manera el apuntador se define como un puntero nulo. Sin embargo, un puntero nulo no tiene mucha utilidada, lo anterior sólo se hizo para proteger el programa. para asignar una dirección se de extraer la dirección de una variable existente y esa dirección se puede asignar a la variable tipo puntero. Observe el siguiente ejemplo.

```
#include <iostream>

using namespace std;

int main(){
```

```
    int miVar = 8;
    int * miPtr = NULL;
    miPtr = &miVar;

    cout << "El valor de miVar es: " << miVar << endl;
    cout << "La direccion de miVar es: " << &miVar << endl;
    cout << "El valor de miPtr es: " << miPtr << endl;

}
```

y la salida de este programa es:

```
El valor de miVar es: 8
La direccion de miVar es: 0x7ffe71732edc
El valor de miPtr es: 0x7ffe71732edc
```

Observe que ahora `miPtr` tiene la dirección de la variable `miVar`. Con lo anterior queda demostrado que el puntero es capaz de guardar la dirección de las variables. Ha sido un gran avance, sin embargo, al inicio de esta sección se mencionó que no es muy útil saber la dirección numérica del puntero.

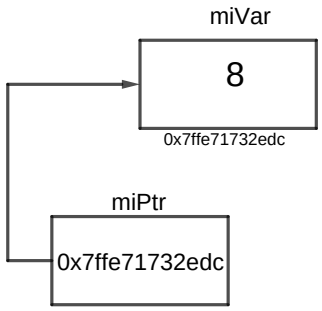

Figura 2: Representación gráfica de la memoria

En la Fig (2) se muestra de manera gráfica el programa anterior. Observe que se han generado dos rectángulos el primero representa a la variable `miVar` y tiene la dirección de memoria `0x7ffe71732edc`, observe que dentro del rectángulo de `miVar` se ha puesto el número 8, que corresponde a la asignación en el momento que se ha declarado la variable. Observe que se ha puesto otro rectángulo que pertenece al puntero `miPtr`, si el puntero estuviera vacío o declarado como nulo, debería de tener la palabra `NULL` dentro pero en la imagen ya se muestra la dirección de la variable `miVar` entonces este puntero ya ha sido asignado.

2.2. Operador de indirección

El **operador de indirección** es el encargado de acceder al dato almacenado en una dirección de memoria, la cual generalmente se obtiene a paratir de una barialge de tipo puntero.

La sintaxis del operadro de indirección es la siguiente

```
* Nombre_puntero
```

Este operador realiza la acción puesta al operador de dirección, permitiendo acceder a los datos localizados en una dirección de memoria. Este operador no se puede aplicar directamente a ninguna variable que no se de tipo puntero.

En otras palabras, para poder inicializar el puntero se le debe de mandar una dirección y se hace através del operador de dirección `&` y para saber el dato al cual se está apuntando se debe de utilizar el operador indirección `*`.

Indirección significa acceder al valor de una variable cuyadirección está guardada en un apuntador. El puntador porporciona una manera indirecta de obtener el valor que se guarda en esa dirección.

Observe el siguiente boque de código.

```
#include <iostream>

using namespace std;

int main(void){
   int Dato = 100;
   int * ptrDato = &Dato;

   int Valor = 0;

   cout << "La direccion de Dato es: " << &Dato << endl;
   cout << "El valor de Dato es: " << Dato << endl;

   cout << "El valor de ptrDato es: " << ptrDato;
   cout << "El valor de la memoria a la que apunta ptrDato es: " << *ptrDato <<
       endl;

   cout << "El valor de Valor es: " << Valor << endl;

   Valor = *ptrDato;

   cout << "El nuevo valor de Valor es: " << Valor << endl;

   *ptrDato = 58;
```

```
   cout << "El nuevo valor de Dato es: " << Dato << endl;
}
```

En donde primero se declara la variable `Dato` y se inicializa con el valor de `100`, después se declara un putero tipo entero `* ptrDato` y se inicializa con la dirección de `Dato` debido a que tiene `int *ptrDato = &Dato`, se declara e inicializa la variable `Valor` y se inicializa con `0`. Se imprimen los valores de la `Dato` y de `ptrDato`, y depués se hace `Valor=*ptrDato`, esta instrucción sería similar a `Valor = Dato`, `*ptrDato` extrae el valor de `Dato` no la dirección, a este hecho se le conoce como *desreferencia o indirección*. Después se hace `*ptrDato = 58` pero como `ptrDato` apunta a `Dato` en realidad se está modificando el valor de `Dato` de manera indirecta a través del puntero.

El operador de indirección `*` se utiliza de dos maneras distintas con lso aputnadores: declaración y desreferencia. Cuando se declara un apuntador el asterisco indica que es unaputador, no una variable normal.

Cuando el puntador es desreferenciado, el operador de indirección indica que se debe de acceder al valor que se encuentra en la dirección de meoria guardada en el apuntador, y no a la dirección en sí.

3. ¿Por qué utilizar apundaores?

La memoria de la computadora básicamente se puede dividir en partes (o por lo menos para el análisis que se muestra en esta sección), la primera es la RAM, que es más rápida, cara y muy limitada en los equipos de cómputo. y está la ROM que generalmente es abundante y muy económica, en un equipo de cómputo la ROM sería el disco duro (para fines prácticos), los programas que se han desarrollado se compilan y cuando se ejecutan se reserva una candiad determinada de memoria en RAM, en esa sección están las variables, funciones, etc, sin embargo, la memoria RAM es limitada. Suponga a hora que quiere hacer varios objetos de alguna clase, si estos objetos son muy grandes comenzarán a consumir más y más memoria RAM, haciendo que el programa y el equipo en general se más lento. Una de las formas de eficientar todo esto es que el programa principal esté en la memoria del programa (RAM) y los objetos o estructuras que son más grandes estén en la memoria ROM y para poder acceder de la RAM a la ROM se utilizan los apuntadores, ya que la ROM es una memoria seccionada y muy organizada.

La memoria asignada a los programas a través del compilador se asigna,reserva y libera de forma automática, sin embargo, la memoria en la ROM debe de ser administrada de manera manual por el programador.

Por lo general, los apuntadores se utilizan para tres cosas:

- Manejar datos en el `heap` (heap en español es montículo y hace referencia a la memoria ROM).

- Tener acceso a los datos miembro y a las funciones de las clases.
- Pasar variables por referencia a las funciones.

4. new

Supongamos que se quiere hacer un apuntador en C++, que sería con la forma que se ha explicado en las secciones anteriores.

```
int * miApuntador;
```

Este apuntador está a la deriva, para asignarle una dirección de memoria en el heap se debe de utilizar la palabra `new` como se muestra en el siguiente bloque de código

```
miApuntador = new int;
```

Si se quisiera declarar e inicializar el apuador, que es lo más recomendable tendría que ser de la siguiente forma:

```
int * miApuntador = new int;
```

5. delete

Al terminar de utilizar la memoria en el heap, es recomendable liberar el espacio utilizado, para esto, es necesario llamar a la instrucción `delete` para liberar la memoria reservada.

```
delete miApuntador;
```

Si se ha reservado memoria a través de `new` y esta no se libera y se termina de ejecutar el programa y la memoria nunca se liberó, entonces esa memoria quedará reservada y no habrá manera de acceder a ella, a menos que se reinicie el equipo, a este fenómeno se le conoce como *fuga de memoria*.

Observe el siguiente bloque de código donde se alcanza a apreciar el uso de las dos nuevas palabras reservadas.

```
#include <iostream>

using namespace std;

int main(){
   int miVariable = 8;
   int * apLocal = &miVariable;
   int * apHeap = new int;

   *apHeap = 90;
```

```
   cout << "El valor de miVariable es: " << miVariable << endl;
   cout << "El valor al que apunto apLocal es: " << *apLocal << endl;
   cout << "El valor al que apunto apHeap es: " << *apHeap << endl;
   delete apHeap;


   int * apHeap_01 = new int;
   *apHeap_01 = 54;
   cout << "*apHeap_01: " << *apHeap_01<< endl;
   delete apHeap_01;
}
```

6. Objetos fuera de la memoria del programa

Así como se puede crear un apuntador a un entero o a un flotante, también se puede crear un apuntador a cualquier objeto. Como se ha mencionado con anterioridad, es recomendable que variables, estructuras y objetos estén ubicados en el Heap y no de manera local para evitar la saturación de memoria.

Considere la clase `Felino` que se muestr a continuación

```
#include <iostream>
#include <string>

using namespace std;

class Perro{
   private:
   int Edad;
   string Nombre;

   public:
   Perro();
   ~Perro();
};

Perro::Perro(){
   cout << "Se ha llamado al constructor" << endl;
   Edad = 1;
   Nombre = "perro";
}

Perro::~Perro(){
   cout << "Se ha llamado al destructor" << endl;
}
```

```cpp
#include <iostream>
#include "perro.h"

using namespace std;

int main(){
   cout << "Se va a crear un Perro local" << endl;
   Perro Pancho;

   cout << "Se va acrear un Perro en el heap" << endl;
   Perro *Orejas = new Perro;

   cout << "Se borrara el Perro del heap" << endl;
   delete Orejas;

   cout << "Fin del programa" << endl;

}
```

```
Se va a crear un Perro local
Se ha llamado al constructor
Se va acrear un Perro en el heap
Se ha llamado al constructor
Se borrara el Perro del heap
Se ha llamado al destructor
Fin del programa
Se ha llamado al destructor
```

Observe que en la salida del programa se crea un objeto de manera local y otro en el Heap, sin embargo, se debe de liberar la memoria del perro creado en el heap, para esto se utiliza `delete`, y cuando se utiliza esta instrucción se manda a llamar al destructor de la clase. Note que el destructor para el perro local no es necesario llamaro debido a que esto lo hace el programa de manera automática.

7. Acceso a los datos miembro de punteros a objetos

Si se tiene un objeto, se ha mencionad en las secciones anteriores, que para poder utilizar o invocar los métodos se hace a través del operador punto (`.`), esto es valido para objetos creados de forma local, Para tener acceso a los métodos de un objeto que está en el heap se tiene que desreferncial el apunto y llamar al operador punto en el objeto. Observe el siguiente ejemplo donde se muestra cómo entrar al método `ladrar()` sólo que en el primer caso es de un objeto creado de manera local y el segundo es de un objeto creado en el heap.

```cpp
PerroLocal.ladrar();
(*PerroHeap).ladrar();
```

Los paréntesis se utilizan para asegurar que `PerroHeap` sea desreferenciado antes de tener acceso a `ladrar()`.

Esto podría parecer bastante extraño, C++ proporciona un perador de método abrebiado para el acceso indirecto: el operador de flecha `->`. de tal manero que las dos líneas que se muestran a continuación son equivalentes

```
(*PerroHeap).ladrar();
PerroHeap->ladrar();
```

A continuación se muestra un ejemplo donde se puede observar de manera completa la implementación de estos términos nuevos.

El archivo cabezera sería el siguiente

```
#include <iostream>
#include <string>

using namespace std;

class Perro{
   private:
   int Edad;
   string Nombre;

   public:
   Perro();
   ~Perro();
   void asignarEdad(int A);
   int obtenerEdad()const;
   void asignarNombre(string N);
   string obtenerNombre()const;
   void ladrar();
   void imprimirInfo();
};

Perro::Perro(){
   cout << "Se ha llamado al constructor" << endl;
   Edad = 1;
   Nombre = "perro";
}

Perro::~Perro(){
   cout << "Se ha llamado al destructor" << endl;
}

void Perro::asignarEdad(int A){ Edad = A; }
int Perro::obtenerEdad()const{ return Edad;}
```

```
void Perro::asignarNombre(string N){ Nombre = N;}
string Perro::obtenerNombre()const{return Nombre;}
void Perro::ladrar(){cout << "Grrrauuuu " << endl; }
void Perro::imprimirInfo(){
   cout << "El perro: " << Nombre << endl;
   cout << "Tiene: " << Edad << " anios de edad" << endl;
}
```

```
#include <iostream>
#include "perro.h"

using namespace std;

int main(){

   cout << "Se crea un perro en en el Heap" << endl;
   Perro *PerroHeap01 = new Perro;

   (*PerroHeap01).asignarNombre("Orejas");
   PerroHeap01->asignarEdad(7);
   (*PerroHeap01).imprimirInfo();
   PerroHeap01->imprimirInfo();

   delete PerroHeap01;

}
```

La salida del programa es

```
Se crea un perro en en el Heap
Se ha llamado al constructor
El perro: Orejas
Tiene: 7 anios de edad
El perro: Orejas
Tiene: 7 anios de edad
```

8. Métodos fuera de la memoria del programa

Uno o más de los datos miembro de una clase pueden ser apuntadores a un objeto que se encuentre en el heap, estos se deben de inicializar en el constructor y también se deben de destruir en el destructor, tal como se muestra en el siguiente bloque de código.

```
#include <iostream>
#include <string>

using namespace std;
```

```
class Perro{
   private:
   int * Edad;
   string * Nombre;

   public:
   Perro();
   ~Perro();
   void asignarEdad(int A);
   int obtenerEdad()const;
   void asignarNombre(string N);
   string obtenerNombre()const;
   void ladrar();
   void imprimirInfo();
};

Perro::Perro(){
   cout << "Se ha llamado al constructor" << endl;
   Edad = new int;
   Nombre = new string;
   *Edad = 5;
   *Nombre = "Dogggi";

}

Perro::~Perro(){
   cout << "Se ha llamado al destructor" << endl;
   cout << "Se eliminaran todos los objetos en el heap" << endl;
   delete Edad;
   delete Nombre;

}

void Perro::asignarEdad(int A){ *Edad = A; }
int Perro::obtenerEdad()const{ return *Edad;}
void Perro::asignarNombre(string N){ *Nombre = N;}
string Perro::obtenerNombre()const{return *Nombre;}
void Perro::ladrar(){cout << "Grrrauuuu " << endl; }
void Perro::imprimirInfo(){
   cout << "El perro: " << *Nombre << endl;
   cout << "Tiene: " << *Edad << " anios de edad" << endl;
}
```

```
#include <iostream>
#include "perro.h"
```

```
using namespace std;

int main(){

   cout << "Se crea un perro en en el Heap" << endl;
   Perro *PerroHeap01 = new Perro;

   (*PerroHeap01).asignarNombre("Orejas");
   PerroHeap01->asignarEdad(7);
   (*PerroHeap01).imprimirInfo();
   PerroHeap01->imprimirInfo();

   delete PerroHeap01;

}
```

9. Apuntador $this->$

Toda función miembro de esta clase tiene un parámetro oculto: el apuntador `this`. El apuntador `this` apunta al propio objeto. Por lo tanto en cada llamada de algún método de la clase se pasa el apuntador `this` de forma oculta. Y es una buena práctica utilizar este apuntador para hacer referencia a los atributos que pertenecen a la clase, de esta manera se evitan confusiones en caso de que pudieran existir variables repetidas.

Observe la siguiente clase y la aplicación del apuntador `this`

```
#include <iostream>
#include <string>

using namespace std;

class Perro{
   private:
   int Edad;
   string Nombre;

   public:
   Perro();
   ~Perro();
   void asignarEdad(int Edad);
   int obtenerEdad()const;
   void asignarNombre(string Nombre);
   string obtenerNombre()const;
   void ladrar();
   void imprimirInfo();
};
```

```
Perro::Perro(){
   cout << "Se ha llamado al constructor" << endl;
   this->Edad = 1;
   this->Nombre = "perro";
}

Perro::~Perro(){
   cout << "Se ha llamado al destructor" << endl;
}

void Perro::asignarEdad(int Edad){ this->Edad = Edad; }
int Perro::obtenerEdad()const{ return this->Edad;}
void Perro::asignarNombre(string Nombre){ this->Nombre = Nombre;}
string Perro::obtenerNombre()const{return this->Nombre;}
void Perro::ladrar(){cout << "Grrrauuuu " << endl; }
void Perro::imprimirInfo(){
   cout << "El perro: " << this->Nombre << endl;
   cout << "Tiene: " << this->Edad << " anios de edad" << endl;
}
```

10. Apuntadores const

```
#include <iostream>

using namespace std;

int main(){
   int A = 7;
   int B = 155;
   int C = -97;

   int *ptrA = &A;

   cout << "ptrA est apuntando a A y el valor de *ptrA es: " << *ptrA << endl;
   *ptrA = -978;
   cout << "Se ha cambiando el valor de *ptrA, se cambia el valor de A: " <<
       *ptrA << endl;

   ptrA = &B;

   cout << "ptrA est apuntando a B y el valor de *ptrA es: " << *ptrA << endl;


   const int * ptrB = &B;
```

```
    cout << "ptrB est apuntando a B y el valor de *ptrB es: " << *ptrB << endl;


    // Si se descomenta maracara error
    //*ptrB = -1;
    //cout << "se ha cambiado *ptrB, ahora B vale: " << *ptrB << endl;

    int * const ptrC = &C;

    cout << "ptrC apunta a C" << endl;
    cout << "Se quiere cambiar el apuntador: " << endl;

    ptrC = &A;
}
```

11. Apuntadores const y funciones miembro const