Apuntadores en *C* y *C++*

Universidad de Carabobo
Facultad Experimental de Ciencias y Tecnología
Prof. Marcos A. Gil T.

8 de diciembre de 2004

1. Introducción

Los apuntadores en *C* y *C++* son una herramienta muy potente de programación que suele causar mucha confusión en los estudiantes que la están aprendiendo. Además, cuando los programadores cometen un error en su utilización, puede ser muy difícil encontrar el error, por lo cual es importante saber utilizarlos muy bien. El uso de apuntadores en *C* y *C++* es muy importante debido a que permite hacer los programas más eficientes y más flexibles. En en este artículo se explica de una manera sencilla y breve todo lo referente a la utilización de apuntadores tanto en *C* como en *C++*.

Todo lo explicado en este artículo aplica tanto para *C* como para *C++*, a menos que se especifique un lenguaje en particular. En algunos ejemplos de código que son aplicables a *C* aparecen instrucciones de entrada y salida de las librerías estándar de *C++*.

2. Definición de apuntador

Cuando se declara una variable, el compilador reserva un espacio de memoria para ella y asocia el nombre de ésta a la dirección de memoria desde donde comienzan los datos de esa variable. Las direcciones de memoria se suelen describir como números en hexadecimal.

Un apuntador es una variable cuyo valor es la dirección de memoria de otra variable. Se dice que un apuntador "apunta" a la variable cuyo valor se almacena a partir de la dirección de memoria que contiene el apuntador. Por ejemplo, si un apuntador p almacena la dirección de una variable x, se dice que "p apunta a x".

3. Referenciación

La referenciación es la obtención de la dirección de una variable. En *C* y *C++* esto se hace a través del operador '&', aplicado a la variable a la cual se desea saber su dirección. Nótese que se trata de un operador unario. Ejemplo:

Código *C* y *C++*

```
int x = 25;
cout << "La dirección de x es: " << &x << endl;
```

Este código imprime un valor del estilo "`0x4fffd34`". Este valor puede variar durante cada ejecución del programa, debido a que el programa puede reservar distintos espacios de memoria durante cada ejecución.

4. Declaración de apuntadores

Para declarar un apuntador se especifica el tipo de dato al que apunta, el operador '*', y el nombre del apuntador. La sintaxis es la siguiente:

<tipo de dato apuntado> *<indentificador del apuntador>

A continuación se muestran varios ejemplos:

Código *C* y *C++*

```
int *ptr1;         // Apuntador a un dato de tipo entero (int)
char *cad1, *cad2; // Dos apuntadores a datos de tipo carácter (char)
float *ptr2;       // Apuntador a un dato de tipo punto-flotante (float)
```

5. Asignación de apuntadores

Se pueden asignar a un apuntador direcciones de variables a través del operador de referenciación ('&') o direcciones almacenadas en otros apuntadores. Ejemplos:

Código *C* y *C++*

```
int i = 5;
int *p, *q;

p = &i;     // Se le asigna a 'p' la dirección de 'i'
q = p;      // Se le asigna a 'q' la dirección almacenada en 'p' (la misma de 'i')
```

6. Desreferenciación de apuntadores

La desreferenciación es la obtención del valor almacenado en el espacio de memoria donde apunta un apuntador. En *C* y *C++* esto se hace a través del operador '*', aplicado al apuntador que contiene la dirección del valor. Nótese que se trata de un operador unario. Ejemplos:

Código *C* y *C++*

```
int x = 17, y;
int *p;

p = &x;
cout << "El valor de x es: " << *p << endl; // Imprime 17
y = *p + 3;  // A 'y' se le asigna 20
```

C++ además provee el operador binario '->', utilizado para obtener campos de un registro con un apuntador al mismo de una manera más fácil y legible. Muchos compiladores de *C* también soportan este operador. Ejemplo:

Código *C++*

```
struct Data
{
  char nombre[20];
  int edad;
};

Data d;
Data *pd = &d;
(*pd).edad = 23; // Acceso al campo 'edad' utilizando el operador '.'
pd->edad = 23;   // Acceso al campo 'edad' utilizando el operador '->'
```

7. Verificación de tipos en apuntadores

Al igual que el resto de las variables, los apuntadores se enlazan a tipos de datos específicos (apuntadores a variables de cierto tipo), de manera que a un apuntador sólo se le pueden asignar direcciones de variables del tipo especificado en la declaración del apuntador. Ejemplo:

Código *C* y *C++*

```
int *p1;
float *p2;
int x;

p1 = &x; // Esto es válido
p2 = &x; // Esto no es válido (el compilador genera un error)
```

8. Direcciones inválidas y la dirección *NULL*

Normalmente, un apuntador inicializado adecuadamente apunta a alguna posición específica de la memoria. Sin embargo, algunas veces es posible que un apuntador no contenga una dirección válida, en cuyo caso es incorrecto desreferenciarlo (obtener el valor al que apunta) porque el programa tendrá un comportamiento impredecible y probablemente erróneo, aunque es posible que funcione bien. Un apuntador puede contener una dirección inválida debido a dos razones:

1. Cuando un apuntador se declara, al igual que cualquier otra variable, el mismo posee un valor cualquiera que no se puede conocer con antelación, hasta que se inicialice con algún valor (dirección). Ejemplo:

   Código *C* y *C++*

   ```
   float *p;

   cout << "El valor apuntado por p es: " << *p << endl; // Incorrecto
   *p = 3.5; // Incorrecto
   ```

2. Después de que un apuntador ha sido inicializado, la dirección que posee puede dejar de ser válida si se libera la memoria reservada en esa dirección, ya sea porque la variable asociada termina su ámbito o porque ese espacio de memoria fue reservado dinámicamente y luego se liberó[1]. Ejemplo:

[1] La asignación dinámica de memoria se explica en la sección 13

Código *C* y *C++*

```
int *p, y;

void func()
{
  int x = 40;
  p = &x;
  y = *p;  // Correcto
  *p = 23; // Correcto
}

void main()
{
  func();
  y = *p;  // Incorrecto
  *p = 25; // Incorrecto
}
```

Si se intenta desreferenciar un apuntador que contiene una dirección inválida pueden ocurrir cosas como las siguientes:

- Se obtiene un valor incorrecto en una o más variables debido a que no fue debidamente inicializada la zona de memoria que se accede a través de la dirección en cuestión. Esto puede ocasionar que el programa genere resultados incorrectos.
- Si casualmente la dirección es la misma de otra variable utilizada en el programa, o está dentro del rango de direcciones de una zona de memoria utilizada, existe el riesgo de sobreescribir datos de otras variables.
- Existe la posibilidad de que la dirección esté fuera de la zona de memoria utilizada para almacenar datos y más bien esté, por ejemplo, en la zona donde se almacenan las instrucciones del programa. Al intentar escribir en dicha zona, fácilmente puede ocurrir que el programa genere un error de ejecución y el sistema operativo lo detenga, o que el programa no responda y deje al sistema operativo inestable.
- En muchos casos el sistema operativo detecta el acceso inadecuado a una dirección de memoria, en cuyo caso detiene abruptamente el programa.

Cuando no se desea que un apuntador apunte a algo, se le suele asignar el valor *NULL*, en cuyo caso se dice que el apuntador es nulo (no apunta a nada). *NULL* es una macro típicamente definida en archivos de cabecera como *stdef.h* y *stdlib.h*. Normalmente, en *C++* se encuentra disponible sin incluir ningún archivo de cabecera. *NULL* se suele definir en estas librerías así:

Código *C* y *C++*

```
#define NULL 0
```

Un apuntador nulo se utiliza para proporcionar a un programa un medio de conocer cuándo un apuntador contiene una dirección válida. Se suele utilizar un *test* condicional para saber si un apuntador es nulo o no lo es, y tomar las medidas necesarias. El valor *NULL* es muy útil para la construcción de estructuras de datos dinámicas, como las listas enlazadas, matrices esparcidas, etc. Es igualmente incorrecto desreferenciar el valor *NULL* por las mismas razones presentadas previamente.

9. Apuntadores a apuntadores

Dado que un apuntador es una variable que apunta a otra, fácilmente se puede deducir que pueden existir apuntadores a apuntadores, y a su vez los segundos pueden apuntar a apuntadores, y así sucesivamente. Estos

apuntadores se declaran colocando tantos asteriscos (‘*’) como sea necesario. Ejemplo:

Código *C* y *C++*

```
char c = 'z';
char *pc = &c;
char **ppc = &pc;
char ***pppc = &ppc;
***pppc = 'm'; // Cambia el valor de c a 'm'
```

10. Apuntadores constantes y apuntadores a constantes

Es posible declarar apuntadores constantes. De esta manera, no se permite la modificación de la dirección almacenada en el apuntador, pero sí se permite la modificación del valor al que apunta. Ejemplo:

Código *C* y *C++*

```
int x = 5, y = 7;
int *const p = &x; // Declaración e inicialización del apuntador constante
*p = 3;            // Esto es válido
p = &y;            // Esto no es válido (el compilador genera un error)
```

También es posible declarar apuntadores a datos constantes. Esto hace que no sea posible modificar el valor al que apunta el apuntador. Ejemplo:

Código *C* y *C++*

```
int x = 5, y = 7;
const int *p = &x; // Declaración e inicialización del apuntador a constante
p = &y;            // Esto es válido
*p = 3;            // Esto no es válido (el compilador genera un error)
y = 3;             // Esto es válido
```

11. Apuntadores, arreglos y aritmética de apuntadores

Los arreglos y apuntadores están fuertemente relacionados. El nombre de un arreglo es simplemente un apuntador constante al inicio del arreglo. Se pueden direccionar arreglos como si fueran apuntadores y apuntadores como si fueran arreglos. Ejemplos:

Código *C* y *C++*

```
int lista_arr[5] = {10, 20, 30, 40, 50};
int *lista_ptr;
lista_ptr = lista_arr; // A partir de aquí ambas variables apuntan al mismo sitio
cout << lista_arr[0];  // Imprime 10
cout << lista_ptr[0];  // Instrucción equivalente a la anterior
cout << *lista_arr;    // Instrucción equivalente a la anterior
cout << *lista_ptr;    // Instrucción equivalente a la anterior
cout << lista_arr[3];  // Imprime 40
cout << lista_ptr[3];  // Instrucción equivalente a la anterior
```

Es posible sumar y restar valores enteros a un apuntador. El resultado de estas operaciones es el desplazamiento de la dirección de memoria hacia adelante (suma) o hacia atrás (resta) por bloques de bytes del tamaño del tipo de dato apuntado por el apuntador. Esto permite recorrer arreglos utilizando apuntadores. Ejemplos:

Código *C* y *C++*

```
int lista[5] = {10, 20, 30, 40, 50};
int *p;
char cad[15];
char *q;

p = &lista[3];      // 'p' almacena la dirección de la posición 3 del arreglo
p = lista + 3;      // Instrucción equivalente a la anterior
cout << lista[2];   // Imprime 30;
cout << *(lista+2); // Instrucción equivalente a la anterior

// Las siguientes instrucciones imprimen la palabra "Programando"
/* Nota: Recuérdese que una constante de cadena de caracteres es
   una secuencia de caracteres en memoria seguidos del cáracter nulo */
strcpy(cad, "Programando");
for (q = cad; *q != '\0'; q++)
  cout << q;
```

También es posible restar dos apuntadores. El resultado de esta operación es el número de bloques de bytes que hay entre las dos direcciones del tamaño del tipo de dato apuntado por los apuntadores. Ejemplo:

Código *C* y *C++*

```
double x[5] = {1.1, 2.1, 3.1, 4.1, 5.1};
double *p = &x[1],
       *q = &x[4];
int n;
n = q - p; // a 'n' se le asigna 3
```

12. Apuntadores para paso de parámetros por referencia

El lenguaje *C* no provee una manera de pasar parámetros por referencia. Sin embargo, es posible hacerlo a través del uso de apuntadores. A continuación se muestra un ejemplo del paso de un parámetro por referencia en *C++*, y luego un código equivalente en *C* o *C++* utilizando un apuntador:

Código *C++*

```
void suma(int a, int b, int& r)
{
  r = a + b;
}

void main()
{
  int x;
  suma(7, 5, x);
  cout << "7 + 5 = " << x;
}
```

Código *C* y *C++*

```
void suma(int a, int b, int *r)
{
  *r = a + b;
}

void main()
{
  int x;
  suma(7, 5, &x);
  cout << "7 + 5 = " << x;
}
```

Nótese que en ambos casos se utiliza el operador '&' para cosas distintas. El operador '&' tiene dos significados como operador unario: señalación de parámetro por referencia y operador de referenciación.

13. Asignación dinámica de memoria

Los programas pueden crear variables globales o locales. Las variables declaradas globales en sus programas se almacenan en posiciones fijas de memoria, en la zona conocida como *segmento de datos* del programa, y todas las funciones pueden utilizar estas variables. Las variables locales se almacenan en la *pila* (*stack*) y existen sólo mientras están activas las funciones donde están declaradas. En ambos casos el espacio de almacenamiento se reserva en el momento de la compilación del programa.

También es posible reservar y utilizar memoria dinámicamente, tomada de la zona de memoria llamada *montículo* (*heap*) o almacén libre. En *C* están disponibles varias funciones que permiten realizar reservar y librerar memoria, pero *C++* además provee un método más fácil y seguro de hacerlo.

13.1. Asignación dinámica de memoria al estilo de *C*

Para asignar memoria dinámicamente en *C* se utilizan las funciones `malloc()` y `free()`, definidas típicamente en el archivo de cabecera *stdlib.h*. La función `malloc()` reserva memoria y retorna su dirección, o retorna *NULL* en caso de no haber conseguido suficiente memoria; y la función `free()` permite liberar la memoria reservada a través de un apuntador. La sintaxis de ambas funciones es como sigue:

Código *C* y *C++*

```
void *malloc(size_t tam_bloque); // size_t es un tipo de datos entero.
                                 // tam_bloque es el tamaño en bytes
                                 // del bloque de memoria a reservar.

void free(void *bloque);         // bloque es un apuntador a la zona
                                 // de memoria a liberar.
```

Como se puede apreciar, `malloc()` reserva memoria sin importar el tipo de datos que se almacenará en ella, retornando un apuntador a `void`. Por esta razón, se hace necesario convertir el tipo de apuntador al tipo del apuntador que guardará la dirección. También es necesario calcular exactamente cuántos bytes se requieren reservar, ayudándose con el uso del operador `sizeof`. Ejemplos:

Código *C* y *C++*

```
typedef struct
{
  char nombre[20];
  int edad;
} Data;

Data *p_data; // Declaración de un apuntador a Data
int i;

p_data = (Data*) malloc(sizeof(Data)); // Reservación de memoria para un registro

if (p_data != NULL)                    // Verificación de reservación
{
  strcpy(p_data->nombre, "Rachel");    // Inicialización de datos
  p_data->edad = 21;                   // en la memoria reservada

  free(p_data);                        // Liberación de memoria
}

// Reservación de memoria para un arreglo de 10 registros
p_data = (Data*) malloc(sizeof(Data)*10);

if (p_data != NULL) // Verificación de reservación
{
  // Lectura de datos del arreglo
  for (i = 0; i < 10; i++)
    cin >> p_data[i].nombre >> p_data[i].edad;

  // Liberación de memoria del arreglo
  free(p_data);
}
```

13.2. Asignación dinámica de memoria al estilo de *C++*

Para asignar memoria dinámicamente en *C++* se utilizan los operadores `new` y `delete`. El operador `new` reserva memoria para un tipo de datos específico y retorna su dirección, o retorna *NULL* en caso de no haber conseguido suficiente memoria; y el operador `delete` permite liberar la memoria reservada a través de un apuntador. La sintaxis de ambos operadores es como sigue:

Para reservar y liberar un solo bloque:

```
<apuntador> = new <tipo de dato>
delete <apuntador>
```

Para reservar y liberar varios bloques (un arreglo):

```
<apuntador> = new <tipo de dato>[<número de bloques>]
delete [] <apuntador>
```

El tipo del apuntador especificado del lado izquierdo del operador `new` debe coincidir con el tipo especificado del lado derecho. De no ser así, se produce un error de compilación. Ejemplos:

Código *C++*

```
struct Data
{
  char nombre[20];
  int edad;
};

Data *p_data; // Declaración de un apuntador a Data
int i;

p_data = new Data; // Reservación de memoria para un registro

if (p_data != NULL)                    // Verificación de reservación
{
  strcpy(p_data->nombre, "Rachel");    // Inicialización de datos
  p_data->edad = 21;                   // en la memoria reservada

  delete p_data;                       // Liberación de memoria
}

// Reservación de memoria para un arreglo de 10 registros
p_data = new Data[10];

if (p_data != NULL) // Verificación de reservación
{
  // Lectura de datos del arreglo
  for (i = 0; i < 10; i++)
    cin >> p_data[i].nombre >> p_data[i].edad;

  // Liberación de memoria del arreglo
  delete [] p_data;
}
```

14. Bibliografía

El presente artículo fue realizado con la ayuda del libro "Programación en *C++*", del autor Luis Joyanes Aguilar, editorial *Mc Graw Hill*.