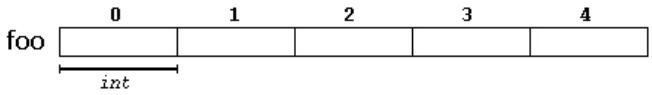
UNIVERSIDAD DEL VALLE, FACULTAD DE CIENCIAS, DEPARTAMENTO DE FISICA

CAPÍTULO III

3.1 Arreglos

Un arreglo es una serie de elementos de un mismo tipo colocados en un espacio de memoria de manera contigua de modo que pueden ser referenciados de manera individual agregando un índice para identificarlos de forma única.
De tal manera que 5 valores de tipo int pueden ser declarados en un arreglo sin tener que declarar 5 variables diferentes. En lugar de ésto usando un arreglo int llamado foo de 5 valores int almacenados en memoria de modo contiguo, todos los 5 se pueden identificar usando el mismo identificador con el indice apropiado



cada espacio en blanco representa un elemento del arreglo. Estos elementos son números de 0 a 4. En C++ el primer elemento de un arreglo es SIEMPRE enumerado con cero (no uno) no importa lo largo del arreglo.
Al igual que una variable regular, un arreglo DEBE ser declarado antes de ser usado de la manera:

```
tipo nombre [elementos];
```

donde `tipo` es un tipo de dato válido (`int`, `float`...), `nombre` es nombre o identificador válido y el espácio de `elementos` el cual está siempre encerrado en paréntesis cuadrados ([ ]), especifica la longitud del arreglo en términos del número elementos. De manera que el arreglo foo con cinco elementos de tipo int puede ser declarada:

```
int foo [5];
```

NOTA: El espacio para elementos dentro de los paréntesis cuadrados [ ], representando el número de elementos en el arreglo, debe ser una expersión constante, ya que el arreglo está hecho de bloques de memoría estática cuyo tamaño debe estar determinado en el momento de compilar, antes que el programa-corra.

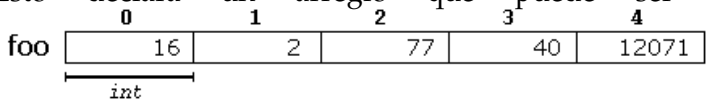
3.1.2 Inicialización de arreglos

Por defecto, los arreglos locales (es decir, los que se encuentran declarados dentro de funciones) son no inicializados a izquierda. Esto significa que a ninguno de sus elementos se le da un valor particular; sus contenidos son indeterminados en el momento que el arreglo se declara. Esto es como en el ejemplo anterior.

Por otro lado los elementos en un arreglo puede ser explicitamente inicializados a valores específicos cuando el arreglo es declarado colocando aquellos valores en corchetes {}, por ejemplo:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

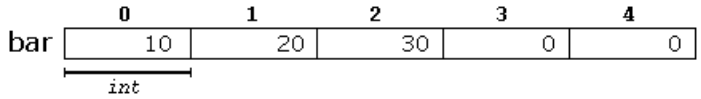
Esto declara un arreglo que puede ser representado de la siguiente manera:



El número de valores entre corchetes { }, no puede ser mayor que el número de elementos en el arreglo. Si se declaran menos elementos, los restantes no declarados se llenarán con ceros, por ejemplo:

```
int bar [5] = { 10, 20, 30 };
```

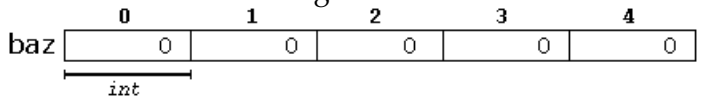
Lo cual se interpreta como:



La inicialización puede ser incluso sin valores, solo los corchetes:

```
int baz [5] = { };
```

Esto crea un arreglo con cinco valores int, cada uno inicializado a cero:



C++ da la posibilidad de dejar vacío los paréntesis cuadrados [ ] cuando se inicializa. En éste caso, el compilador asume automáticamente que le tamaño del arreglo es el número de valores incluidos entre los corchetes { }:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

Luego de ésta declaración el arreglo foo tendrá de longitud 5 int, por lo cual se incializará con un valor de 5.

Finalmente, la evolución de C++ ha llegado a la adopción de una inicialiación universal para arreglos, donde ya no es necesario el signo de igual entre la declaración y la inicialización. Las siguientes dos declaraciones son equivalentes:

```
1 int foo[] = { 10, 20, 30 };
2 int foo[] { 10, 20, 30 };
```

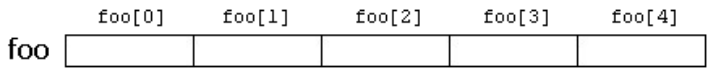
Los arreglos estáticos, y aquellos declarados directamenet en un namespace (fuera de una función), son siempre inicializados. Si no están inicializados explicitamente, todos los elementos por defecto son inicializados a cero.

3.1.3. Manipulación de los elementos de un arreglo

Los valores de cualquier arreglo de elementos pueden ser accesados como cualquier valor de una variable regular del mismo tipo.

La sintaxis es:
`nombre[indice]`

Siguiendo el ejemplo anterior, en el cual foo tiene 5 elementos y cada uno de ellos tiene un tipo int, el nombre con el cual pueden ser usados cada elemento es el siguiente:



Por ejemplo, la siguiente declaración guarda el valor 75 en el tercer elemento del arreglo foo:

```
foo [2] = 75;
```

Y por ejemplo, se puede copiar el valor del tercer elemento del arreglo foo a la variable x previamente declarada como int:

```
x = foo[2];
```

La expresión foo[2] es en si misma una variable de tipo int. Note que el tercer elemento del arreglo foo es foo[2], ya que el primero es foo[0], y el último es foo[4]. De tal manera que se se escribe foo[5], se está solicitando el sexto elemento de foo y por tanto se excede el tamaño real del arreglo.

En C++, es una sintaxis correcta exceder el rango válido de los indices de un arreglo. Esto puede crear problemas, ya que está accediendo a elementos fuera del rango, pero no causa errores en la compilación, pero causa errores al correr el programa. La razón de ésto se vera cuando sean introducidos los pointers.

Es importante distinguir claramente entre los dos usos de los parénesis cuadrados [ ] en los arreglos. Estas dos maneres determinan diferentes tareas: una es especificar el tamaño del arreglo cuando los arreglos son declarados. La segunda es para especificar el indice para acceder a un elemento concreto del arreglo. No se pueden confundir los dos usos de los paréntesis en los arreglos.

```
1 int foo[5];         // declaración de un nuevo arreglo
2 foo[2] = 75;        // acceso a un elemento de un arreglo.
```

La mayor diferencia es que la declaración está precedida por el tipo de dato de los elementos, mientras que el acceso a los elementos no.

Operaciones válidas con arreglos:

```
1 foo[0] = a;
2 foo[a] = 75;
3 b = foo [a+2];
4 foo[foo[a]] = foo[2] + 5;
```

Por ejemplo:

Código fuente 26:

```
1  // ejemplo de arreglo
2  #include <iostream>
3  using namespace std;
4
5  int foo [] = {16, 2, 77, 40, 12071};
6  int n, resultado=0;
7
8  int main ()
9  {
10   for ( n=0 ; n<5 ; ++n )
11   {
12     resultado += foo[n];
```

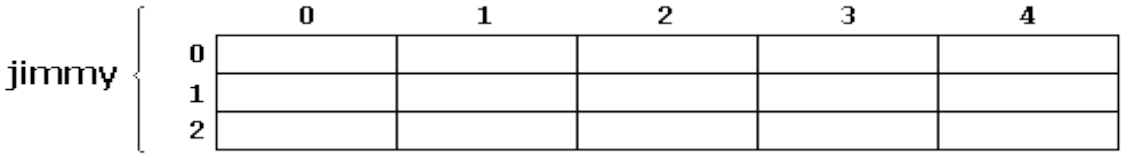
Salida:

```
12206
```

```
13   }
14   cout << resultado;
15   return 0;
16 }
```

3.1.4. Arreglos multidimensionales

Arreglos multidimensionales pueden ser descritos como "arreglos de arreglos". Por ejemplo, un arreglo bidimensional puede ser visto como una tabla con elementos organizados en dos dimensiones, todos ellos con el mismo tipo de dato.

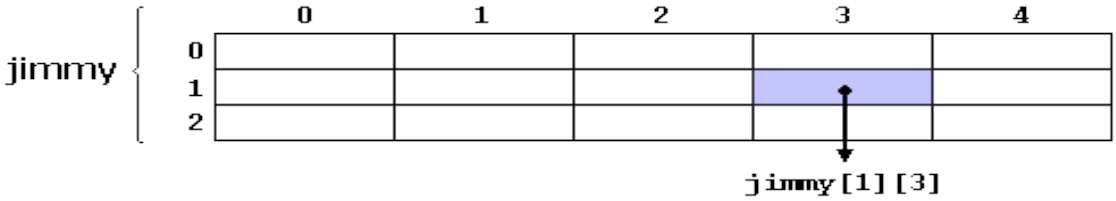


`jimmy` representa un arreglo bidimensional de 3 por 5 elementos de tipo `int`. La sintaxis para esto es:

```
int jimmy [3][5];
```

y por ejemplo, la manera de referenciar la segunda fila – cuarta columna es:

```
jimmy[1][3]
```



(recuerde que los índices de los arreglos SIEMPRE comienzan con CERO)

Los arreglos multidimensionales no están limitados a dos indices (esto es, dos dimensiones). Los arreglos pueden tener tantos indices como necesite. Tenga cuidado: a cantidad de memoria necesaria para un arreglo incrementa exponencialmente con cada dimensión. Por ejemplo:

```
char century [100][365][24][60][60];
```

En el ejemplo se declara un arreglo con elementos tipo `char` para cada segundo en un siglo. Esto suma más de 3 billones de `char`....quiere decir que esta declaración consumiría mas de 3 GB de memoria!!!.

Tenga en cuanta que los arreglos multidimensionales son una abstracción adicional, y en ocasiones puede procesar la misma información usando un arreglo más sencillo, un ejemplo muy particular:

```
1 int jimmy [3][5];   // es equivalente a
```

```
int jimmy [15];      // (3 * 5 = 15)
```

La única diferencia que un arreglo multidimensional proporciona es el hecho que el compilador recuerda automáticamente la dimensión en memoria. Los siguientes códigos producen el mismo resultado, donde uno utiliza un arreglo bidimensional y el otro un arreglo simple:

Arreglo multidimensional	Arreglo pseudo-multidimensional

`#define COLUMNAS 5`
`#define FILAS 3`
 	
`int jimmy [FILAS][COLUMNAS];`
`int n,m;`
 	
`int main ()`
`{`
`  for (n=0; n<FILAS; n++)`
`    for (m=0; m<COLUMNAS; m++)`
`    {`
`      jimmy[n][m]=(n+1)*(m+1);`
`    }`
`}`
	`#define COLUMNAS 5`
	`#define FILAS 3`
	
	`int jimmy [FILAS * COLUMNAS];`
	`int n,m;`
	
	`int main ()`
	`{`
	`  for (n=0; n<FILAS; n++)`
	`    for (m=0; m<COLUMNAS; m++)`
	`    {`
	`      jimmy[n*WIDTH+m]=(n+1)*(m+1);`
	`    }`
	`}`

Ninguno de los dos códigos producen una salida en pantalla, pero ambos asignan valores al bloque de memoria llamado jimmy de la siguiente manera:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Note que los códigos usados definen constantes para las filas y columnas en lugar de usar sus valores directamente. Esto le da al código facilidad de lectura y permite realizar cambios en el código de fácilmente.

3.1.5. Arrays como parametros en funciones

Una herramienta importante es pasar un arreglo a una función como un parámetro. En C++ **NO** es posible pasaar el bloque entero de memoria representado por el arreglo a una función directamente como argumento. Pero lo **SI** es posible es pasar su dirección. En la práctica, el efecto es el mismo y la operación es más rápida y eficiente.

Para que una función acepte un arreglo como parámetro, los parámetros pueden ser declarados como un arreglo pero con parétesis cuadrados vacíos, omitiendo el tamaño real del arreglo. Por ejemplo:

```
void procedimiento (int arg[])
```

Esta función acepta un parámetro tipo "arreglo de `int`" llamado `arg`. Para pasar a esta función un arreglo previamente declarado como el siguiente:

```
int myarray [40];
```

Sería suficiente con escribir un llamado de la función como el siguiente:

```
procedimiento (myarray);
```

Por ejemplo:

Código fuente 27: Salida:

```
1  // arreglos como parámetros de funciones
2  #include <iostream>
3  using namespace std;
4
5  void printarray (int arg[], int length) {
6    for (int n=0; n<length; ++n)
7      cout << arg[n] << ' ';
8    cout << '\n';
9  }
10
11 int main ()
12 {
13   int firstarray[] = {5, 10, 15};
14   int secondarray[] = {2, 4, 6, 8, 10};
15   printarray (firstarray,3);
16   printarray (secondarray,5);
17 }
```

```
5 10 15
2 4 6 8 10
```

En este código, el primer parámetro (`int arg[]`) acepta un arreglo cuyos elementos son de tipo `int`, cualquiera que sea su longitud. Por esta razón, se incluye un segundo parámetro que dice a la función la longitud del arreglo. Esto permite al loop del for que dibuje el arreglo para saber como realizar la iteración.

En una declaración de función, también es posible incluir un arreglo multidimensional. La sintaxis para un arreglo tridimensional está dado por:

```
base_type[][depth][depth]
```

Por ejemplo, una función con un arreglo multidimensional como argumento puede ser:

```
void procedimiento (int myarray[][3][4])
```

Note que los primeros paréntesis cuadrados [ ] están vacíos, mientras que los otros especifican sus tamaños de dimensión. Ésto es necesario para que el compilador permita determinar el tamaño de las dimensiones adicionales.

De alguna manera, pasar un arreglo como argumento SIEMPRE pierde una dimensión. La razón dentras de ésto es histórica, los arreglos no pueden ser copiados y lo que realmente se pasa es un puntero. Esto es una fuente de error común en los nuevos programadores. Los punteros serán explicados después.

3.1.6. Librería de arreglos

Los arreglos explicados anteriormente son implementados directamente con el lenguaje plano heredado de C. Para mejorar el problema de la copia y los punteros el lenguaje C++ provee un tipo de arreglo alternativo al estandar. Este es un tipo de plantilla "template" (en realidad una clase template) definida en el encabezado `<array>` . Que contiene una librería característica la cual no se explicará en

detalle pero es suficiente decir que opera de una manera similar como ya se ha explicado pero que permite copiar los arreglos, sin embargo esto es una operación costosa copiando bloques entenero de memoria, así que debe tratarse con cuidado. Y también tiene la posibilidad de trabajar con punteros cuando se especifica explicitamente hacerlo a través de su mienbro `data`.

A modo de ejemplo están estas dos versiones del mismo ejemplo usando la construcción de arreglos explicados anteriormente y la librería de arreglos de la presente sección:

Construcción de arreglos	Libraría de arreglos

`#include <iostream>`
 	
`using namespace std;`
 	
`int main()`
`{`
`  int myarray[3] = {10,20,30};`
 	
`  for (int i=0; i<3; ++i)`
`    ++myarray[i];`
 	
`  for (int elem : myarray)`
`    cout << elem << '\n';`
`}`
	`#include <iostream>`
	`#include <array>`
	`using namespace std;`
	
	`int main()`
	`{`
	`  array<int,3> myarray {10,20,30};`
	
	`  for (int i=0; i<myarray.size(); ++i)`
	`    ++myarray[i];`
	
	`  for (int elem : myarray)`
	`    cout << elem << '\n';`
	`}`

Como se puede ver, ambas clases de arreglos usan la misma sintaxis para accesar a los elementos `myarray[i]`. La mayor diferencia radica en la declaración del arreglo y la inclusión de la librería de arreglos. Note que el acceso al tamaño del arreglo es muy sencilla usando esta librería.

3.2. Punteros

En el primer capítulo se hablaron de las variables como una locación de memoria del computador que puede ser accedida por su identificador o nombre. Y el computador cada vez que necesite usa el identificador (con su espacio en memoria) para referirse a la variable.

Para C++, la memoria del computador es como una sucesión de celdas de memoria con una dirección única. Este memoria de celdas de único bit son ordenadas de tal manera que permite representaciones de datos mucho más largas que un bit para ocupar las celdas de memoria que tienen direcciones consecutivas.

Cuando las variables son declaradas, la memoria necesaria para almacenar su valor asigna una locación específica en memoria (su dirección de memoria). Así que puede ser útil para un programa ser capaz de obtener la dirección de una variable durante su computo para poder acceder a los datos que están en dicha posición.

3.2.1 Operador de Dirección de Referencia (&)

La dirección de una variable puede ser obtenida precediendo el nombre de una variable con un signo

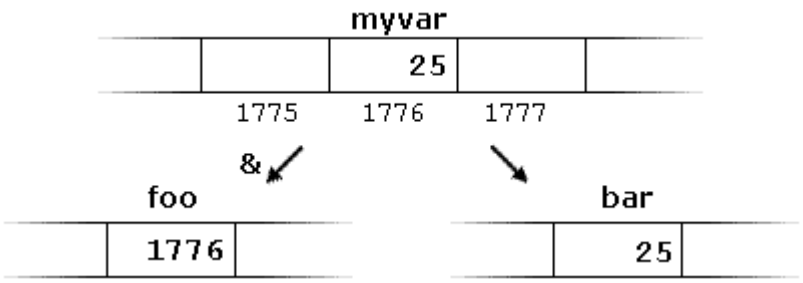
de ampersand (&), conocido como el operador de dirección. Por ejemplo:

```
foo = &myvar;
```

Este asignará la dirección de la variable `myvar` a `foo`; no se está asignando el contenido de la variable como tal a `foo` sino su dirección. El real valor de la variable en memoria no puede ser conocido sino hasta que se corre el programa. Si se conociera que la ubicación en memoria de la variable es 1776, entonces consideremos el siguiente ejemplo:

```
1 myvar = 25;
2 foo = &myvar;
3 bar = myvar;
```

Los valores contenidos en cada variable después de la ejecución del programa son:



Primero se asigna el valor de 25 a `myvar`, donde se asume que la ubicación en memoria de la variable es 1776.

En segunda instancia se asigna a `foo` la dirección de `myvar`, la cual fue asumida se 1776.

Y Finalmente, la tercera declaración asigna el valor contenido en myvar a la varialbe bar. El cual es un procedimiento estándar.
La principal diferencia entre la segunda y la tercera línea es el operador de dirección (&). La variable que almacena la dirección de la otra variable es lo que en C++ se conoce como un puntero.

Los punteros son una herramienta muy poderosa en el lenguaje que tiene muchos usos. En lo siguiente aprendemos como declarar y usar los punteros.

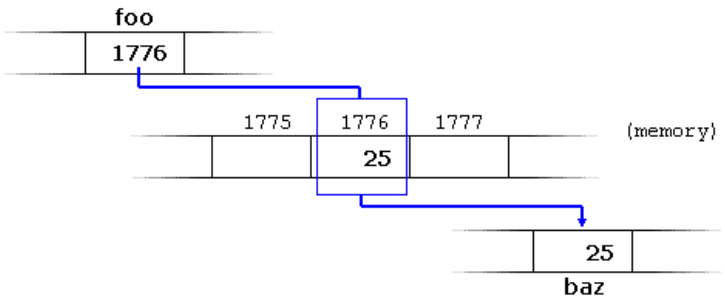
3.2.2. Operador de Desreferencia o de Indirección (*)

Como ya se vio, una variable que almacena la dirección de otra se llama puntero. Los punteros se dice que apuntan a la variable de la dirección que almacena.

Una propiedad interesante de los punteros es que ellos pueden ser usados para acceder a la variable que ellos apuntan. Esto puede ser echo precediendo al nombre del puntero con el operador de desreferencia (*). El operador en si mismo puede ser leido como "el valor apuntado por". Por lo tanto, la siguiente declaración:

```
baz = *foo;
```

Esta puede ser leida como: "baz es igual al valor apuntado por foo", y la declaración asigna el valor de 25 a baz, siendo foo 1776 y el valor apuntado por 1776 es 25 siguiendo el ejemplo anterior.



Es importante dejar en claro que foo tiene asignado el valor 1776 mientras que *foo tiene el valor almacenado en la dirección 1776 cuyo caso es 25:

```
baz = foo;   // baz igual a foo (1776)
baz = *foo;  // baz igual al valor apuntado por foo (25)
```

Los operadores de Referencia y de Desreferencia son complementarios:

- & es el operador de dirección, y se lee como "direcciona a"
- * es el operador de desreferencia, y se lee como "el valor apuntado por"

Tienen significados opuestos: Una dirección obtenida con & puede desreferenciarse con *.

Siguiendo las siguientes asignaciones:

```
myvar = 25;
foo = &myvar;
```

Las siguientes declaraciones son verdaderas:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

Y por lo tanto la siguiente declaración es también verdadera.

```
*foo == myvar
```

3.2. 3. Declaración de Punteros:

Debido a la habilidad de un puntero de referirse directamente al valor que éste apunta es necesario que la declaración de un puntero incluya el tipo de dato que va a apuntar.

La declaración de los punteros sigue la siguiente sintaxis:

`tipo * name;`

donde el tipo es el tipo de dato apuntado por el puntero. Este no es el tipo del puntero en si mismo sino el tipo de dato del valor que el puntero apunta. Por ejemplo:

```
int * numeros;
char * caracteres;
double * decimales;
```

Aunque los tres ejemplos de variables son todos punteros, tiene realmente diferentes tipos, `int*, char*, y double*` respectivamente, dependiendo del tipo de dato al que ellos apuntan. El asterisco (*) usado hace parte de la especificación del tipo compuesto de dato para el puntero. No debe ser confundido con el operador de desreferencia visto antes el cual también es un asterisco (*). Son dos cosas diferentes representadas con el mismo signo.

Ejemplo:
Código fuente 28:

```
// mis primeros punteros
#include <iostream>
using namespace std;

int main ()
{
  int primervalor, segundovalor;
  int * mipuntero;

  mipuntero = &primervalor;
  *mipuntero = 10;
  mipuntero = &segundovalor;
  *mipuntero = 20;
  cout <<"primervalor es " << primervalor << '\n';
  cout <<"segundovalor es " << segundovalor << '\n';
  return 0;
}
```

```
primervalor es 10
segundovalor es 20
```

Note que ni primervalor ni segundovalor tienen asignado un valor directamente en el programa, ambos terminan con un valor asignado de manera indirecta a travéz de mipuntero, de la siguiente manera:

Primero a `mipuntero` se le asigna la dirección de primervalor usando el operador de dirección (&). Luego el valor apuntado por `mipuntero` es asignado el valor de 10. Debido a que en ese momento, `mipuntero` esta apuntando a la alocación de memoria de primervalor, esto efectivamente modifica el valor de primer valor.

Para mostrar que un puntero puede apuntrar a diferentes variables durante su tiempo de vida en el programa, el ejemplo repite el proceso con segundovalor y el mismo puntero `mipuntero`.

Ejemplo:
Código fuente 29:

```
// más punteros
#include <iostream>
using namespace std;

int main ()
{
  int primerv = 5, segundov = 15;
  int * p1, * p2;

  p1 = &primerv;  // p1 = dirección de primerv
```

```
primerv es 10
segundov es 20
```

```
  p2 = &segundov; // p2 = dirección de segundov
  *p1 = 10;          // valor apuntado por p1 = 10
  *p2 = *p1;         // valor apuntado por p2 = valor
apuntado por p1
  p1 = p2;           // p1 = p2 (valor del puntero
es copiado)
  *p1 = 20;          // valor apuntado por p1 = 20

  cout << "primerv es " << primerv << '\n';
  cout << "segundov es " << segundov << '\n';
  return 0;
}
```

Cada operación de asignación incluye un comentario de cómo cada línea puede ser leida reemplazando (&) por "dirección de", y asteriscos (*) por "valor apuntado por".

Note que hay expresiones con los punteros p1 y p2 con y sin operadores de desreferencia (*). El significado de una expresión usando el operador de desreferencia (*) es muy diferente de las que no lo tienen. Cuando el operador precede al nombre del puntero, la expresión se refiere al valor que es apuntado, mientras que el nombre de un puntero aparece sin el operador, este ser refiere al valor del puntero en si mismo esto es la dirección a la cual está apuntando.

En la línea:

```
int * p1, * p2;
```

Se declaran los dos punteros usados, el asterisco (*) está frente a cada nombre, esto en contraposición a

```
int * p1, p2;
```

Donde p1 es un puntero entero y p2 es una variable tipo entero.

Pero mejor utilizar diferentes declaraciones, una para variables y otra para punteros.

3.2.4. Punteros y Arreglos

El concepto de arreglos es relacionado al de punteros, de hecho un arreglo trabaja como un puntero a su primer elemento. Y un arreglo puede ser implicitamente convertido a un puntero de tipo adecuado. Por ejemplo:

```
int myarray [20];
int * mypointer;
```

La siguiente asignación será válida:

```
mypointer = myarray;
```

Luego de la línea anterior mypointer y myarray serían equivalentes y tienen propiedades similares. La diferencia principal es que mypointer puede ser asignada a diferentes direcciones, mientras que myarray nunca puede ser asignada a nada y será siempre representada por el mismo bloque de 20 elementos del tipo int. Por lo tanto la siguiente asignación **no** será válida:

```
myarray = mypointer;
```

Miremos un ejemplo que mezcla arreglos y punteros:

Código fuente 30

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

Salida

```
10, 20, 30, 40, 50,
```

Con los punteros y los arreglos se pueden realizar el mismo tipo de operaciones con el mismo significado. La principal diferencia está en que los punteros pueden ser asignados a nuevas direcciones mientras que los arreglos NO.

En la explicación sobre arreglos, los paréntesis cuadrados ([ ]) fueron explicados para especificar el índice de un elemento de un arreglo. El mismo trabajo realiza el operador (*). Por ejemplo:

```
a[5] = 0;       // a [el valor de 5] = 0
*(a+5) = 0;     // apuntado a (a+5) = 0
```

Estas dos expresiones son equivalentes y válidas, no solo si `a` es un puntero sino también si `a` es un arreglo. Recuerde que si tiene un arreglo puede usarlo justo como un punero a su primer elemento.

3.2.5 Inicialización de punteros

Los punteros pueden ser inicializados para apuntar a ubicaciones específicas cuando son definidos:

```
int myvar;
int * myptr = &myvar;
```

El estado resultante de las variables de este código es equivalente a:

```
int myvar;
int * myptr;
myptr = &myvar;
```

Cuando los punteros son inicializados, lo que se inicializa es la dirección hacia la que apuntan (es decir, `myptr`), y no el valor apuntado (esto es, `*myptr`). Por lo tanto, el código anterior no debe confundirse con el siguiente el cual no tiene sentido y no es válido:

```
int myvar;
```

```
2 int * myptr;
3 *myptr = &myvar;
```

El asterisco (*) en la declaración del puntero (linea 2) indíca que éste último es un puntero, no es el operador de desreferencia (como en la línea 3). Simplemente ambas operaciones emplean el mismo símbolo: (*). Como es usual, los espacios son irrelevantes y jamás cambian el signifacado de la expresión.

Los punteros pueden ser inicializados tanto hacia la dirección de una variable (como en el caso anterior), como hacia el valor de otro puntero (o arreglo):

```
1 int myvar;
2 int *foo = &myvar;
3 int *bar = foo;
```

3.2.6 Aritmética de punteros

Llevar a cabo operaciones aritméticas con punteos es un poco diferente que hacerlo sobre tipos enteros ordinarios. Para comenzar, sólo están permitidas operaciones de adición y sustracción; las otras carecen de sentido en el mundo de los punteros. Sin embargo, ambas operaciones tienen un comportamiento ligeramente diferente sobre los punteros, acorde con el tamaño del tipo de dato a los cuales apuntan.

Cuando los tipos de datos fundamentales fueron introducidos, vimos que diferentes tipos poseen diferentes tamaños. Por ejemplo: `char` tiene un tamaño de 1 byte, `short` es generalmente más grande que eso, e `int` y `long` son aún más grandes; el tamaño exacto de éstos es dependiente del sistema. Por ejemplo, supongase que en un sistema dado, `char` toma 1 byte, `short` toma 2 bytes, y `long` toma 4.

Supongase ahora que se definen tres punteros en este compilador:

```
1 char *mychar;
2 short *myshort;
3 long *mylong;
```

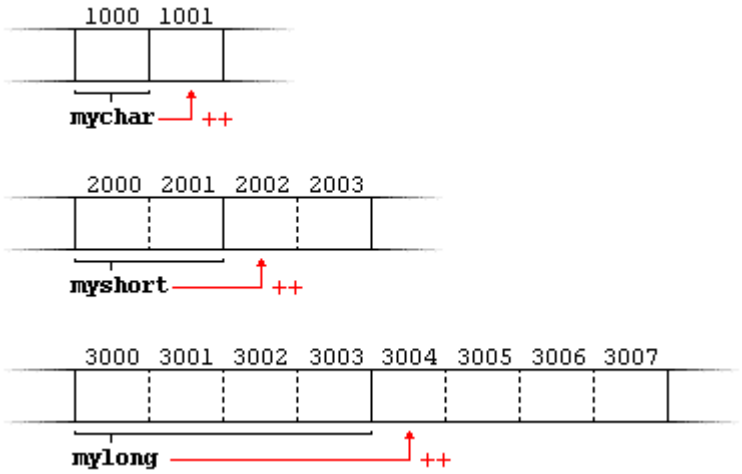
Y que sabemos que apuntan a las ubicaciones en memoria `1000`, `2000`, y `3000`, respectivamente. Por cosiguente, si escribimos:

```
1 ++mychar;
2 ++myshort;
3 ++mylong;
```

`mychar`, como uno esperaría, contendría el valor 1001. Pero no es tan obvio que, `myshort` contendría el valor 2002, y `mylong` contendría 3004, aunque hayan sido incrementados sólo en uno. La razón es que cuando se adiciona uno a un puntero, se hace que el puntero apunte al siguiente elemento del mismo tipo, y, por lo tanto, al puntero se le añaden los bytes

del tipo correspondiente. Esto se aplica tanto en la adición como en la sustracción de cualquier número al puntero. Sucedería exactamente lo mismo si se escribe:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```



Considerando los operadores de incremento (++) o de disminución (--), pueden ser empleados como pre- o sufijos de una expresión, con un comportamiento ligeramente diferente: como prefijo, el incremento se aplica antes de que la expresión sea evaluada, y como sufijo, el incremento se realiza después de evaluar la expresión. De la misma manera sucede a expresiones de incremento o disminución de punteros, las cuales pueden ser parte de expresiones más elaboradas que incluyen operadores de desreferencia (*). Teniendo en cuenta las reglas de jerarquía de los operadores, se recuerda que operadores posfijos, tales como el de incremento y disminución, tienen una jerarquía mayor que los prefijos, tales como el operador de desreferencia (*). Asi, la siguiente expresión:

```
*p++
```

Es equivalente a `*(p++)`. Y lo que hace es incrementar el valor de `p` (de manera que ahora apunta al siguiente elemento), pero debido a que ++ es usado posfijo, la expresión entera es evaluada como el valor apuntado originalmente por el puntero (la dirección apuntda antes de ser incrementada).

Escencialmente, las siguientes son cuatro posibles combinaciones de el operador de desreferencia con las versiones prefijas y sufijas del operador de incremento (lo mismo aplica para el operador de disminución):

```
*p++ //igual a *(p++):incrementa el puntero y dereferencia a la dirección original
*++p //igual a *(++p):incrementa el puntero y dereferencia a la dirección //incrementada
++*p   // igual a ++(*p): dereferencia al puntero, e incrcementa el valor al //que apunta
(*p)++ // dereferencia al puntero, y post-incrementa el valor al que apunta
```

Una declaración típica -pero no tan simple- de estos operadores es:

```
*p++ = *q++;
```

Ya que ++ tiene una jerarquía mayor a `*`, ambos `p` y `q` son incrementadas, pero debido a que ambos operadores de incremento (++) son usados como posfijos y no prefijos, el valor asignaso a `*p` es `*q` antes de que `p` y `q` sean incrementados. Y entonces ambos son incrementados. Sería burdamente igual a:

```
1 *p = *q;
2 ++p;
3 ++q;
```

Como siempre, paréntesis reducen confusión añadiendo legibilidad a las expresiones.

3.2.7 Punteros y `const`

Los punteros pueden ser usados para acceder una variable a través de su dirección, y éste acceso puede incluir la modificación del valor apuntado. No obstante, también es posible declarar punteros que pueden acceder al valor apuntado para leerlo, pero no modificarlo. Para ello, es suficiente con asignar el tipo apuntado por el puntero como `const`. Ejemplo:

```
1 int x;
2 int y = 10;
3 const int * p = &y;
4 x = *p;          // ok: leyendo p
5 *p = x;             // error: modificando p, el cual es calificado como
constante
```

Aquí, `p` apunta a una variable, pero lo apunta en forma de asignación a `const`, queriendo decir que se puede leer el valor apuntado, pero no se puede modificar. Nótese también, que la expresión `&y` es del tipo `int*`, pero éste es asignado a un puntero del tipo `const int*` . Esto está permitido: un puntero a algo diferente a un `const` puede ser implicitamente convertido en un puntero a `const`. Aunque no al revés! Como medida de seguridad, punteros a `const` no pueden ser implicitamente transformados a punteros a algo diferente.

Uno de los casos más útiles de punteros a elementos `const` es la de a parametros de una función: una función que toma un puntero a algo diferente a `const` como parámetro puede modificar el valor que se pasa como argumento, mientras que una función que toma un puntero a `const` como parámetro no lo puede hacer.

Código fuente 31:

```
1 // punteros como argumentos:
2 #include <iostream>
3 using namespace std;
4
5 void increment_all (int* start, int* stop)
```

Salida

```
11
21
31
```

```
6  {
7    int * current = start;
8    while (current != stop) {
9      ++(*current);  // incremente el valor apuntado
10     ++current;     // incremente el pointer
11   }
12 }
13
14 void print_all (const int* start, const int* stop)
15 {
16   const int * current = start;
17   while (current != stop) {
18     cout << *current << '\n';
19     ++current;     // incremente el pointer
20   }
21 }
22
23 int main ()
24 {
25   int numbers[] = {10,20,30};
26   increment_all (numbers,numbers+3);
27   print_all (numbers,numbers+3);
28   return 0;
29 }
```

Nótese que `print_all` emplea punteros que apuntan a elementos constantes. Éstos punteros apuntan a contenidos constantes que no pueden ser modificados, auque no sean constantes en sí mimos: es decir, los punteros pueden ser incrementados o ser asignárseles una dirección diferente, auqnue no puedan modificar el contenido que apuntan.

Y es aquí donde una segunda dimensión se añade al tipo `const` con punteros: los punteros pueden también ser const. Y esto se especifica agregando const al tipo apuntado (después del asterisco):

```
int x;
      int *       p1 = &x;  // puntero no const a int no const
const int *       p2 = &x;  // puntero no const a int const
      int * const p3 = &x;  // puntero const a int no const
const int * const p4 = &x;  // puntero const a int const
```

La sintaxis con `const` y punteros es muy truculenta, y reconocer los casos que mejor se ajusta a cada instancia implica bastante experiencia. En todo caso, es importante mezclar constantes y punteros (y referencias) tan pronto como sea posible, aunque el programador no debe preocuparse mucho acerca de entenderlo todo si es la primera vez que se introduce en esta mezcla. Más ejemplos de cómo se debe emplear se presentarán en caítulos venideros.

Con el fin de añadir un poco más de confusión a la sintaxis de `const` con punteros, la asignación `const` puede tanto preceder como seguir al tipo de puntero, con exactamente el mismo significado:

```
const int * p2a = &x;  //  puntero no const a int const
int const * p2b = &x;  //también puntero no const a int const
```

Así como con los espacios que rodean el asterisco, la posición de `const` en este caso es parte del estilo. Este capítulo usa un prefijo `const`, y por razones históricas ésto parece ser más popular, aunque ambos son totalemente equivalentes. Las ventajas de cada estilo son aún intensamente debatidos en la Internet.

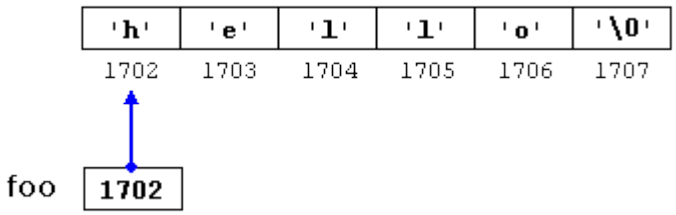
3.2.8 Punteros y cadenas de caracteres

Como se indicó anteriormente, las cadenas de caracteres son arreglos que contienen sucesiones de caracteres terminados en el caracter nulo. En secciones anteriores, cadenas de caracteres han sido empleadas para ser introducidas directamente en `cout`, también en la inicialización de cadenas y para inicializar arreglos de caracteres.

Sin embargo, éstas pueden también ser accedidas directamentes. Las cadenas de caracteres son arreglos del tipo de arreglo correcto para contener todos sus caracteres más el caracter nulo final, con cada uno de sus elementos siendo del tipo `const char` (como caracteres, no pueden ser modificados). Ejemplo:

```
const char * foo = "hello";
```

Se declara así un arreglo con la representación literal de "`hello`", y un puntero a su primer elemento es asignado a `foo`. Si imaginamos que "`hello`" es almacenado en las posiciones de memoria que comienzan con la dirección 1702, podríamos representar la declaración anterior por:



Nótese que aquí `foo` es un puntero que contiene el valor 1702, y ni 'h', ni "hello", aunque 1702 es ciertamente la dirección de ambos.

El puntero `foo` apunta a una serie de caracteres. Y debido a que los punteros y los arreglos se comportan escencialmente igual en expresiones, `foo` puede ser usado para acceder los caracteres de la misma manera que lo hacen los arreglos terminados con el caracter nulo. Ejemplo:

```
1 *(foo+4)
2 foo[4]
```

Ambas expresiones tienen el valor de 'o' (el quinto elemento del arreglo).

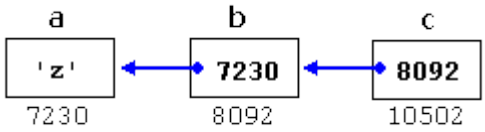
3.2.9 Punteros a punteros

C++ permite el uso de punteros que apuntan a punteros, los cuales, a su vez, apuntan a datos (o eventualmente otros punteros). La sintaxis simplemente requiere un asterisco (*) por cada nivel de

indirección en la declaración del puntero:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

Esto, asumiendo que las ubicaciones en memoria se escojen aleatoriamente para cada variable como 7230, 8092, y 10502, puede ser representado como:



Con el valor de cada variable representado dentro de su celda correspondiente, y su ubicación en memoria representada por el valor bajo la misma.

Lo novedoso de este ejemplo es la variable c, la cual es un puntero a otro puntero, y puede ser empleada en tres diferentes niveles de indirección que corresponden a diferentes valores:

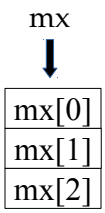
- `c` del tipo `char**` y tiene un valor de 8092
- `*c` es del tipo `char*` con valor de 7230
- `**c` es del tipo `char` y su valor es 'z'

3.2.9.1 Punteros Multiples

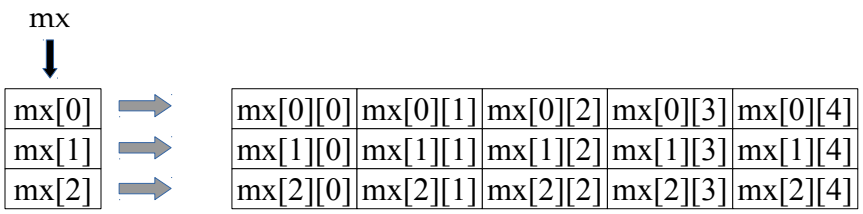
Arreglos bidimensionales y matrices pueden ser accedidos a través de punteros dobles (un puntero a un puntero es llamado un puntero doble), por ejemplo:

```
int **mx; //puntero doble: puntero a puntero
mx=new int*[n]; //nuevo espacio para almacenar n punteros tipo int
                //mx apunta al elemento inicial mx[0]
for (int i=0; i<n; i++)
mx[i]=new int[m]; //crea m objetos para cada uno de los n punteros
                  //mx[i] apunta al elemento inicial mx[i][0]
```

En el ejemplo la primera declaración declara `mx` para ser un puntero a un puntero, esto es un puntero doble. La segunda declaración aloca `n` objetos del tipo `int*` y asigna la dirección del elemento inicial de `mx`. Los `n` objetos por tanto son punteros tipo `int`. Ahora mx tiene el valor `&mx[0]`. De manera pictórica para `n=3` se tiene:



La tercera declaración, dentro del loop for, aloca `m` objetos de tipo int para cada uno de los `n` punteros `mx[i], i=0, 1, ...., n-1`. Una representación gráfica del puntero doble `mx` en el caso de `n=3` y `m=5` puede ser dibujada de la siguiente manera:



El proceso anterior puede también ser entendido usando la palabra clave o reservada *typedef* (ver Taller 9). Una declaración con el prefijo *typedef* introduce un nuevo nombre a un tipo de dato, por ejemplo:

```
1 typedef int* intptr; //intptr es sinónimo de int*
```

La declaración introduce `intptr` como un sinónimo de `int*`. Typedef no introduce un nuevo tipo de dato, lo que hace es que `intptr` significa lo mismo que `int*`. Entonces el doble puntero `mx` puede ser declarado de manera equivalente como

```
1 intptr *mx=new intptr [n];
2 for (int i=0; i<n; i++)
3   mx[i]=new int[m];
```

Lo cual indica que `mx` es un puntero a `intptr`, y `n` objetos: `mx[i], i=0,1,..., n-1`, del tipo `intptr` son creados usando el operador `new`. Entonces `m` objetos del tipo `int` son creados para cada `mx[i]`.

Y ahora `mx` puede ser usado de la misma manera que un arreglo bidimensional:

```
1 for (int i=0; i<n; i++) //mx[i] es un puntero a int
2   for (int j=0; j<m; j++) //usado como un arreglo 2D
3     mx[i][j]=i*i+9;  //acceso a los elementos mx[i][j]
```

Luego de usado, el espacio puede ser liberado:

```
1 for (int i=0; i<n; i++){ //liberar espacio después del uso
2   delete[] mx[i]; } //primer se liberan los n punteros
3 delete[] mx  ;  //luego se libera los punteros dobles mx
```

Las dimensiones de n y m pueden ser computadas mientras el programa corre y el espacio para mx puede ser alocado dinámicamente. Esto a diferencia de un arreglo el cual no puede ser declarado a menos que su dimensión se conozca antes de ser compilado el programa. Nota: las ordenes de `allocate` y `delete` pueden ser usadas con los punteros dobles y sus definiciones son opuestas entre si.

3.2.10 Punteros void

El puntero del tipo `void` es un tipo especial de puntero. En C++, void representa la ausencia de tipo. Por consiguiente, los punteros void son punteros que apuntan aun valor sin tipo (y entonces tambien son indeterminadas la longitud y sus propiedades de desreferencia).

Esto permite a los punteros `void` gran flexibilidad, permitiéndoles apuntar a cualquier tipo de dato, desde un valor entero hasta un float o a una cadena de caracteres. Sin emabargo, tienen una gran limitante: los datos que apuntan no pueden ser desreferenciados directamente (lo cual es lógico, ya que no se tiene el tipo al cual desreferenciar), y por esta razón, cualquier dirección en un puntero void necesita ser transformada a otro tipo de puntero que apunte a un dato concreto antes de ser desreferenciada.

Uno de los posibles usus puede ser el de pasar parámetros genéricos a una función. Ejemplo:

Código fuente 32:

```
1  // ejemplo: el aumentador
2  #include <iostream>
3  using namespace std;
4
5  void increase (void* data, int psize)
6  {
7    if ( psize == sizeof(char) )
8    { char* pchar; pchar=(char*)data; ++(*pchar); }
9    else if (psize == sizeof(int) )
10   { int* pint; pint=(int*)data; ++(*pint); }
11 }
12
13 int main ()
14 {
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << '\n';
20   return 0;
21 }
```

Salida:

```
y, 1603
```

`sizeof` es un operador integrado al lenguage C++ que devuelve el tamaño en bytes de su argumento. Para tipos de datos no dinámicos, éste valor es constante. De manera que, por ejemplo, `sizeof(char)` es 1, debido a que char es siempre de tamaño un byte.

3.2.11 Punteros no válidos y punteros nulos

En principio, los punteros deben apuntar a direcciones válidas, tales como las direcciones de una variable o de un elemento de un arreglo. No obstante, los punteros pueden apuntar a cualquier dirección, incluidas las direcciones que no hacen referencia a ningún elemento válido. Ejemplos típicos de estao son los *punteros no inicializados* y punteros a elementos inexistentes de un arreglo:

```
1 int * p;                // puntero no inicializado(variable local)
2
3 int myarray[10];
4 int * q = myarray+20;  // elemento fuera del arreglo
```

Ni `p` ni `q` apuntan a direcciones que contengan un valor conocido, sin embargo, ninguna de las declaraciones anteriores causa error. En C++, se le permite a los punteros tomar el valor de cualquier

dirección, sin importar si hay verdaderamente en esa dirección o no. Lo que puede generar un error es hacer desreferencia a dicho puntero (es decir, hacer un llamado al valor apuntado). Accesar ese puntero causa un comportamiento indefinido, que puede ir desde un error en tiempo de ejecución hasta el acceso a un valor aleatorio.

Aún así, a veces es necesario que un puntero no apunte a ningún lado, sin que esto implique una dirección inválida. Para estos casos, existe un valor especial que puede ser tomado por cualquier tipo de puntero: el *valor de puntero nulo*. Este valor puede ser expresado en C++ de dos formas: o bien con el valor entero de cero, ó con la palabra reservada `nullptr`:

```
int * p = 0;
int * q = nullptr;
```

En este ejemplo, tanto p como q son *punteros nulos*, queriendo esto decir que ellos no apuntan a ninguna dirección específica, y pueden ser comparados como iguales: todos los *punteros nulos* se comaran iguales a otros *punteros nulos*. Tambien es frecuente encontrar la constante de valor definido `NULL` siendo usada como referencia al *puntero nulo* en código antiguo:

```
int * r = NULL;
```

`NULL` está definido en una variedad de encabezados de las librerías estándar, y es definido como un sobrenombre de algún valor constante de *puntero nulo* (tal como 0 o `nullptr`).

Jamás confunda los punteros nulos con los punteros void! Un *puntero nulo* es un valor que cualquier puntero puede tomar y representa no estar apuntando a una dirección específica, mientras que un puntero void es un tipo de puntero que puede apuntar a cualquier dirección sin tipo específico. Uno se refiere al valor almacenado en el puntero, mientras que el otro al tipo de datos a los que apunta.

3.2.12 Punteros a funciones

C++ permite operaciones con punteros a funciones. El epleo típico de esto es pasar a función como argumento de otra función. Punteros a funciones son declarados con la misma sintaxis que la declaración de una función convencional, excepto que el nombre de la función esta entre paréntesis () y un asterisco (*) es añadido antes del nombre:

Código fuente 33:

```
// puntero a funciones
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
```

Salida:

```
8
```

```
14   g = (*functocall)(x,y);
15   return (g);
16 }
17
18 int main ()
19 {
20   int m,n;
21   int (*minus)(int,int) = subtraction;
22
23   m = operation (7, 5, addition);
24   n = operation (20, m, minus);
25   cout <<n;
26   return 0;
27 }
```

En el ejemplo anterior, `minus` es un puntero a una función que tiene dos parámetros del tipo int. Está directamente inicializado para apuntar a la funcion `subtraction`:

```
int (* minus)(int,int) = subtraction;
```

3.3 Memoria Dinámica

Se ha visto hasta el momento, que toda la memoria se ha determinado antes de la ejecución del programa definiendo las variables usadas. Sin embargo, existen casos donde la memoria se puede determinar mientras éste se ejecuta. Por ejemplo, cuando la memoria depende de la entrada del usuario, en éste caso el programa necesita alocar memoria de manera dinámica, y para ésto el lenguaje C++ tiene los operadores `new` y `delete`.

3.3.1 Operadores `new` y `new []`

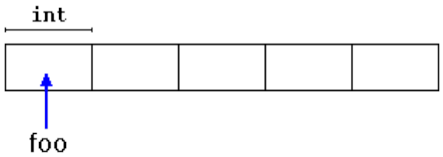
La memoria dinámica es alocada usando el operador new, el cual es seguido por el tipo de dato especificado y si se requiere un conjunto de elementos, la cantidad se encuentran entre paréntesis cuadrados [ ]. Ésto retorna un puntero al inicio del nuevo bloque de memoria. Se usa la sintaxis:

```
puntero = new tipo_de_dato
puntero = new tipo_de_dato [numero_de_elementos]
```

La primera expresión se usa para alocar memoria para contener un único elemento. La segunda se usa para alocar un bloque (o arreglo) de elementos, donde el numero_de_elementos es un entero que representa la cantidad de éstos. Por ejemplo:

```
1 int * foo;
2 foo = new int [5];
```

En éste caso, el sistema dinámicamente separa el espacio para 5 elementos de tipo entero y retorna un puntero al primer elemento de la secuencia, todo es asignado al puntero `foo`.



Recuerde que `foo` es un puntero y por lo tanto se puede acceder a su primer elemento bien sea con la expresión `foo[0]` o con la expresión `*foo` (ambas son equivalentes). El segundo elementos puede ser accedido bien sea con `foo[1]` o con `*(foo+1)` y así sucesivamente.

Hay una diferencia substancial entre la declaración normal de un arreglo y la alocación de memoria dinámica para un bloque de memoria usando new. La diferencia es que el tamaño de un arreglo necesita ser una *expresión constante*, y por lo tanto su tamaño debe ser determinado en el diseño del programa, antes de que éste sea ejecutado, mientras que la alocación de momoria dinámica usando new permite asignar memoria durante la ejecución usando cualquier tamaño.

La memoria dinámica requerida por el programa es alocada de la memoria de la máquina, la cual es

limitada y naturalmente se puede acabar, así que no se garantiza que todo requerimiento de alocación de memoria usando el operador `new` vaya a ser dado por la máquina.

C++ provee dos mecanimos estándar para verificar si la alocación ha sido exitosa:

1) Usando excepciones: usando este método, una excepción del tipo `bad_alloc` es devuelta cuando la alocación falla. Las excepciones son una herramienta poderosa en C++ pero por ahora es suficiente con saber que si la excepción es devuelta porque no se puede manipular la cantidad de memoria solicitada y el programa termina la ejecución del programa. Este método de excepción es el utilizado por defecto por el operador `new` en una declaración del tipo:

```
foo = new int [5];  // si la alocación falla, una excepción se devuelve
```

2) El otro método es conocido como `nothrow` que significa no devuelto, y lo que sucede cuando es usado es que si la alocación de memoria falla, en lugar de devolver una excepción `bad_alloc` o terminar el programa, el puntero devuelto por `new` es un puntero nulo, y el programa continúa su ejecución normalmente. Éste método se especifica usando un objeto especial llamado `nothrow`, declarado en el encabezado `<new>` y en el argumento por new:

```
foo = new (nothrow) int [5];
```

En este caso, si la alocación de el bloque de memoria falla, la falla puede ser detectada verificando si el puntero foo es un puntero nulo.

```
1 int * foo;
2 foo = new (nothrow) int [5];
3 if (foo == nullptr) {
4   // error asignando memoria. Tomar medidas al respecto.
5 }
```

Este método `nothrow` produce códigos ménos eficientes que el de las excepciones, ya que esto implica le implica estar verificando los valores retornados luego de cada alocación. Por tanto el mecanismo de excepción es generalmente preferido, así muchos ejemplos utilicen el mecanismo `nothrow` por su simplicidad.

3.3.2 Operadores `delete` y `delete[]`

En la mayoría de los casos la memoria alocada dinámicamente se necesita durante periodos específicos de tiempo dentro de un programa y luego ya no se necesita más. Ésta memoria se puede

liberar de tal manera que la memoria esté disponible nuevamente para cualquier otro requisito de memoria dinámica. Para tal fin es el operador `delete`, cuya sintaxis es:

```
delete puntero;
delete[] puntero;
```

La primera declaración borra de la memoria un elemento simple alocado usando `new` y el segundo borra la memoria alocada para arreglos de elementos usando `new[]` donde el tamaño está en los braquetes cuadrados ([ ]).

Los valores pasados como argumento a `delete` deben ser o bien sea un puntero o un bloque de memoria previamente alocado con `new`, o un puntero nulo (en el caso de un puntero nulo, `delete` no produce efecto).

Código fuente 35:

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "Cuantos numeros quiere escribir? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memoria no puede ser alocada";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Entre un numero: ";
      cin >> p[n];
    }
    cout << "Usted ha escrito: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
Cuantos numeros quiere escribir? 5
Entre un numero : 75
Entre un numero : 436
Entre un numero : 1067
Entre un numero : 8
Entre un numero : 32
Usted ha escrito: 75, 436, 1067, 8,32,
```

Note que el valor entre el paréntesis cuadrado del operador `new` es el valor de una variable entrada por el usuario (`i`) y no una expresión constante.

```
p= new (nothrow) int[i];
```

Existe la posibilidad que el usuario introduzca un valor muy grande para i y por tanto el sistema no puede alocar suficiente memoria para ello. Por ejemplo, si se responde que se desea escribir un billon de números, la máquina no puede alocar toda esta memoria y por lo tanto aparece el mensaje que se preparó para éste caso (`Error: memoria no puede ser alocada`)

Es considerada una buena práctica, siempre proporcionar mecanismos para manipular errores en la alocación de memoria, bien sea verificando el valor del puntero (`if nothrow`) o usando una expresión adecuada.

3.3.3 Memoria dinámica en C

C++ tiene los operadores `new` y `delete` para alocar la memoria dinámicamente. Pero estos no estaban disponibles en el lenguaje C; en lugar de ellos se utilizaba la libreria `<cstdlib>` que se definía en el encabezado (conocida como `<stdlib.h>` en C) la cual tiene las funciones `malloc, calloc, realloc` y `free.` . Estas funciones son también usadas para alocar y desalojar memoria dinámicamente.

Note que los bloques de memoria alocados por estas funciones de C, NO son necesariamente compatibles con aquellos retornados por `new`, así que ellos no deben ser mezclados, cada uno debe ser manejado con su propio grupo de funciones u operadores.

3.3.4. Sobrenombre "alias" de tipos de datos (typedef / using)

3.3.4.1 Typedef

Un sobrenombre o álias de un tipo de dato es un nombre diferente por el cual se puede identificar un cierto tipo de dato. En C++, cualquier tipo de dato válido puede ser cambiado por un sobrenombre En C++, hay dos sintaxis para crear estos sobrenombres: El primero es heredado por el lenguaje C, en el que usa la palabra clave o reservada `typedef`:

```
typedef tipo_existente nuevo_nombre_tipo ;
```

donde `tipo_existente` es cualquier tipo de dato, bien sea fundamental o compuesto y `nuevo_nombre_tipo` es un identificador con el nuevo nombre que se le da al tipo de dato.

Por ejemplo:

```
1 typedef char C;
2 typedef unsigned int palabra;
3 typedef char * pChar;
4 typedef char campo [50];
```

El ejemplo define cuatro tipos de sobrenombres: `C, palabra, pChar y campo,` como `char, unsigned int, char*, y char[50]` respectivamente. Una vez estos sobrenombres o alias son definidos, se pueden usar en cualquier declaración como cualquier otro tipo de dato válido:

```
1 C mychar, anotherchar, *ptc1;
2 palabra myword;
3 pChar ptc2;
4 campo name;
```

3.3.4.2 Using

Más recientemente, una segunda sintaxis para definir sobrenombre de tipos de datos fue introducida en el leguaje C++:

```
using nuevo_nombre_tipo = tipo_existente ;
```

Por ejemplo, los mismos sobrenombres de los tipos de datos anteriores se pueden definir como:

```
1 using C = char;
2 using palabra = unsigned int;
3 using pChar = char *;
4 using campo = char [50];
```

Ambos sobrenombres, los definidos con `typedef` y los definidos con `using` son semánticamente equivalentes. La única diferencia es que `typedef` tiene ciertas limitaciones en el ámbito de las plantillas o templates (tema que aún no se ha visto) que `using` no tiene. De esta manera, `using` es más genérico, aunque `typedef` tiene mayor historia y por lo tanto es más común verlo en los códigos.

Note que ni `typedef` ni `using` crean nuevos tipos de datos. Ellos solo crean sinónimos de los tipos existentes.

Los sobrenombres de los tipos de datos se usan para reducir nombres de tipos de datos muy largos o confusos, y son como herramientas, por ejemplo si usa un sobrenombre para referirse al tipo de dato `int` en lugar de usar `int` directamente, le permite remplazar fácilmente éste por un `long` en una versión posterior de su programa sin tener que cambiar el tipo `int` en cada línea donde lo usó en el programa.

3.3.5. Estructuras de Datos

Una estructura de datos es un grupo de elementos de datos agrupados bajo un sólo nombre. Éstos elementos son conocidos como *miembros*, y pueden tener diferentes tipos de datos y diferentes longitudes. La estructura de datos pueden ser declarados en C++ usando la siguiente sintaxis:

```
struct nombre_tipo {
miembro_tipo1 miembro_nombre1;
miembro_tipo2 miembro_nombre2;
miembro_tipo3 miembro_nombre3;
.
.
} nombres_objetos;
```

Donde `nombre_tipo` es el nombre del tipo de estructura `nombre_objetos` puede ser un conjunto de nombres o identificadores válidos para objetos que tienen el tipo de esta estructura, dentro de los corchetes `{ }`, hay una lista de miembros, cada uno de los cuales está especificado con un tipo de dato y un nombre o identificador válidos. Por ejemplo:

```
1 struct producto {
2   int peso;
3   double precio;
4 } ;
5
6 producto manzana;
7 producto banana, melon;
```

Este código declara un tipo de estructura llamada `producto` y es definida con dos miembros: `peso` y `precio` , cada uno con un tipo de dato diferente. Esta declaración crea un nuevo tipo de dato

`producto`) el cual es usado para declarar tres nuevos objetos o variables: manzana, banana, y melon. Note que una vez `producto` es declarado, se puede usar como cualquier otro tipo de dato.

Justo después del corchete de cierre de la definición de la estructura se finaliza con un punto y coma (`;`), el campo `nombres_objetos` es opcional, y puede ser usado para declarar objetos del tipo de la estructura directamente. El ejemplo anterior quedaría:

```
struct producto {
  int peso;
  double precio;
} manzana,banana,melon;
```

Es importante difereciar entre el nombre del tipo de la estructura (`producto`) y un objeto de este tipo de dato (`manzana, banana, melon`). Se pueden declarar muchos objetos en un sólo tipo de dato dado por una estructura.

Una vez los objetos de un determinado tipo de estructura son declarados sus miembros se pueden acceder directamente, La sintaxis para tal fin es colocar un punto (.) entre el nombre del objeto y el nombre del miembro. Por ejemplo, podemos operar con cualquiera de éstos elementos como si ellos fueran variables estándar de sus respectivos tipos de datos:

```
manzana.peso
Manzana.precio
banana.peso
banana.precio
melon.peso
melon.precio
```

Cada uno de estos elementos tiene un tipo de dato correspondiente al miembro que ellos se refieren: `manzana.peso, banana.peso, melon.peso` son del tipo `int` , mientras que `manzana.precio, banana.precio, melon.precio` son del tipo `double`.

Miremos un ejemplo completo: Código fuente 3

```
// ejemplo de estructuras

#include <iostream>
#include <string>
#include <sstream>

using namespace std;

struct peliculas_t {
  string titulo;
  int anyo;
} mia, suya;

void imprimirpelicula (peliculas_t
pelicula);

int main ()
{
  string mystr;

```

```
Entre titulo: Alien
Entre anyo: 1979

Mi pelicula
favorita es: 2001 A
Space Odyssey
(1968)
Y la suya es: Alien
(1979)
```

```
  mia.titulo = "2001 A Space Odyssey";
  mia.anyo = 1968;

  cout << "Entre titulo: ";
  getline (cin,suya.titulo);
  cout << "Entre anyo: ";
  getline (cin,mystr);
  stringstream(mystr) >> suya.anyo;

  cout << "Mi pelicula favorita es:\n";
  imprimirpelicula (mia);
  cout << "Y la suya es:\n ";
  imprimirpelicula (suya);

  return 0;
}
void imprimirpelicula (peliculas_t
pelicula)
{
  cout << pelicula.titulo;
  cout << " (" << pelicula.anyo << ")\n";
}
```

El ejemplo muestra como el miembro de un objeto actúa justo como una variable regular. Por ejemplo el miembro `suya.anyo` es una variable válida tipo `int`, y `mia.titulo` es una variable válida del tipo `string`.

Y los objetos `mia` y `suya` son también variables válidas con el tipo `peliculas_t`. Por ejemplo, ambos objetos han pasado a la función `imprimirpelicula` como simples variables. De las características principales de las estructuras es la habilidad de referirse bien sea a los miembros individualmente o a la estructura entera como un todo. En ambos casos se usa el mismo identificador : el nombre de la estructura.

Ya que las estructuras son tipos de datos, también pueden ser usadas como tipos de dato para arreglos para contruir tablas o bases de datos con ellos.

Código fuente 37:

```
// arreglos de estructuras

#include <iostream>
#include <string>
#include <sstream>

using namespace std;

struct peliculas_t {
  string titulo;
  int anyo;
} films [3];
```

```
Entre titulo: Alien
Entre anyo: 1979

Mi pelicula
favorita es: 2001 A
Space Odyssey
(1968)
Y la suya es: Alien
(1979)
```

```
14
15
16 void imprimirpelicula (peliculas_t
17 pelicula);
18
19 int main ()
20 {
21   string mystr;
22   int n;
23
24   for (n=0; n<3; n++)
25   {
26     cout << "Entre titulo: ";
27     getline (cin,films[n].titulo);
28     cout << "Entre anyo: ";
29     getline (cin,mystr);
30     stringstream(mystr) >> films[n].anyo;
31   }
32   cout << "\n Usted ha escrito estas
33 peliculas:\n";
34
35   for (n=0; n<3; n++)
36     imprimirpelicula (films[n]);
37
38   return 0;
39 }
40 void imprimirpelicula (peliculas_t
41 pelicula)
42 {
43   cout << pelicula.titulo;
44   cout << " (" << pelicula.anyo << ")\n";
45 }
```

3.3.5.1 Punteros a estructuras

Como cualquier otro tipo de dato, las estructuras pueden ser apuntadas por su propio tipo de punteros.

```
1 struct peliculas_t {
2   string titulo;
3   int anyo;
4 };
5
6 peliculas_t apelicula;
7 peliculas_t * ppelicula;
```

Aquí `apelicula` es un objeto del tipo estructural `peliculas_t`, y `ppelicula` es un puntero que apunta a los objetos del tipo estrucutral `peliculas_t`. De aquí que el siguiente código es válido:

```
ppelicula = &apelicula;
```

Al puntero `ppelicula` se le asigna a la dirección del objeto `apelicula`.

El siguiente ejemplo mezcla punteros y estructuras, y nos ayuda a introducir el operador flecha (`->`)

Código fuente 38:

```
// punteros a estructuras
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct peliculas_t {
  string titulo;
  int anyo;
};

int main ()
{
  string mystr;

  peliculas_t apelicula;
  peliculas_t * ppelicula;
  ppelicula = &apelicula;

  cout << "Entre titulo: ";
  getline (cin, ppelicula->titulo);
  cout << "Entre anyo: ";
  getline (cin, mystr);
  (stringstream) mystr >> ppelicula->year;

  cout << "\nUsted ha escrito:\n";
  cout << ppelicula->titulo;
  cout << " (" << ppelicula->anyo <<
")\n";

  return 0;
}
```

```
Entre titulo: Invasion of the body
snatchers
Entre anyo: 1978

Usted ha escrito:
Invasion of the body snatchers (1978)
```

El operador flecha (->) es un operador de desreferencia que es usa exclusivamente con punteros hacia objetos que tienen miembros. Este operador sirve para acceder al miembro de un objeto directamente desde su dirección. En el ejemplo anterior:

```
ppelicula->titulo
```

Esta declaración es SIEMPRE equivalente a:

```
(*ppelicula).titulo
```

Ambas expresiones, `ppelicula->titulo` y `ppelicula->titulo` son válidas y ambas acceden al miembro `titulo` de la estructura de datos apuntada por el puntero llamado `ppelicula.` Y es definitivamente diferente a:

```
*ppelicula.titulo
```

Lo cual es equivalente a:

```
*(ppelicula.titulo)
```

Esto último accedería al valor apuntado por un hipotetico puntero miembro llamado `titulo` del objeto estructural `ppelicula` (el cual NO es el caso, ya que `titulo` no es tipo puntero). La siguiente

tabla resume las posibles combinaciones de los operadores para los punteros y para los miembros de las estructuras:

Expresión	Lo que evalúa	Equivalente
`a.b`	Miembro `b` del objeto `a`	
`a->b`	Miembro `b` del objeto apuntado por `a`	`(*a).b`
`*a.b`	Valor apuntado por un miembro `b` del objeto `a`	`*(a.b)`

5.2 Anidando estructuras

Las estructuras pueden también anidarse de tal manera que un elemento de una estructura es llamado por otra esctructura:

```
struct peliculas_t {
  string titulo;
  int anyo;
};

struct friends_t {
  string nombre;
  string email;
  peliculas_t pelicula_favorita;
} charlie, maria;

friends_t * pfriends = &charlie;
```

Después de las declaraciones anteriores, todas las expresiones siguientes son válidas:

```
charlie.nombre
maria.pelicula_favorita.titulo
charlie.pelicula_favorita.anyo
pfriends->pelicula_favorita.anyo
```

Donde las últimas dos expresiones refieren al mismo miembro.

En lo siguiente, encontrará tres ejemplos usando estructuras para una mejor comprensión.
Código fuente 39:

```
// ejemplo de estructura
#include <iostream>
#include <string>
using namespace std;

struct datoPersonal{
  string nombre;
  char inicial;
  int edad;
  double altura;
};


int main ()
{
  datosPersonal persona;
```

```
17
18    persona.nombre="Juan";
19    persona.inicial='J';
20    persona.edad=20;
21    persona.altura=1.50;
22
23    cout<<"La edad de "<<persona.nombre<<" es "<< persona.edad<<endl;
24
25    return 0;
26  }
```

Código fuente 40:

```
1   // ejemplo de punteros tipo estructural
2   #include <iostream>
3   #include <string>
4   using namespace std;
5
6   struct datoPersonal{
7     string nombre;
8     char inicial;
9     int edad;
10    double altura;
11  };
12
13  int main ()
14  {
15    datosPersonal *persona = new datosPersonal[4];
16
17    for(int i=0; i<4; i++){
18    cout<<"Escriba el nombre de la persona "<< i << endl;
19    cin>>persona[i].nombre;
20    }
21    cout<<"La tercera persona es "<< persona[2].nombre << endl;
22
23
24    return 0;
25  }
```

Código fuente 41:

```
1   // ejemplo de anidación de estructuras
2   #include <iostream>
3   #include <string>
4   using namespace std;
5
6   struct datosNacimiento{
7     int dia;
8     int mes;
9     int anyo;
10  };
11
12  struct datosPersonal{
13    string nombre;
14    char inicial;
15    int edad;
```

```
16   double altura;
17   datosNacimiento fechaNacimiento;
18 };
19
20 int main (){
21
22   datosPersonal persona;
23
24   persona.nombre="Juan";
25   persona.inicial='J';
26   persona.edad=20;
27   persona.altura=1.50;
28   persona.fechaNacimiento.mes=8;
29
30   cout<<"La edad de "<<persona.nombre<<" es "<< persona.edad;
31    cout<<" años y nació en el mes "<<persona.fechaNacimiento.mes<<" del
32 año.";
33   cout<<endl;
34
35   return 0;
36 }
```

3.4. Clases

Clases son un concepto expandido de las estructuras de datos, `structur`: como las estructuras, las clases pueden contener miembros, pero también puede contener funciones como miembros.

Un objeto es una instancia de una clase. En términos de variables, una clase sería el tipo de dato y un objeto sería la variable.

Las clases son definidas usando la palabra `class`, con la siguiente sintaxis:

```
class nombre_clase {
  especificaciones_acceso1:
    miembro1;
  especificaciones_acceso2:
    miembro2;
  ...
} nombre_objeto;
```

Donde `nombre_clase` debe ser un identificador válido para la clase, el `nombre_objeto` es una lista opcional de nombres para los objetos de dicha clase. El cuerpo de la declaración puede contener miembros, el cual puede ser bien sea declaración de variables o de funciones y opcionalmente *especificaciones de acceso* (`especificaciones_acceso`).

Las clases tienen el mismo formato que una estructura, excepto que ellas pueden también incluir funciones y tienen como novedad las llamadas especificaciones de acceso.

Un especificador de acceso es uno de las siguientes tres palabras reservadas: `private`, `public` o `protected`. Estas modifican los derechos de acceso para los miembros que los siguien de la siguiente manera:

`private`: son los miembros de la clase que son accesibles sólo desde otros miembros de la misma clase (o desde alguna clase amiga "*friends*").

`protected`: son los miembros de la clase que son accesibles desde otro miembros de la misma clase (o desde una clase amiga "*friends*"), pero también desde miembros o sus clases derivadas.

`public`: son los miembros de la clase que son accesibles desde cualquier parte donde el objeto sea visible.

Por defecto, todos los miembros de una clase declarada con la palabra reservada `class` tienen acceso privado para todos sus miembros. Por tanto, todo miembro que es declarado antes que cualquier especificador de acceso tiene acceso privado atomáticamente. Por ejemplo:

```
class Rectangulo {
    int ancho, alto;
  public:
    void valores (int,int);
    int area (void);
} rec;
```

El código declara una `class` (este es el tipo de dato) llamada `Rectangulo` y un objeto (es decir variable) de esta clase llamado `rec`. Esta clase contiene cuatro miembros: dos miembros con tipo de dato `int` (el miembro `ancho` y `alto`) con *acceso privado* (porque por defecto tienen este nivel de acceso) y otros miembros que son funciones con *acceso público* (porque están escritas bajo la palabra reservada **`public`** y por tanto tienen este nivel de acceso). De las funciones `valores` y `area`, por ahora sólo se ha incluido su declaración pero no su definición.

Luego de las declaraciones de `Rectangulo` y `rec` cualquier miembro público del objeto `rec` puede ser accesado como si fueran funciones o variables normales, usando unicamente el punto (`.`) entre los *nombre de los objetos* y los *nombres de los miembros*. Esto sigue la misma sintaxis que se utiliza para accesar los miembros de las estructuras, por ejemplo:

```
rec.valores (3,4);
miarea = rec.area();
```

Los únicos miembros de `rec` que no pueden ser accedidos desde afuera de la clase son `ancho` y `alto` , ya que ellos tienen acceso privado y ellos sólo pueden ser referenciados desde otro miembro desde de la misma clase. Aquí tenemos un ejemplo completo de la clase `Rectangulo` :

Código fuente 43:

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, alto;
  public:
    void valores (int,int);
    int area() {return ancho*alto;}
};

void Rectangulo::valores (int x, int y) {
```

```
area: 12
```

```
ancho = x;
  alto = y;
}

int main () {
  Rectangulo rec;
  rec.valores (3,4);
  cout << "area: " << rec.area();
  return 0;
}
```

Este ejemplo reintroduce el operador (`::` , doble dos puntos), visto anteriormente en relación a los namespaces. Aquí es usado en la definición de la función `valores` para definir un miembro de una clase fuera de la clase como tal.

Note que la definición del función miembro `area` ha sido incluida directamente en la definición de la clase `Rectangulo` debido a su simpleza. Mientras que la función `valores` es solamente declarada con su prototipo dentro de la clase y su definición se encuentra fuera de la clase y el operador (`::`) es usado para especificar que la función es un miembro de la clase `Rectangulo` y de una función regular sin membresía.

3.4.1. Constructores

Que sucedería en el ejemplo anterio si la función miembro `area` fuera llamada antes que `valores`? Se tendría una indeterminación como resultado, ya que los miembros `ancho` y `alto` nunca han tenido asignados unos valores.

Para evitar esto, una clase puede incluir un tipo especial de función llamada su *constructor*, el cual es automáticamente llamada siempre que un nuevo objeto de la clase es creado, permitiendo a la clase inicializar variables miembros o alocar memoria.

La función del constructor es declarado justo como cualquier función mimbro pero con el MISMO nombres de la clase y SIN ningún tipo de dato que retornar, ni siquiera `void`.

La clase `Rectangulo` puede ser mejorada implementando un constructor de la siguiente manera:

Código fuente 44:

```
// ejemplo: constructor
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, alto;
  public:
    Rectangulo (int,int);
    int area () {return ancho*alto;}
};

Rectangulo::Rectangulo (int a, int b) {
  ancho = a;
  alto = b;
```

```
rect area: 12
rectb area: 30
```

```
}

int main () {
  Rectangulo rec (3,4);
  Rectangulo recb (5,6);
  cout << "rec area: " << rec.area() << endl;
  cout << "recb area: " << recb.area() << endl;
  return 0;
}
```

Los resultados de este ejemplo son exactamente igual que antes, pero ahora la clase `Rectangulo` no tiene la función miembro `valores` y en su lugar tiene un constructor que realiza la misma acción: inicializa los valores de `ancho` y `alto` con los argumentos que le son pasados. Note que ahora estos argumentos son pasados al constructor al momento en que los objetos son creados:

```
Rectangulo rec (3,4);
Rectangulo recb (5,6);
```

Los constructores no pueden ser llamados explicitamente como si ellos fueran funciones miembro regulares. Pueden ser unicamente ejecutados una vez, cuando un nuevo objeto de la clase es creado.

Note como ni la declaración del prototipo de constructor (dentro de la clase) ni la definición posterior del constructor tienen valores que retornar, ni siquiera son `void`. Los constructores nunca retornan valores, ellos unicamamente inicializan los objetos.

3.4.2. Sobre definición de constructores

Como cualquier otra función, un constructor puede ser sobredefinido con diferentes versiones que tienen diferentes parámetros: un un número diferente de parámetros y/o con parametros de diferentes tipos de datos. El compilador automáticamente llamará al constructor cuyos parámetros sean idénticos a los argumentos que se dan:

Código fuente 45:

```
//  sobredefición del constructor de la clase
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, alto;
  public:
    Rectangulo ();
    Rectangulo (int,int);
    int area (void) {return ancho*alto;}
};

Rectangulo::Rectangulo () {
  ancho = 5;
  alto = 5;
}

Rectangulo::Rectangulo (int a, int b) {
  ancho = a;
```

```
rect area: 12
rectb area: 25
```

```
20   alto = b;
21 }
22
23 int main () {
24   Rectangulo rec (3,4);
25   Rectangulo recb;
26   cout << "rec area: " << rec.area() << endl;
27   cout << "recb area: " << recb.area() << endl;
28   return 0;
29 }
```

En el ejemplo, dos objetos de la clase `Rectangulo` son contruidos: `rec` y `recb`.

El objeto `rec` es construido con dos argumentos como se había hecho anteriormente.

Pero este ejemplo introduce un constructor especial: *el constructor por defecto*. El constructor de defecto es un constructor que no tiene parámetros, y es un caso especial porque es llamado cuando un objeto es declarado pero no inicializado con ningún argumento. En el ejemplo anterior, el constructor por defecto es llamado por `recb`. Note que `recb` es construido sin tener incluso los paréntesis vacíos, de hecho los paréntesis vacíos no pueden ser usados para llamar un *constructor por defecto*:

```
Rectangulo recb;   // ok, constructor defecto llamado
Rectangulo recc(); // oops, constructor defecto NO llamado
```

Esto es justo porque el paréntesis haría de `recb` una declaración de una función en lugar de una declaración de un objeto: Sería una función que no toma argumentos y retorna un valor del tipo `Rectangle`.

3.4.3. Inicialización uniforme

La manera como se han llamado los constructores, colocando los argumentos en paréntesis, como se hizo anteriormente, es conocida como *forma funcional*. Pero los constructores pueden ser llamados usando otra sintaxis también.

Primero, constructores que tienen un único parámetro pueden ser llamados usando la sintaxis de inicialización de variables, usando el signo igual (=):

```
nombre_clase nombre_objeto = valor_inicializacion;
```

De manera reciente en C++ se ha introducido la posibilidad que los constructores puedan ser llamados usando una *inicialización uniforme*, lo cual esencialmente es lo msimo que la forma funcional pero en lugar de usar paréntesis redondos (`()`) se usan corchetes (`{}`):

```
nombre_clase nombre_objeto {valor1, valor 2, valor3, ...};
```

De manera opcional, esta sintaxis admite el signo igual (=) antes de los corchetes:

```
nombre_clase nombre_objeto = {valor1, valor 2, valor3, ...};
```

Miremos un ejemplo usando cuatro (4) maneras de construir objetos de una misma clase cuyo constructor toma un único parámetro:

Código fuente 46:

```
1  //  clases e inicialización uniforme
2  #include <iostream>
3  using namespace std;
4  #define PI 3.14159265358979323846
5
6  class Circulo {
7      double radio;
8    public:
9      Circulo (double r){radio=r;};
10     double circunf () {return 2*radio*PI;}
11 };
12
13 int main () {
14   Circulo cir1 (10.0); //forma funcional
15   Circulo cir2 = 20.0; //asignación
16   Circulo cir3 {30.0}; //inicializacion uniforme
17   Circulo cir4 = {40.0} //pod-like
18
19   cout << "circunferencia de cir1: "<<cir1.circunf()<< endl;
20   return 0;
21 }
```

```
circunferencia
de cir1: 62.8319
```

Una ventaja de la inicialiación uniforme sobre la forma funcional es que, a diferencia de los parétesis redondos, los corchetes no pueden confundirse con la declaración de una función, y por tanto puede se usada para llamar de manera explicita constructores por defecto:

```
1 Rectangulo recb;   // constructor por defecto es llamado
2 Rectangulo recc(); // declaración de una función, constructor defecto NO llamado
3 Rectangulo recd{}; // constructor por defecto es llamado
```

La sintaxis para el llamado del constructor es cuestión de estilo personal. La mayoría de los códigos existentes utilizan la forma funcional y algunos nuevos estilos sugieren la escogencia de la inicialización uniforme sobre las otras, incluso aunque ésta tenga potenciales problemas por la preferencia de `listas_de_inicializacion` como de su tipo.

3.4.3.1 Inicialización de objetos miembros en costructores

Cuando un constructor es usado para inicializar otros miembros de la clase, estos miembros puede ser inicializados directamente, sin restaurar las declaraciones en su cuerpo. Esto se hace insertando dos puntos (`:`) antes del constructor y después una lista de inicializaciones para los miembros de la clase. Por ejemplo, consideremos la clase del código fuente 44:

```
class Rectangulo {
    int ancho, alto;
  public:
    Rectangulo (int,int);
    int area () {return ancho*alto;}
};
```

El constructor de esta clase puede ser definido de la manera usual:

```
Rectangulo::Rectangulo(int x,int y) { ancho=x; alto=y; }
```

Pero el constructor también puede ser definido usando la inicialización de miembros:

```
Rectangulo::Rectangulo(int x,int y) : ancho(x) { alto=y; }
```

O incluso:

```
Rectangulo::Rectangulo(int x,int y) : ancho(x), alto(y) { }
```

En este último caso, lo que hace el constructor es simplemente inicializar sus miembros, de aquí que tenga el cuerpo vacío o corchetes vacíos.

Para miembros de tipos fundamentales no hace diferencia cuál de las anteriores definciones de constructor es usada, ya que no son inicializadas por defecto, pero para objetos miembros (aquellos cuyo tipo de dato es una clase), si no están inicializados después de los dos puntos quiere decir que estan construidos por defecto.

Construir por defecto todos los miembros de la clase, puede o no puede ser conveniente, depende del caso. Es un desperdicio de memoria cuando los miembros son reinicializados de otra manera en el constructor. Pero en algunos casos, la construcción por defecto puede incluso ser no posible cuando la clase no tiene un constructor por defecto, en estos casos los miembros tienen que ser inicializados en la lista de inicialización de miembros. Por ejemplo:

Código fuente 47:

```
1  // Inicialización de miembros
2  #include <iostream>
3  using namespace std;
4  #define PI 3.14159265358979323846
5
6  class Circulo {
7      double radio;
8    public:
9      Circulo (double r): radio (r) {}
10     double area () { return radio*radio*PI; }
11 };
12
13 class Cilindro {
14     Circulo base;
15     double altura;
16   public:
17     Cilindro (double r,double h): base (r), altura (h){}
18     double volumen () { return base.area()*altura; }
19 };
20
21 int main () {
22   Cilindro cil (10,20);
23
24   cout << "volumen del cilindro: " << cil.volumen() << endl;
25   return 0;
26 }
```

```
Volumen del cilindro: 6283.19
```

En éste ejemplo, la clase `Cilindro` tiene un objeto miembro cuyo tipo es de otra clase (el tipo de `base` es `Circulo`). Debido a que los objetos de la clase `Circulo` sólo pueden ser construidos con un ́solo parámetro, el constructor de la clase `Cilindro` necesita llamar al constructor de `base`, y la única manera de hacerlo es en la *lista de inicializadores de miembros*.

Estas inicializaciones pueden usar únicamente la sintaxis de la inicialización uniforme, usando braquets `{}` en lugar de paréntesis redondos `()`:

```
Cilindro::Cilindro(double r,double h) : base{r}, altura{h} { }
```

3.4.4. Punteros a clases

Los objetos también pueden ser apuntados usando punteros. Una vez declarada, una clase es un tipo de dato válido, así que puede ser usado como tipo de dato para un puntero. Por ejemplo:

```
Circulo * cir;
```

Se tiene `cir` el cual es un puntero a un objeto de la clase `Circulo`.

De manera similar a las estructuras, los miembros de un objeto pueden ser accedidos directamente desde un puntero usando el operador flecha (`->`). Se presenta un ejemplo con algunas posibles combinaciones:

Código fuente 48:

```
1  // punteros a clases
2  #include <iostream>
3  using namespace std;
4
5  class Rectangulo {
6      int ancho, alto;
7    public:
8      Rectangulo(int x,int y) : ancho(x), alto(y) { }
9
10     int area (void) {return ancho*alto;}
11 };
12
13 int main () {
14   Rectangulo obj (3,4);
15   Rectangulo * rec, * tan, * gul;
16   cre = &obj;
17   tan = new Rectangulo (5,6);
18   gul = new Rectangulo[2]{{2,5},{3,6}};
19   cout << "Area de obj " << obj.area() << endl;
20   cout << "Area de rec " << rec->area() << endl;
21   cout << "Area de tan " << tan->area() << endl;
22   cout << "Area de gul[0] " << gul[0].area() << endl;
23   cout << "Area de gul[1] " << gul[1].area() << endl;
24   delete tan;
25   delete[] gul;
26
27   return 0;
28 }
```

El código fuente 48 hace uso de varios operadores para operar sobre objetos y punteros, como lo son los operadores `*, &, ., ->, []` . Estos se interpretan de la siguiente manera:

Expresión	**Puede ser leída de manera:**
`*x`	Puntero `x`
`&x`	Dirección de `x`
`x.y`	Miembro `y` del objeto `x`
`x->y`	Miembro `y` del objeto apuntado por el puntero `x`
`(*x).y`	Miembro `y` del objeto apuntado por el puntero `x` (Equivalente a la anterior)
`x[0]`	Primer objeto al que apunta el puntero `x`
`x[1]`	Segundo objeto al que apunta el puntero `x`
`x[n]`	(n+1)-ésimo objeto al que apunta el puntero `x`

La mayoría de éstas expresiones han sido introducidas en secciones anteriores. Recuerde que en el taller sobre arreglos se introduce el operador (`[]`) y en el de estructuras se introduce el operador flecha (`->`).

3.4.5. Clases definidas con `struct` y `union`

Las clases no son únicamente definidas con la palabra reservada `class`, también es posible utilizar las palabras reservadas `struct` y `union`.

La palabra `struct`, generalmente es usada para declarar estructuras, pero puede también ser utilizada para declarar clases que tienen funciones miembros, con la misma sintaxis como con `class`. La única diferencia entre ambas es que los miembros de las clases declarados con `struct` tienen acceso público por defecto, mientras que las clases declaradas con `class` tienen acceso privado por defecto. De resto, para cualqueir otro propósito, ambas `struct` y `class` son equivalentes en el contexto discutido hasta ahora en estas notas.

Por el contrario, el concepto de uniones, es diferente de las clases construidas con `struct` y `class`, debido a que las uniones sólo almacenan un miembro o tipo de dato a la vez, pero sin embargo, también son clases y por tanto pueden también soportar funciones como miembros. El acceso por defecto en clases tipo unión es público.