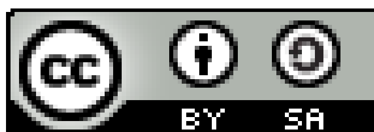
Notas del lenguaje C++

Federico Améndola
Martín Fernández

Consultas y sugerencias: laboratorio.ayda@alumnos.exa.unicen.edu.ar



Licencia creative commons Atribución-Compartir Obras Derivadas Igual 2.5 Argentina
http://creativecommons.org/licenses/by-sa/2.5/ar/

Usted es libre de:

- COPIAR, DISTRIBUIR, EXHIBIR Y EJECUTAR LA OBRA
- HACER OBRAS DERIVADAS

Bajo las siguientes condiciones:

- Atribución: Usted debe atribuir la obra en la forma especificada por el autor o el licenciante.
- Compartir Obras Derivadas Igual. Si usted altera , transforma, o crea sobre esta obra, sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

Índice de contenido

Introducción........1
Programas........2
Identificadores........2
Bloques........2
Tipos y Variables........3
Declaración........3
Ámbito de las variables........3
Visibilidad........4
Tipos básicos de C++........5
Modificadores opcionales........5
Secuencias de escape........6
Constantes........6
Tipos de datos compuestos........7
Enumeraciones........7
Registros........7
Uniones........8
Arreglos........9
Arreglos como parámetros........10
Operadores........11
Estructuras de control........12
Estructuras de selección........12
Sentencia if........12
Sentencia switch........12
Estructuras de iteración........13
Sentencia while........13
Sentencia do-while........13
Sentencia for........13
Funciones........14
Declaración........14
Definición........14
Pasaje de parámetros........15
Sobrecarga de funciones........16
Funciones recursivas........17
Comparación entre iteración y recursión........17
Argumentos de los programas........18
Bibliotecas........19
Directiva include........19
Espacios de nombres........19
Bibliotecas para las funciones más comunes........20
Manejo de entrada/salida por consola........21
Manejo de la consola en C++........21
Salida por consola 21
Entrada por consola........21
Sincronización de la entrada........21
Leer líneas completas........22
Configuración del formato de la entrada y la salida........23
Manejo de la consola en C........23
Entrada por consola........24
Manejo de cadenas........24
Cadenas de C++........24

Cadenas de C....................25
Inicialización de las cadenas de caracteres....................25
Operaciones de entrada/salida por consola....................26
Funciones para el manejo de cadenas....................26
Manejo de archivos....................27
Streams de archivos de C++....................27
Archivos de texto....................29
Archivos binarios....................31
Streams de archivos de C....................33
Archivos de texto....................33
Archivos binarios....................35
Punteros y manejo de memoria....................36
Operadores relacionados con punteros....................37
Operaciones para obtener y liberar memoria....................38
Operaciones para obtener memoria....................38
Operaciones para liberar memoria....................38
Consideraciones sobre la utilización de malloc y free....................39
Obteniendo bloques de memoria contigua....................39
Arreglos y punteros....................40
Arreglos como parámetros de funciones....................41
Arreglos dinámicos....................41
Aritmética de punteros....................43
Clases....................45
Definición de clases - Declaración de la interfaz....................45
Instanciación y uso....................46
Definición/Implementación de la interfaz....................47
Métodos consultores....................48
Constructores....................48
Destructor....................51
Sobrecarga de operadores....................51
Operador de salida de streams....................52
Operador de asignación....................53
La palabra reservada this....................53
Clases parametrizadas....................54
Herencia....................56
Creación de una clase derivada....................56
Constructores y destructores de las clases derivadas....................57
Redefinición de miembros en las clases derivadas....................58
Polimorfismo....................59
Métodos virtuales....................59
Clases abstractas....................60

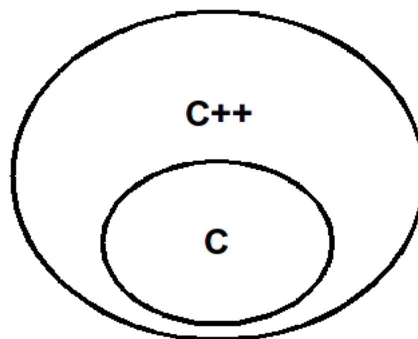
Introducción

El lenguaje que se utilizará durante la cursada de la cátedra de Análisis y Diseño de Algoritmos I y II es C++.

C++ tiene sus raíces en el lenguaje de programación C. El lenguaje C es imperativo y procedural, orientado a la implementación de sistemas operativos y por ello es que cuenta con características de bajo nivel (es decir, permite operaciones dependientes directamente de la arquitectura de la máquina). Si bien C también fue utilizado para el desarrollo de aplicaciones de usuario, en la actualidad su uso principal es en el desarrollo de software base o de sistemas: compiladores, sistemas operativos, bibliotecas, sistemas embebidos, etc; debido a su simplicidad y a que, al utilizar operaciones cercanas a la arquitectura de la máquina subyacente, nos permite conocer exactamente lo que nuestro programa está realizando y desarrollar técnicas que optimicen el funcionamiento del mismo.

C++ es una extensión del lenguaje C desarrollado en 1979, que introduce el soporte para la abstracción de datos, la programación orientada a objetos y la programación genérica. Por esto se dice que C++ es un lenguaje multi-paradigma (permitiendo mezclar la programación procedural y la programación orientada a objetos). Las características que agrega C++ son: clases, herencia múltiple, polimorfismo, sobrecarga de operadores, plantillas y manejo de excepciones. Todas estas características son propias de lenguajes de alto nivel, por lo que C++ se considera un lenguaje con un nivel de abstracción mucho mayor que C (aunque sigue manteniendo las características de bajo nivel de C). Es por esto que, el lenguaje C++, es más apropiado que C para el desarrollo de aplicaciones grandes gracias a la mayor facilidad y rapidez de desarrollo que brindan sus características avanzadas.

C++ tiene la misma sintaxis e implementa casi todas las características de C, de forma que la mayoría de los programas escritos en C compilaran con un compilador de C++, sin la necesidad de realizar cambios en el código fuente. Por otra parte, si bien se realizan esfuerzos para maximizar la compatibilidad entre ambos lenguajes, los mismos son independientes (de hecho, el estándar de C++ no soporta todas las características del estándar de C y hay puntos donde los mismos entran en conflicto[1]). Teniendo en cuenta lo anterior y dejando de lado las definiciones estrictas, conceptualmente, podemos considerar que C++ es un superconjunto de C, como se indica en la siguiente figura.



El estándar del lenguaje C++ se sigue actualizando con el paso de los años, ratificando y agregando nuevas características para mejorar la experiencia a la hora de desarrollar aplicaciones. La última actualización fue en el año 2011 y se espera otra para el 2014. También es importante mencionar que existen muchos compiladores de C++, comerciales y libres, donde cada uno adopta el estándar del lenguaje de una forma particular. Debido a esto, siempre es recomendable utilizar un compilador que se ajuste y respete lo más posible el estándar definido.

Al soportar todas las características de C, C++ puede considerarse un lenguaje complicado para aprender y dominar. Esto viene de la confusión que surge de la posibilidad de realizar la misma operación de varias formas distintas, utilizando los mecanismos y el estilo de C o el de C++. Debido a esto, esta guía no pretende cubrir todos los aspectos de este lenguaje (ni reemplazar la bibliografía recomendada), sino introducir los mecanismos más importantes, centrándose en aquellos más complejos de una forma que resulte lo más clara posible.

[1] Para una comparación más detallada ver http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

Programas

El siguiente es el programa más simple que se puede realizar en C++:

```
int main()
{
    return 0;
}
```

En C++ todo programa debe implementar una función principal llamada **`main`** que debe retornar un valor entero indicando el resultado de la ejecución del mismo; 0 si el programa finalizó normalmente o distinto de 0 si ocurrieron errores durante su ejecución.

Identificadores

Es el nombre que permite hacer referencia a las constantes, variables y funciones de un programa. Las reglas para escribir identificadores son:

- Deben comenzar con una letra o guión bajo.
- Sólo letras (A-Z, a-z), dígitos (0-9) o el guión bajo (_) pueden seguir al primer símbolo.

Una característica del lenguaje de extrema importancia es que es *case sensitive*, es decir, que distingue entre mayúsculas y minúsculas a la hora de nombrar identificadores. Es por eso que “`main`”, “`Main`”, “`MAIN`”, “`mAiN`”, etc, son considerados identificadores distintos en C++ (a diferencia de lo que sucede en lenguajes como Pascal y BASIC, donde se considera que dichos nombres hacen referencia al mismo identificador).

Bloques

Un bloque es una sentencia compuesta, se trata de una sucesión (que puede estar vacía) de sentencias delimitadas por un par de corchetes `{ }`:

```
{
    <sentencia1>;
    <sentencia2>;
    <sentencia3>;
    ...
}
```

Desde el punto de vista sintáctico, un bloque puede ser considerado como una sola sentencia. Teóricamente, los bloques pueden ser anidados a cualquier nivel (profundidad). El aspecto de los bloques "anidados" es el siguiente:

```
{
    ...
    {
        <sentencia1>;
        <sentencia2>;
        <sentencia3>;
        ...
    }
    ...
}
```

Tipos y Variables

Declaración

Para utilizar una variable en C++, primero se debe declarar especificando el nombre de la variable (un identificador válido) y que tipo de datos se quiere almacenar:

```
<tipo> <variable1>, <variable2>, ..., <variablen>;
```

Por ejemplo:

```
int a, b, c;
```

Es posible inicializar una variable en el mismo momento en que se declara:

```
<tipo> <variable1> = <valor_inicial>, <variable2> = <valor_inicial>, ...;
```

Por ejemplo:

```
int a=1, b=2, c=4;
```

Ámbito de las variables

Dentro de un programa se pueden encontrar dos tipos de ámbito para las variables:

- **variables locales**: únicamente visibles dentro de la función o bloque donde se han declarado y, de hecho, no existen fuera de ese ámbito. Las variables locales pueden ser declaradas en cualquier parte de un bloque o función[2] pero sólo podrán intervenir en sentencias posteriores a su declaración.
- **variables globales**: declaradas exteriormente a las funciones, pueden ser utilizadas por aquellas que fueron declaradas luego de las mismas.

```
// Declaración de variables globales a funcion1, funcion2 y main

<tipo> <funcion1>
{
    ...
    // Declaración de variables locales
    ...
}

// Declaración de variables globales a funcion2 y main

<tipo> <funcion2>
{
    ...
    // Declaración de variables locales
    ...
}

...
// Declaración de variables globales a main

int main()
{
    ...
    // Declaración de variables locales
    ...
}
```

[2] A diferencia de C que exige que todas las variables se declaren al principio del bloque.

Por ejemplo, analizando el siguiente fragmento de código:

```
int a = 20;
int main(){
      int b = 10;
      { // Inicio subbloque
            int c = 15;
            cout << "Variable a:" << a;
            cout << "Variable b:" << b;
            cout << "Variable c:" << c;
      } // Fin subbloque
      cout << "Variable a:" << a;
      cout << "Variable b:" << b;
      cout << "Variable c:" << c; // Error: c no está declarada en este ámbito
      return 0;
}
```

La variable **a** es global a todo el archivo donde se encuentra el `main`, por lo tanto podrá ser utilizada por cualquier función o procedimiento que se defina luego de la declaración de la variable.

La variable **b** es local a la función `main` y global a todos los bloques definidos dentro de la misma, como por ejemplo, el subbloque. Dentro de este bloque se encuentra declarada la variable **c**, la cual es local al subbloque. Fuera de este bloque, la variable **c** no es visible. La porción de código en donde cada variable es visible se denomina alcance o ámbito. Para las variables locales, el alcance comienza en la sentencia en donde la variable está declarada y termina en la llave final que delimita el bloque en el cual fue definida.

Visibilidad

Un mismo identificador puede ser “ocultado” por otro del mismo nombre declarado en un bloque interior al primero. Por ejemplo, la declaración de una variable local puede “ocultar” una variable global con el mismo identificador:

```
int a = 20;
int main(){
      int b = 10;
      { // Inicio subbloque
            int b = 15;
            cout << "Variable A: " << a << "\n";
            cout << "Variable B: " << b << "\n";
      } // Fin subbloque
      cout << "Variable A: " << a << "\n";
      cout << "Variable B: " << b << "\n";
      return 0;
}
```

En este caso, la variable **b** declarada como local en el subbloque, oculta a la variable **b** declarada en la función `main`; debido a esto, el resultado del código anterior es el siguiente:

```
> Variable A: 20
> Variable B: 15
> Variable A: 20
> Variable B: 10
```

Tipos básicos de C++[3]

<table>
<tr><th>Nombre</th><th>Descripción</th><th>Valores</th></tr>
<tr><td><code>char</code></td><td>Caracter (o un entero pequeño)</td><td>Los caracteres imprimibles del código ASCII más algunos no imprimibles representados en el código fuente a través de secuencias de escape.
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
_ $ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
<HT> <VT> <NL> <FF> <SPACE></td></tr>
<tr><td><code>w_char</code></td><td>Caracter ancho</td><td>Caracteres del juego de caracteres Unicode[4].</td></tr>
<tr><td><code>int</code></td><td>Enteros</td><td>con signo: -2147483648 a 2147483647
sin signo: 0 a 4294967295</td></tr>
<tr><td><code>float</code></td><td>Punto flotante</td><td>3.4e +/- 38 (7 dígitos)</td></tr>
<tr><td><code>double</code></td><td>Punto flotante de doble precisión</td><td>1.7e +/- 308 (15 dígitos)</td></tr>
<tr><td><code>bool</code></td><td>Booleano</td><td>true o false</td></tr>
<tr><td><code>void</code></td><td>Ausencia de valor</td><td>Es un tipo especial que representa la ausencia de valor. Generalmente se utiliza para indicar que una función no devuelve un valor o no requiere ningún parámetro. También tiene otras formas de uso que involucran punteros, pero no se verán por el momento.
No está permitida la declaración:
<code>void <identificador>;</code></td></tr>
</table>

Modificadores opcionales

A excepción de los tipos `void` y `bool`, los tipos básicos pueden tener modificadores opcionales, que se usan para modificar el significado del tipo base (en cuanto a rango de valores y espacio de almacenamiento). Estos modificadores indican: **con signo** (`signed`), **sin signo** (`unsigned`), **largo** (`long`) y **corto** (`short`**).**

```
[signed|unsigned] [short|long|long long] <tipo> <identificador>;
```

Por ejemplo:

```
signed int a;      // Equivalente a: int a;
unsigned int a;
unsigned long int a;
```

Los modificadores opcionales aplicados a un tipo básico definen un nuevo tipo, de forma que por ejemplo, `short` y `unsigned short` son tipos distintos.

Un `char` puede ser con signo (`signed`), sin signo (`unsigned`), o no especificado; por defecto se suponen con signo. Mientras que sólo se permite un modificador adicional (`long`), para el tipo `double`, dando lugar a los `long double`.

[3] El tipo w_char junto con el tipo bool son exclusivos de C++.

[4] Para ver las ventajas de código Unicode frente al ASCII ver http://www.i18nguy.com/UnicodeBenefits.html

A continuación se presenta una tabla con el rango de valores para todos los tipos fundamentales:

Tipo	Rango de valores
`unsigned char`	0 <= X <= 255identificador
`char (signed)`	-128 <= X <= 127
`short (signed)`	-32,768 <= X <= 32,767
`unsigned short`	0 <= X <= 65,535
`unsigned (int)`	0 <= X <= 4,294,967,295
`int (signed)`	-2,147,483,648 <= X <= 2,147,483,647
`unsigned long (int)`	0 <= X <= 4,294,967,295
`long (int)`	-2,147,483,648 <= X <= 2,147,483,647
`float`	32 1.18e-38 <= |X| <= 3.40e38
`double`	2.23e-308 <= |X| <= 1.79e308
`long double`	3.37e-4932 <= |X| <= 1.18e4932

Secuencias de escape

Secuencia	Carácter	Descripción
****	\	Barra invertida
\'	'	Comilla simple (apóstrofo)
\"	"	Comillas dobles
\?	?	Interrogación
\0	<NULL>	Cero binario
\a	<BEL>	Campana (sonido audible)
\b	<BS>	Retroceso
\f	<FF>	Salto de página
\n	<NL>	Salto de línea
\r	<CR>	Retorno de carro
\t	<HT>	Tabulación horizontal
\v	<VT>	Tabulación vertical

Constantes

- Números enteros: es posible escribirlos como números decimales, octales (precedidos de 0) o hexadecimales (precedidos por `0x`).
- Números de punto flotante: pueden incluir el punto decimal y/o la letra “`e`” acompañada de un número entero X, lo cual se interpreta como multiplicar por 10^X.
- Caracteres: deben escribirse entre apóstrofos (").
- Cadenas de caracteres: son secuencias de cero o más caracteres encerrados entre comillas dobles.
- Declaradas (`const`): de modo similar al de las variables, es posible asociar un valor constante, a un identificador de un tipo determinado:

```
const <tipo> <constante> = <valor_constante>;
```

Por ejemplo:

```
const double PI = 3.14;
```

- Definidas (#define): existe la posibilidad de definir constantes utilizando la directiva de compilador define:

```
#define <constante> <valor_constante>;
```

Por ejemplo:

```
#define MAX 20;
```

Las constantes definidas de esta forma **no tienen tipo** lo cual hace que no sea recomendable su uso y se utilice, siempre que sea posible, constantes declaradas con const.

Tipos de datos compuestos

Además de los tipos de datos básicos ya vistos, C++ permite definir tipos de datos llamados compuestos o estructurados. Esta denominación se debe a que los tipos de datos compuestos son agrupaciones de otros tipos de datos.

Dentro de los tipos de datos compuestos se pueden encontrar las enumeraciones, los registros, las uniones, los arreglos, las cadenas de caracteres, los punteros y la memoria dinámica (esta última se verá en una sección posterior).

Enumeraciones

Una enumeración es una lista de valores enteros constantes:

```
enum <tipo_enumeración> {<constante1>, <constante2>, ..., <constanten>};
```

En la definición anterior, la primer constante del **enum** tiene valor 0, la siguiente 1, y así sucesivamente incrementando de a 1. También pueden especificarse valores explícitos:

```
enum <tipo_enumeración> {<constante1>=<valor_constante>,
                           <constante2>=<valor_constante>, ...};
```

Por ejemplo:

```
enum posicion{primero, segundo, tercero};
```

Registros

La declaración de un registro permite agrupar una o más variables bajo un mismo nombre. Estas variables pueden ser de cualquier tipo. Para declarar una estructura se utiliza la siguiente sintaxis:

```
struct <tipo_registro>
{
    <tipo> <nombre_campo1>;
    <tipo> <nombre_campo2>;
    ...
    <tipo> <nombre_campon>;
};
```

El identificador del registro puede ser utilizado como el identificador de un nuevo tipo definido, es decir, para declarar variables del tipo del registro. Por ejemplo, se puede definir una estructura para almacenar información de empleados:

```
struct Empleado {
    int id;
    string nombre;
    string apellido;
    float salario;
};
```

Para acceder a los distintos campos que componen una estructura debe emplearse el operador "**.**", del siguiente modo:

```
<variable_registro>.<nombre_campo>
```

Por ejemplo:

```
int main()
{
    Empleado emp;
    emp.id = 1;
    emp.salario = 999.99;
    emp.nombre = "Juan";
    emp.apellido = "Perez";
}
```

Una estructura se puede componer de variables de cualquier tipo de datos, y las estructuras en sí, se utilizan como tipos de datos; por lo tanto, se permite la creación de estructuras anidadas. Es decir, es posible definir estructuras donde algunos de sus campos sean, a su vez, otras estructuras. Por ejemplo:

```
struct Seccion
{
    int id;
    string nombre;
    Empleado encargado;
};
```

Y acceder a los campos, de forma similar:

```
int main() {
    Seccion seccion;
    seccion.id = 1;
    seccion.encargado.salario = 999.99;
    seccion.encargado.nombre = "Juan";
    ...
}
```

Uniones

Las uniones son similares a los registros en la definición pero difieren en el funcionamiento durante su utilización. Esto se debe a que una unión permite almacenar en la misma porción de memoria diferentes tipos de datos. Para declarar una unión se utiliza la siguiente sintaxis:

```
union <tipo_union>
{
    <tipo> <nombre_campo1>;
    <tipo> <nombre_campo2>;
    ...
};
```

Debido a que todos los elementos de una unión ocupan el mismo espacio de memoria física, el tamaño de la unión será equivalente al tamaño del elemento declarado, que ocupa más lugar. Otra concecuencia de este funcionamiento es que la modificación de un elemento llevará a la modificación de todos los elementos, a veces con resultados impredecibles.
Por ejemplo, si se define:

```
union Descuento {
    int monto;
    float porcentaje;
    };

Descuento descuento;
```

Se podrá acceder a `descuento.monto` y `descuento.porcentaje`, pero no a ambos elementos al mismo tiempo. Por lo tanto, generalmente se suele crear un registro con un elemento adicional para

identificar el elemento que fue almacenado en cada variable, y, de esta forma accederlo, sin inconvenientes. Por ejemplo:

```
struct Descuento {
    char tipo;
    union {
        int monto;
        float porcentaje;
    } metodo;
};
```

Luego, para acceder a cada elemento se realizaría de la siguiente forma:

```
Descuento descuento;
descuento.tipo = 'm';
descuento.metodo.monto = 10;
```

Arreglos

Un arreglo es una colección de elementos del mismo tipo de datos, que ubica a todos los elementos en regiones de memoria contigua y a los cuales permite acceder mediante la utilización de un índice.

Para declarar un arreglo se utiliza la siguiente sintaxis:

```
<tipo> <nombre_arreglo>[<tamaño>];
```

El parámetro `<tamaño>` debe ser una constante o variable entera positiva que indique el número de elementos máximo que podrá contener el arreglo. Por ejemplo para declarar un arreglo de diez enteros:

```
int arr[10];
```

Para acceder al valor de una posición del arreglo se utiliza el operador []:

```
<nombre_arreglo>[<índice>]
```

Donde `0 <= <índice> < <tamaño>`, es decir, los arreglos siempre comienzan en 0, por lo tanto, el último índice válido será `tamaño - 1`.

Este acceso se utilizará tanto para modificar como para obtener el valor almacenado. En cualquiera de los casos, el resultado del acceso será equivalente a trabajar con una variable del tipo del arreglo. Por ejemplo, en el caso siguiente ambos accesos equivalen a una variable de tipo **`int`**:

```
arr[2] = 10;
int variable = arr[4];
```

Es importante tener en cuenta que no se realiza ninguna verificación de acceso al arreglo (ni en tiempo de compilación ni en tiempo de ejecución). Es decir, que si se accede a posiciones del arreglo, más allá de las definidas por su tamaño, es muy probable que el programa no termine correctamente su ejecución a raíz del acceso inválido. Este acceso probablemente resultará en un valor "basura" (lo cual llevará a errores en el comportamiento del programa, cuyo origen puede ser difícil de detectar).

También pueden declararse arreglos multidimensionales:

```
<tipo> <nombre_arreglo>[<tamaño_dim1>][<tamaño_dim2>]...[<tamaño_dimn>];
```

Los arreglos pueden ser inicializados en la misma declaración como el resto de las variables.

```
<tipo> <nombre_arreglo>[<tamaño>] = {<valor1>, <valor2>, ... <valorn>};
```

Ejemplos:

```
char vocales [5] = {'a', 'e', 'i', 'o', 'u'};
//Al inicializar el arreglo es posible no definir su tamaño, el compilador lo
tomará de la cantidad de valores.
char vocales [] = {'a', 'e', 'i', 'o', 'u'};
int matriz [2][3] = {{ 1, 2, 3 }, { 4, 5, 6 }};
```

```
int matriz [2][3] = { 1, 2, 3, 4, 5, 6 };
```

A los arreglos de dos dimensiones generalmente se los denomina ***Matrices***, ya que es la estructura que se forma de un arreglo donde cada posición es, a su vez, otro arreglo.

Los arreglos no están limitados a dos dimensiones, ya que pueden contener la cantidad de índices que sean necesarios (dependiendo de la memoria disponible).

Arreglos como parámetros

En C++ no es posible pasar un bloque completo de memoria como parámetro por valor o copia a una función, pero si es posible pasar su dirección. Por ejemplo, para declarar una función que tiene como argumento un arreglo se debe declarar como sigue:

```
void imprimir(int arreglo[]);
```

Luego se podrá invocar mediante:

```
int valores;
imprimir(valores);
```

Como se puede ver no se indica el tamaño del arreglo que se va a pasar, sólo que es un arreglo de elementos de un tipo de datos determinado (en este caso `int`). Por lo tanto, cualquier arreglo que respete el tipo de datos declarado podrá ser pasado como parámetro a la función. Debido a esto es que, generalmente, se agrega a las funciones que trabajan con arreglos un segundo parámetro para determinar el tamaño de dicho arreglo.

```
void imprimir(int arreglo[], int capacidad);
```

También es posible pasar por parámetro un arreglo multidimensional. En estos casos, el tamaño de la primera dimensión no se especifica pero si se debe incluir el tamaño de las demás dimensiones en la declaración de la función, por ejemplo:

```
void imprimir(int arreglo[][4], int capacidad);
```

El tamaño de la segunda dimensión es necesario para que el compilador determine la capacidad de dicha dimensión. Debido a esto, la función sólo podrá ser invocada con arreglos multidimensionales cuya segunda dimensión sea igual a 4.

El pasaje de arreglos, de cualquier dimensión, como parámetros a una función plantea una situación especial. Debido a que los bloques de memoria contigua no se pueden copiar automáticamente, al pasar un arreglo por copia a una función en realidad se está pasando un puntero al arreglo (esto se explicará en detalle en el apartado de punteros). Por lo tanto, se podrán modificar los valores de los elementos que el mismo contiene. Por ejemplo, se puede definir la siguiente función para cargar valores por defecto a un arreglo:

```
void inicializar(int arreglo[], int capacidad) {
    for(int i=0; i < capacidad; i++)
        arreglo[i] = 0;
}
```

Esta característica de los arreglos, como argumentos de las funciones, es algo que hay que tener muy en cuenta a la hora de estructurar los programas ya que puede generar errores inesperados.

Operadores

C++ incorpora el tipo de datos `bool`, el cual contiene dos literales, `false` y `true`. Una expresión booleana o lógica es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores de verdad posibles: verdadero o falso. C no tiene tipos de datos lógicos o booleanos para representar los valores verdadero o falso. En su lugar utiliza el tipo `int` para ese propósito, con el valor 0 representando falso y cualquier otro valor representando verdadero.

A continuación se presenta una lista de los operadores ordenados de mayor a menor precedencia:

<table>
<tr><th>Nivel de precedencia</th><th>Operadores</th><th>Descripción</th><th>Agrupamiento</th></tr>
<tr><td>1</td><td><code>::</code></td><td>Ámbito</td><td>Izquierda a derecha</td></tr>
<tr><td>2</td><td><code>() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid</code></td><td>Posfijo</td><td>Izquierda a derecha</td></tr>
<tr><td rowspan="3">3</td><td><code>++ -- ~ ! sizeof new delete</code></td><td>Unarios (prefijo)</td><td rowspan="3">Derecha a izquierda</td></tr>
<tr><td><code>* &</code></td><td>Indirección y referencia (punteros)</td></tr>
<tr><td><code>+ -</code></td><td>Signo</td></tr>
<tr><td>4</td><td><code>(type)</code></td><td>Conversión de tipo</td><td>Derecha a izquierda</td></tr>
<tr><td>5</td><td><code>.* ->*</code></td><td>Acceso a miembro (punteros)</td><td>Izquierda a derecha</td></tr>
<tr><td>6</td><td><code>* / %</code></td><td>Multiplicativos</td><td>Izquierda a derecha</td></tr>
<tr><td>7</td><td><code>+ -</code></td><td>Aditivos</td><td>Izquierda a derecha</td></tr>
<tr><td>8</td><td><code><< >></code></td><td>Desplazamiento</td><td>Izquierda a derecha</td></tr>
<tr><td>9</td><td><code>< > <= >=</code></td><td>Relacionales</td><td>Izquierda a derecha</td></tr>
<tr><td>10</td><td><code>== !=</code></td><td>Igualdad</td><td>Izquierda a derecha</td></tr>
<tr><td>11</td><td><code>&</code></td><td>AND a nivel de bits</td><td>Izquierda a derecha</td></tr>
<tr><td>12</td><td><code>^</code></td><td>XOR a nivel de bits</td><td>Izquierda a derecha</td></tr>
<tr><td>13</td><td><code>|</code></td><td>OR a nivel de bits</td><td>Izquierda a derecha</td></tr>
<tr><td>14</td><td><code>&&</code></td><td>AND lógico</td><td>Izquierda a derecha</td></tr>
<tr><td>15</td><td><code>||</code></td><td>OR lógico</td><td>Izquierda a derecha</td></tr>
<tr><td>16</td><td><code>?:</code></td><td>Condicional</td><td>Derecha a izquierda</td></tr>
<tr><td>17</td><td><code>= *= /= %= += -= >>= <<= &= ^= |=</code></td><td>Asignación</td><td>Derecha a izquierda</td></tr>
<tr><td>18</td><td><code>,</code></td><td>Coma</td><td>Izquierda a derecha</td></tr>
</table>

- Si dos o más operadores aparecen en una expresión los mismos se aplican en el orden dado por su nivel de precedencia.
- En el caso que se tenga que resolver operadores del mismo nivel, los operadores se evalúan siguiendo el orden dado por el agrupamiento.
- Para forzar un orden específico para la evaluación de una expresión se pueden utilizar los paréntesis.

Estructuras de control

Estructuras de selección

Las estructuras de selección o estructuras condicionales controlan si una sentencia, o secuencia de sentencias, se ejecutará en función del cumplimiento o no de una condición o expresión lógica. C++ tiene dos estructuras de control para la selección, `if` y `switch`.

Sentencia `if`

La sentencia **`if`** decide si una sentencia o bloque de código debe ejecutarse o no en base al valor de una o más expresiones lógicas.

```
if (<expresión_lógica>)
    (<sentencia>|<bloque>)
{else if (<expresión_lógica>)
    (<sentencia>|<bloque>)}
[else
    (<sentencia>|<bloque>)]
```

Las expresiones lógicas se definen entre paréntesis. Una sentencia `if`, puede tener cero o más sentencias `else if` y opcionalmente una sentencia `else`. Las sentencias `else` y `else if` permiten incoporar decisiones de ejecución, en base a distintos criterios de selección. Por ejemplo:

```
int b;
cin >> b;
if (b > 10) {
    cout << "Error: valor mayor de lo permitido";
} else if (b < 0) {
    cout << "Error: valor menor de lo permitido";
} else {
    procesarValor(b);
}
```

Sentencia `switch`

La sentencia **`switch`** se utiliza para seleccionar una de entre múltiples alternativas. Esta sentencia es especialmente útil cuando la selección se basa en el valor de una variable de un tipo simple o de una expresión de un tipo simple, denominada expresión de control o selector.

```
switch (<expresión>) {
    case <expresión_constante> : (<sentencia>|<bloque>); break;
    {case <expresión_constante> :} (<sentencia>|<bloque>); break;
    [default : (<sentencia>|<bloque>) ; break;]
}
```

Una sentencia `switch` puede tener una o más sentencias `case` y opcionalmente una sentencia `default`. Todas las expresiones deben terminar con el comando `break`, de esta forma se evita que se evalúe la condición siguiente.
Por ejemplo:

```
int opcion;
cin >> opcion;
switch (opcion) {
    case 1:
        cout << "Opción 1";
        break;
    case 2:
    case 3:
        cout << "Opción 2 o 3";
        break;
    default: {
            cout << "Opción invalida\n" << "Ingrese una nueva opción";
        } break;
}
```

Estructuras de iteración

Las sentencias de iteración permiten repetir un conjunto de sentencias un número determinado de veces.

Sentencia `while`

Un ciclo **`while`** tiene una condición de control o expresión lógica que controla la repetición de la ejecución de una secuencia de sentencias.

```
while (<expresión_lógica>)
     (<sentencia> | <bloque>)
```

El cuerpo del ciclo se ejecuta mientras la expresión lógica sea cierta.

Por ejemplo:

```
int cont = 0;
while (cont <= 20) {
     cout << "Valor del contador: " << cont << "\n";
     cont++;
}
```

Sentencia `do-while`

La sentencia **`do-while`** se utiliza para definir un bucle en el que la condición de terminación se evaluará al final del ciclo.

```
do
     (<sentencia>|<bloque>)
while (<expresión_lógica>);
```

La expresión lógica se comprueba después de la ejecución del cuerpo del ciclo.

Por ejemplo:

```
int cont = 0;
do {
     cout << "Valor del contador: " << cont << "\n";
     cont++;
} while (cont <= 20);
```

El resultado de este ejemplo es similar al ejemplo anterior para el **`while`**, pero la diferencia reside en que el cuerpo de una sentencia **`do-while`** siempre se ejecuta al menos una vez.

Sentencia `for`

En general, la sentencia **`for`** se utiliza para definir un ciclo en el que una variable se incrementa o decrementa de manera constante en cada iteración y la finalización del ciclo se determina mediante una expresión que involucra dicha variable.

```
for (<inicializacion>; <expresión_lógica>; <incremento>)
     (<sentencia>|<bloque>)
```

donde en `<inicializacion>` se inicializa la variable de control del bucle. El cuerpo del ciclo se ejecuta mientras la expresión lógica sea cierta.

Por ejemplo:

```
for (int cont = 0; cont <= 20; cont++) {
     cout << "Valor del contador: " << cont << "\n";
}
```

`<inicializacion>`, `<expresión_lógica>` e `<incremento>` pueden omitirse, manteniendo los punto y coma.

Funciones

Una función es un bloque de código dentro de un programa, que posee las siguientes características:

- tiene un nombre que la identifica;
- puede retornar un resultado, llamado valor de retorno (cuyo tipo es el correspondiente al tipo de retorno); y,
- puede tener parámetros, que actúan como variables locales al bloque del cuerpo de la función.

Las funciones son ejecutadas a través de su invocación o llamado, para lo cual se escribe su nombre y entre paréntesis la lista de argumentos, los cuales son utilizados para inicializar los parámetros correspondientes. Al llamar a una función el control de ejecución del programa pasa al bloque de sentencias de la misma; hasta que se llega al final del bloque o se devuelve un valor, luego de lo cual el control vuelve a la función que realizó la llamada. Una función puede ser llamada por sí misma -un llamado recursivo- o por otras funciones.

Mediante el uso de funciones se pueden modularizar los programas, lo cual permite mejorar su estructura y evitar la duplicación de código, obteniendo un código más legible y fácil de mantener.

Declaración

La declaración de las funciones (también llamado prototipado de funciones) indica al compilador la existencia de las mismas y es necesaria antes de poder utilizarlas. La declaración incluye el nombre de la función, la lista de parámetros y el tipo de valor de retorno:

```
<tipo_retorno> <nombre_función>(<lista_de_parámetros>);//;para finalizar la
                                                          declaración
```

Debido a que en C++ todos los procedimientos son funciones, para indicar que una función no devuelve un valor se utiliza el tipo especial void, que indica la ausencia de valor. El tipo de valor de retorno puede ser cualquier tipo de datos excepto arreglos.

La lista de parámetros es una lista de declaraciones de objetos o referencias separadas por comas. En algunos casos, cuando se está definiendo un prototipo de una función, la lista de parámetros sólo incluye los tipos de los parámetros sin los nombres. La lista de parámetros también puede estar vacía lo que indica que la función no toma parámetros.

Una función puede ser declarada más de una vez en un mismo programa.

Definición

La definición de una función incluye el nombre, la lista de parámetros, el tipo de valor de retorno y el bloque de sentencias que constituyen el cuerpo de la función. La definición de las funciones completa la especificación de las mismas.

Es necesario que todas las funciones utilizadas cuenten con una definición para que el proceso de construcción del programa finalice con éxito. A diferencia de las declaraciones, sólo debe haber una única definición para cada función. Es más, las funciones pueden definirse sin necesidad de declararlas previamente, ya que el encabezado de la definición contiene la misma información que la declaración (nombre, cantidad y tipo de los parámetros y el tipo de retorno).

El formato de una definición es el siguiente:

```
<tipo_retorno> <nombre_función>(<lista_de_parámetros>)
{
    <sentencias>
}
```

Todas las funciones cuyo tipo de retorno no es `void` deben retornar un valor. El valor de retorno se especifica utilizando una sentencia con la palabra clave `return`:

```
return <expresión>;
```

El valor de la expresión debe coincidir con el valor del tipo de retorno. Cuando se llega a una sentencia `return` el control de la ejecución retorna a la función que realizó el llamado. Debido a esto, no se ejecutará código ubicado por debajo de la sentencia `return`.

Puede haber más de una sentencia `return` y las mismas pueden ubicarse en cualquier parte del código. Es importante asegurar que en todos los casos de ejecución posibles se devolverá un valor, es por eso que se recomienda que las sentencias `return` sean las mínimas indispensables y que se las ubique sobre el final del cuerpo de la función.

Ejemplos:

```
// Declaración de la función obtenerPromedio
float obtenerPromedio(int enteros[], int cantidad);

// Definición directa de la función imprimirEnteros
void imprimirEnteros(int enteros[], int cantidad)
{
    for (int i = 0; i < cantidad; i++) {
        cout << "Valor " << i << ": " << enteros[i] << "\n";
    }
}

const int N = 10;

int main()
{
    int arr[N] = {2,3,5,12,344,45,23,55,99,0};
    imprimirEnteros(arr, N);
      /* Se puede utilizar la función obtenerPromedio aunque no se haya
      especificado completamente, ya que para realizar el llamado es
      suficiente con la información provista por la declaración */
    cout << obtenerPromedio(arr, N) << "\n";
    return 0;
}

// Definición de función obtenerPromedio declarada anteriormente
float obtenerPromedio(int enteros[], int cantidad)
{
    float suma = 0;
    for (int i = 0; i < cantidad; i++) {
         suma = suma + enteros[i];
    }
    return suma / cantidad;
}
```

Pasaje de parámetros

La declaración de los parámetros permite indicar si los mismos estarán relacionados con los argumentos a través de la copia o la referencia.

El pasaje de argumentos por valor o copia se especifica a través de tipos comunes:

```
<tipo> <parámetro>
```

El pasaje de argumentos por referencia se especifica a través de tipos de referencia (utilizando el declarador de referencia `&` luego del tipo):

```
<tipo> & <parámetro>
```

En el caso del pasaje por valor, cada parámetro es un objeto local a la función, que es inicializado con el valor del argumento durante su creación, pero luego de eso opera en forma independiente. Esto significa que los cambios en el valor de los parámetros no afectarán al valor de los argumentos al momento de finalizar la ejecución de la función.

Por otra parte, los parámetros que son referencias constituyen un alias a los objetos utilizados como argumentos. Durante el pasaje de parámetros por referencia no se crean nuevos objetos ni ocurre ningún tipo de duplicación de los argumentos, sino que las referencias son inicializadas para operar directamente sobre estos argumentos. De esta forma los parámetros de referencia quedan vinculados a los argumentos y todas las operaciones que modifiquen su estado se verán reflejadas en el valor de los mismos cuando termine la ejecución de la función.

Ejemplo de pasaje de argumentos por copia:

```
void modificar (int variable)
{
    variable = 10;
    cout << "modificar - valor variable: " << variable << "\n";
}

int main()
{
    int variable = 2;
    modificar(variable);
    cout << "main - valor variable: " << variable << "\n";
}
```

La salida es la siguiente:

```
> modificar - valor variable: 10
> main - valor variable: 2
```

Ejemplo de pasaje de argumentos por referencia:

```
void modificar(int & variable)
{
    variable = 10;
    cout << "modificar - valor variable: " << variable << "\n";
}

int main()
{
    int variable = 2;
    modificar(variable);
    cout << "main - valor variable: " << variable << "\n";
}
```

La salida es la siguiente:

```
> modificar - valor variable: 10
> main - valor variable: 10
```

Se puede utilizar el pasaje de argumentos por referencia cuando es necesario que una función retorne más de un valor, ya que las funciones permiten un único valor de retorno.

El pasaje de argumentos por referencia es una mecanismo introducido en C++. C sólo permite el pasaje por valor o copia; por lo tanto, para que una función modifique el valor de un argumento se deben utilizar punteros (lo cual se ve reflejado en muchas de las declaraciones de la biblioteca estándar de C). Sin embargo, para programas escritos en C++ se recomienda utilizar el pasaje de parámetros por referencia y sólo utilizar punteros cuando sea realmente necesario.

Sobrecarga de funciones

Las funciones pueden tener el mismo nombre y valor de retorno pero con distintos tipos y cantidad de parámetros. El compilador determinará a qué función invocar examinando los argumentos y buscando una lista de parámetros que se ajuste a los mismos entre las funciones candidatas.

Por ejemplo:

```
float operar(int x, int y) {...}
float operar(float x, float y) {...}
```

Una función no puede ser sobrecargada sólo retornando diferentes tipos. Al menos uno de sus parámetros debe tener un tipo diferente.

Funciones recursivas

Se dice que una función es recursiva cuando se define en función de sí misma, es decir, en algún momento de su definición se encuentra una llamada a la misma función que se está definiendo. Existen dos tipos de recursión:

1. Directa: cuando una función se llama a sí misma.
2. Indirecta: cuando una función invoca a una segunda función y esta, a su vez, llama a la primera función.

La recursividad es un técnica elemental de programación que no todos los lenguajes permiten utilizar. C++ soporta la recursividad, para esto, cada vez que se llama a una función, se crea un juego de variables locales. Si la función hace una llamada a sí misma, se guardan sus variables y parámetros en la pila, y la nueva instancia de la función trabajará con su propia copia de las variables locales. Cuando esta segunda instancia de la función retorna, recupera las variables y los parámetros de la pila y continua la ejecución en el punto en que había sido llamada.

Una función recursiva puede dar origen a un problema muy común en programación, la recursión infinita, que es un error generado por una función que se llama a sí misma infinitas veces. Para que esto no suceda una función recursiva debe saber resolver el caso más simple, llamado caso base. Si la función es llamada con el caso base, inmediatamente retorna el resultado (no necesita volver a llamarse a sí misma para poder resolverlo). Si la función es llamada con un caso más complejo, las sucesivas llamadas a sí mismas irán virtualmente descomponiendo ese caso hasta llegar a un caso base, para luego determinar el resultado final de la función. A partir de esto se puede esquematizar a una función recursiva mediante el siguiente código:

```
<tipo_retorno> <nombre_función>(<lista_de_parámetros>)
  {
      si <condición_caso_base>
          return <valor_base>
      sino
          <sentencias>
          return <nombre_función>
  }
```

Por ejemplo, para calcular el factorial de un número (n! = n * (n – 1)!) se puede definir la siguiente función recursiva:

```
int factorial(int numero) {
      if (numero == 0)
          return 1;
      else
          return (numero * factorial(numero-1));
  }
```

Comparación entre iteración y recursión

La recursión es una forma de iteración, sin embargo, existen diferencias que son importantes resaltar. La iteración consiste en la repetición de sentencias hasta que se cumple cierta condición que marca la detención del proceso. Si se llama a funciones durante una función iterativa, estas funciones se iniciarán y detendrán en cada ciclo de la iteración. En cambio, en las funciones recursivas, en cada ciclo recursivo se inicia la llamada a una función recursiva, pero no su detención, dado que la función se está invocando a sí misma hasta alcanzar la condición del caso base.

La recursividad permite resolver problemas quizás de forma más simple que mediante funciones iterativas; incluso, hay problemas que inherentemente requieren una solución mediante esta técnica por sus características. Sin embargo, esta técnica generalmente trae un costo asociado en el consumo de memoria dado que se reserva memoria para todas las variables usadas por la función, tantas veces como dicha función sea invocada.

Argumentos de los programas

La función main puede definirse con o sin parámetros. Para que incluya parámetros se suele definir de la siguiente forma:

```
int main(int argc, char * argv[])
```

- argc : indica el número de los argumentos que se utilizaron al ejecutar el programa.
- argv : contiene cada una de las cadenas de caracteres que constituyen esos argumentos.

Al ejecutar cualquier programa siempre se pasa como primer argumento el nombre del mismo.

Un pequeño programa que muestra cómo acceder a los argumentos es el siguiente:

```
int main(int argc, char * argv[])
{
    cout << "Cantidad de argumentos: " << argc << "\n");
    cout << "Argumentos: ";
    for (int i = 0; i < argc; i++)
    cout << argv[i] << " ";
    return 0;
}
```

Un ejemplo de una ejecución del programa anterior (desde la línea de comandos):

```
>ejemplo arg1 arg2 arg3
>Cantidad de argumentos: 4
>Argumentos: ejemplo arg1 arg2 arg3
```

Bibliotecas

Al igual que otros lenguajes de programación, C++ provee un mecanismo para hacer uso de bibliotecas durante el desarrollo de aplicaciones. Una biblioteca es un conjunto de código y datos útiles que brindan servicios a programas independientes. Por ejemplo, hay bibliotecas para el manejo de entrada y salida de datos por consola, para el dibujo de gráficos en 2D y 3D e interfaces de usuario, para el manejo de matrices y funciones matemáticas avanzadas, entre otras utilidades.

Cuando se desarrolla una aplicación en C++, se cuenta con la "biblioteca estándar de C" y con la "biblioteca estándar de C++". Ambas bibliotecas proveen un conjunto de funcionalidad básica como, por ejemplo, funciones para realizar operaciones matemáticas, manejar cadenas de caracteres, realizar entrada y salida de datos a través de archivos y la consola, etc. Sin la biblioteca estándar los programas se verían muy restringidos en utilidad, ya que no se podría realizar ninguna de las funciones básicas mencionadas anteriormente.

Para incluir una biblioteca se utiliza la directiva del compilador **`include`**. Por ejemplo, para extender el programa mínimo e imprimir un mensaje por pantalla, se puede utilizar:

- La biblioteca de streams de entrada/salida de C++ `iostream`, la cual provee el objeto `cout` que cuenta con el operador <<:

```
#include <iostream>

int main()
{
    std::cout << "Hola Mundo!";
    return 0;
}
```

- La biblioteca estándar de C que también incluye funciones para entrada y salida de datos. Por lo tanto, es posible utilizarla para escribir un programa equivalente al anterior, empleando la biblioteca `cstdio` y la función `printf`:

```
#include <cstdio>

int main() {
    printf("Hola Mundo!");
    return 0;
}
```

Directiva `include`

Existen dos formas para especificar el nombre de la bibliteca que será incluida, la única diferencia es el conjunto de directorios en donde el compilador buscará el archivo indicado:

`#include "archivo"`	Si se especifica el archivo entre comillas dobles, primero se buscará el archivo en el mismo directorio en el que se encuentra el archivo fuente que contiene la directiva. Si el archivo no se encuentra ahí el compilador lo busca en los directorios configurados por defecto para buscar los archivos de cabecera estándar.
`#include <archivo>`	Si el nombre del archivo se encierra entre llaves angulares el archivo se busca directamente en los directorios configurados por defecto del compilador.

Espacios de nombres

Los espacios de nombres (*namespaces*) permiten estructurar los programas en unidades lógicas. Cada una de estas unidades puede contener tipos, funciones y objetos agrupados bajo un nombre común.

Separando las funciones en espacios de nombres diferentes se evita que se utilicen los mismos identificadores para definir distintas entidades, dentro del código de un programa y en las bibliotecas que se utilizan.

Existen dos formas de indicar que se va a utilizar una entidad que pertenece a un espacio de nombres determinado:

- Especificar en cada referencia a la entidad a qué espacio de nombres pertenece. Esto se consigue utilizando el operador de ámbito "`::`", y anteponiendo el nombre del espacio al nombre del elemento:

```
<espacio_nombres>::<tipo|función|objeto>
```

- Indicarle al compilador que en una región declarativa se utilizarán entidades pertenecientes a un espacio de nombres determinado, mediante las palabras clave `using namespace`:

```
using namespace <espacio_nombres>;
```

A continuación se presenta un ejemplo utilizando el objeto `cout`, que forma parte de la biblioteca estándar de C++, la cual declara todas sus entidades dentro del espacio de nombres **`std`**.

<table>
<tr><th>Alternativa 1</th><th>Alternativa 2</th></tr>
<tr><td><pre>#include <iostream>

int main() {
    std::cout << "Hola Mundo!";
    return 0;
}</pre></td><td><pre>#include <iostream>

using namespace std;

int main() {
    cout << "Hola Mundo!";
    return 0;
}</pre></td></tr>
</table>

Bibliotecas para las funciones más comunes

La tabla siguiente muestra muy resumidamente los encabezados y funciones/objetos de las bibliotecas estándar de C y C++ para tarea básicas, como la entrada y salida de datos por consola, el manejo de archivos y el manejo de cadenas:

Funciones para	C	C++
Entrada/Salida por consola	**`cstdio`**: `scanf, printf, fgets, ...`	**`iostream`**: `cin, cout, getline, ...`
Manejo de Cadenas	**`cstring`**: `strcpy, strlen, strcat, ...`	**`string`**: `string (+, size, c_str, ...)`
Manejo de archivos	**`cstdio`**: `fopen, fprintf, fwrite, ...`	**`fstream`**: `fstream (open, >>, <<, ...)`

Por convención, cuando se incluye una parte de la biblioteca estándar de C en un programa escrito en C++, se agrega delante del nombre de la biblioteca la letra "c" y no se escribe la extensión .h.

Manejo de entrada/salida por consola

Manejo de la consola en C++

Al incluir el encabezado **`iostream`**, se dispondrá del objeto **`cin`** para realizar operaciones de entrada, del objeto **`cout`** para operaciones salida, y de un conjunto de operadores para interactuar con dichos objetos. A continuación se detalla la forma de uso y operaciones más comunes de cada uno.

Salida por consola

Para esto se utiliza el operador **<<** del objeto `cout`; este objeto emite por pantalla el valor de una variable o constante de cualquier tipo de dato estándar de C++ (caracteres, cadenas, números enteros o reales, etc.), y también movimientos especiales del cursor (por ejemplo, “`\n`” o “`\t`”).

La sintaxis es la siguiente:

```
cout << var;                              cout << cte;
cout << var1 << var2 << ... << varn;      cout << cte1 << cte2 << ... << cten;
cout << var1 << cte1 << var2 << cte2 << ... << varn << cten;
```

Por ejemplo:

```
int num = 512;
cout << "El valor del número es: " << num << "\n";
```

Como se ve, no se indica el tipo de variable que se va a imprimir, por lo tanto, es el sistema el que determina el tipo de variable y la forma más adecuada de imprimirlo.

Entrada por consola

Mediante el operador **>>** del objeto `cin`, se puede leer, con formato, los valores de variables numéricas (`int`, `long`, `float`, `double`, etc), caracteres (`char`) y cadenas de caracteres (`string` y `char*`). Al decir con formato, se hace referencia al hecho de que se aplicarán las transformaciones necesarias para convertir la cadena de caracteres ingresada por la consola, al tipo de la variable pasada como parámetro. La sintaxis es la siguiente:

```
cin >> var;
cin >> var1 >> var2 >> ... >> varn;
```

Al ingresar el texto por la consola se utilizarán los espacios y retornos de línea como separadores. Por ejemplo:

```
int num1, num2;
string cadena;
cin >> num1 >> num2 >> cadena;
```

Cuando se ejecuta este código, al llegar a la línea de `cin`, se pedirá el ingreso de los valores para `num1`, `num2` y `cadena`. Si se ingresa lo siguiente:

```
> 123 456 hola
```

los valores que se asignarán son:

```
num1      →    123
num2      →    456
cadena    →    hola
```

Sincronización de la entrada

Si se utilizan espacios como separadores, se intentará asignar cada parte de la cadena de entrada a una variable. De no ser posible, al no haber suficientes variables en los parámetros de `cin`, el resto de la entrada no asignada será guardada por el objeto `cin`. Como consecuencia, la próxima vez que se invoque

a “`cin >>`” se intentará asignar el resto de la entrada guardada previamente.

Por ejemplo:

Código	Descripción
`string cad1, cad2;`	
`cin >> cad1;`	Se ingresa “hola mundo!”. Se asigna “hola” a cad1.
`cout << cad1 << '\n';`	Se imprime “hola”.
`cin >> cad2;`	Se asigna automáticamente “mundo!” a cad2.
`cout << cad2 << '\n';`	Se imprime “mundo!”.

Para evitar este comportamiento se debe utilizar el comando `cin.ignore()`[5]. El mismo descarta un cierto número de caracteres de la secuencia de entrada o bien descarta todo lo que se encuentre hasta un carácter delimitador. Para corregir el ejemplo anterior es conveniente construir una función que utilice el método `cin.ignore()` de modo que descarte todo lo que se encuentra hasta el final de la línea:

```
#include <iostream>
#include <limits>

void ignorarLinea() {
    cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
}
```

Una vez definida se puede utilizar la función antes o después de cada operación de lectura sobre `cin`:

Código	Descripción
`string cad1, cad2;`	
`cin >> cad1;`	Se ingresa “hola mundo!”. Se asigna “hola” a cad1.
`cout << cad1 << '\n';`	Se imprime “hola”.
`ignorarLinea();`	Se vacía la secuencia de entrada.
`cin >> cad2;`	Se ingresa “otra cadena”. Se asigna “otra” a cad2.
`cout << cad2 << '\n';`	Se imprime “otra”.
`ignorarLinea();`	Se vacía la secuencia de entrada.

Leer líneas completas

Existen dos formas de leer líneas completas a través de `cin`. La primer opción es utilizar la función `getline()` provista por la biblioteca `string`, la cual carga una línea desde `cin` a una cadena del tipo `string`. La segunda opción es utilizar el método `cin.getline()`, que realiza la misma función pero trabaja con cadenas de caracteres del tipo `char*`.

- `getline()`: mediante este método se pude leer desde cualquier stream de entrada una línea completa hasta el fin de línea u otro carácter que se indique. La definición de `getline()` es la siguiente:

  ```
  istream& getline(istream& is, string& str)

  istream& getline(istream& is, string& str, char delim)
  ```

  - “is” es el stream de entrada del cual se extraerá la línea. Generalmente se utilizará el objeto `cin` como parámetro, pero también es posible usar un stream de entrada asociado a un archivo de texto, por ejemplo.
  - “str” es la cadena de caracteres donde se escribirá la línea leída desde “is”;
  - “delim” es el separador que se utilizará. Cuando no se especifica el separador por defecto será el retorno de línea.

[5] A veces se propone utilizar `cin.sync()`, pero esta solución es dependiente de la plataforma que se esté utilizando ya que, en general, no se garantiza que dicha función descartará los caracteres no leídos de la secuencia de entrada. Es por eso que se recomienda utilizar `cin.ignore()` para dicha tarea.

- `cin.getline()`: mediante este método se lee desde la consola una línea completa hasta el fin de línea u otro carácter que se indique. La definición de `cin.getline()` es la siguiente:

```
istream& istream::getline(char* str, streamsize n);

istream& istream::getline(char* str, streamsize n, char delimitador);
```

  - "str" debe ser una cadena de caracteres de un tamaño mayor o igual a "n";
  - "n" será la cantidad máxima de caracteres de la línea que se escribirán en "cadena" (incluyendo el carácter nulo);
  - "delim" es el separador que se utilizará. Cuando no se especifica el separador por defecto será el retorno de línea.

Tanto `getline()` como `cin.getline()` leerán la línea hasta encontrar el separador. En el caso de `cin.getline()` se leerá hasta almacenar la cantidad máxima de caracteres indicada si no se encuentra el delimitador antes. El separador es extraído de la entrada pero no se asigna a la cadena de destino.

Para usar sin inconvenientes cualquiera de las dos funciones (en particular `cin.getline()`) junto con el operador >> de `cin`, es necesario llamar a `cin.ignore()` antes de intentar leer una línea completa. Esto se debe a que luego de utilizar el operador >>, queda guardado en `cin` un separador sin asignar.

Por ejemplo:

```
string cad, linea;
cin >> cad;
ignorarLinea();
getline(cin, linea);
cout << cad << '\n';
cout << linea << '\n';
```

Configuración del formato de la entrada y la salida

Tanto `cin` como `cout` cuentan con una serie de métodos que permiten definir ciertos parámetros del formato del texto que se leerá o escribirá:

`fmtflags flags(fmtflags fmtfl);`	Permite cambiar a través de flags la forma en que se interpreta la entrada o se muestra la salida[6].
`streamsize precision(streamsize prec);`	Especifica cuántos dígitos se mostrarán luego del punto decimal.
`streamsize width(streamsize wide);`	Especifica el número mínimo de caracteres que se mostrará al escribir un valor. Si la representación del valor de salida es menor que el ancho del campo de salida el mismo se completará con caracteres de relleno (espacios por defecto).

Manejo de la consola en C

Las funciones para entrada y salida por consola de la biblioteca de C se encuentran definidas en el encabezado **`cstdio`**.

La principal diferencia que existe al realizar entrada/salida por consola en C, es que las funciones que se utilizan para esto necesitan que se especifique el tipo de valor de la variable que se imprimirá o cargará por consola. Esta información se indica a través de una "cadena de formato", la cual contiene dos tipos de objetos: **caracteres ordinarios**, que son copiados textualmente a la salida y **especificaciones de conversión**, cada uno de los cuales causa la conversión de los argumentos sucesivos. Cada especificación de conversión comienza con un **%** y termina con un carácter de conversión. Entre el % y el carácter de conversión puede haber otros indicadores[7].

---

[6] Para acceder la lista completa de flags ver: *http://www.cplusplus.com/reference/iostream/ios_base/fmtflags.html*

[7] Para ver la lista completa de caracteres de conversión y el resto de los especificadores de la cadena de formato ver: *http://www.cplusplus.com/reference/clibrary/cstdio/printf.html* *http://cplusplus.com/reference/clibrary/cstdio/fscanf.html*

Caracteres de conversión (lista parcial)	**Significado**
`d, i`	Entero decimal
`f, e, E, g, G`	Punto flotante
`h`	Entero corto
`c`	Carácter
`s`	Cadena de caracteres (`char*`)

Salida por consola

Para imprimir datos en la consola se utiliza la función **`printf`**:

```
int printf(const char * format, ...)
```

Esta función convierte, da formato e imprime sus argumentos en la salida estándar, de acuerdo a las especificaciones de conversión que se pueden encontrar en la cadena de formato. En el siguiente ejemplo, se puede ver que cada uno de los valores de las variables deberá ser convertido en un valor entero decimal.

```
int dia = 1, mes = 1, año = 1999;

printf("%d/%d/%d", dia, mes, año);
```

Entrada por consola

Para leer valores ingresados desde la consola se utiliza la función **`scanf`**:

```
int scanf(const char * format, ...);
```

Esta función lee caracteres de la entrada estándar, los interpreta de acuerdo con las especificaciones que están en la cadena de formato, y almacena los resultados a través de los argumentos restantes, donde cada uno de estos argumentos debe ser una referencia[8].

Por ejemplo, se pueden leer fechas ingresadas desde la entrada con el formato <int>/<int>/<int>, mediante la utilización de `scanf` como sigue:

```
int dia, mes, año;
scanf("%d/%d/%d", &dia, &mes, &año);
```

Manejo de cadenas

Cadenas de C++

Para el manejo de cadenas de caracteres C++ agrega una clase llamada `string`, la cual se encuentra definida en la biblioteca `string`. Las principales ventajas de esta representación de las cadenas de caracteres son el carácter dinámico del tamaño de las mismas, y la posibilidad de realizar muchas de las funciones más comunes, como la comparación y la concatenación, utilizando los operadores clásicos (==, !=, >, <, +, etc). Además, las cadenas de C++ pueden ser indexadas para acceder a cada carácter en forma individual utilizando el operador [] (como si fuesen arreglos comunes).

Otra ventaja es la integración de la clase `string` con los streams de entrada/salida (por consola y archivos), por ejemplo:

```
string cadena;
cin >> cadena;
cout << cadena;
getline(cin, cadena);
cout << cadena;
```

[8] Ver la sección de "Punteros y manejo de memoria".

Por ejemplo:

```
for(int i = 0; i < cadena.length(); i++) {
    cout << cadena[i];
}
```

Algunas funciones útiles de la clase `string`[9]:

`size_t lenght() const;` `size_t size() const;`	Devuelve el número de caracteres de la cadena.

`size_t find(const string& patron, size_t posicion = 0) const;`
`size_t find(const char& patron, size_t posicion = 0) const;`
`size_t find(char c, size_t posicion = 0) const;`
	Devuelve la posición de la primer ocurrencia del patrón o el valor especial `npos` si no es encontrado. El parámetro de la posición puede utilizarse para indicar a partir de qué lugar debe empezar la búsqueda, pero el mismo puede omitirse.

`string substr(size_t posicion = 0, size_t longitud = npos) const;`	Devuelve una subcadena de acuerdo a los valores enteros pasados.
`const char* c_str() const;`	Devuelve el valor del string como una cadena C terminada por el carácter nulo.

Cadenas de C

En C las cadenas de caracteres se representan a partir de arreglos de caracteres (también llamados c-string). Así, se puede definir una cadena de caracteres que permita almacenar hasta 50 caracteres mediante:

```
char cadena[50];
```

Debido a que el arreglo de caracteres puede almacenar cadenas de longitud menor a 50, es necesario contar con algún carácter especial que determine el fin de dicha cadena. Para esto se utiliza el carácter nulo '\0'. Si el arreglo no incluye este carácter al final no se considerará como una cadena de caracteres, sino como un simple arreglo de caracteres. Esto es muy importante, principalmente cuando se utilizan las funciones que define la biblioteca estándar de C para trabajar con cadenas.

Un error muy frecuente es intentar copiar con el operador de asignación (`cadena1 = cadena2`). Esto no es válido, debido a que no está permitida la copia por asignación de arreglos (o bloques de memoria).

Inicialización de las cadenas de caracteres

Al igual que los arreglos de otros tipos de datos, las cadenas de caracteres pueden ser inicializadas durante su declaración.

```
char palabra [5] = {'H', 'o', 'l', 'a', '\0'};
char palabra [] = {'H', 'o', 'l', 'a', '\0'};
```

Como se puede ver, el último carácter debe ser utilizado para almacenar el carácter nulo, por lo tanto, si el tamaño del arreglo es **n**, el espacio disponible para caracteres significativos será **n-1**.
Pero las cadenas de caracteres también pueden ser inicializados mediante el uso de cadenas literales, es decir:

```
char cadena[] = "Hola";
```

En este último caso no es necesario agregar el carácter '`\0`', ya que se añade por defecto a todos las cadenas de literales. El arreglo resultante tendrá un tamaño de 5 caracteres (4 caracteres de la palabra más el carácter nulo).

Para cargar una cadena de caracteres por posición, se puede realizar lo siguiente:

```
char cadena[20];
cadena[0] = 'a';
cadena[1] = 'b';
cadena[2] = '\0';
```

[9] Para ver la lista completa de métodos de la clase `string` visitar: *http://www.cplusplus.com/reference/string/string/*

Además de los arreglos, también es posible utilizar cualquier secuencia de caracteres para representar las cadenas de C. Es por esto que el tipo asociado a las cadenas es un puntero a un bloque de caracteres, `char*`, donde el valor del mismo indica la posición del primer carácter. Una última consideración respecto a lo anterior es que utilizando esta definición para un parámetro, si bien no se podrá modificar a la variable para que referencie a otra cadena, aún será posible modificar su contenido. Es por eso que para los parámetros que presenten referencias a cadenas de sólo lectura se deberá utilizar la definición `const char*`.

Operaciones de entrada/salida por consola

Para mostrar una cadena de caracteres, se puede utilizar:

```
cout << cadena;
```

o bien

```
printf("%s", cadena);
```

Ambas funciones mostrarán todos los caracteres situados a partir de la dirección de inicio hasta encuentrar el carácter '\0', el cual no se imprime.

Para leer una cadena de C desde la consola hay que considerar el carácter estático de las secuencias de caracteres utilizadas para implementarlos. Esta situación representa una complicación (respecto a los strings de C++ que son dinámicos) ya que el procedimiento de lectura podría sobrepasar su capacidad durante la carga. Es por eso que se recomienda indicar siempre el tamaño del arreglo a las funciones de lectura. Por ejemplo:

```
cin.getline(cadena, 20);
```

También se puede definir el número máximo de caracteres a leer a través del indicador de ancho de la cadena de formato de `scanf()` "**`%[ancho][tipo]`**" o bien utilizar la función `fgets()`:

```
scanf("%20s", cadena);
fgets(cadena, 20, stdin);  // Función definida en la biblioteca cstdio
```

Funciones para el manejo de cadenas

La biblioteca de `cstring` proporciona muchas funciones útiles para el manejo de cadenas de C, como la comparación, copia, concatenación y búsqueda de cadenas:

`int strcmp(const char * str1, const char * str2, size_t num);`	Compara hasta `num` caracteres de las cadenas `str1` y `str2`. Devuelve un valor entero indicando la relación entre las cadenas: ● negativo indica que `str1` es menor que `str2` ● cero indica que las cadenas son iguales ● positivo indica que `str1` es mayor a `str2`
`char * strcat(char * dest, const char * src, size_t num);`	Añade los primeros `num` caracteres de la cadena `src` al final de la cadena `dest` incluyendo un carácter nulo para la terminación. Se supone que la cadena de destino es lo suficientemente grande para contener el resultado.
`char * strcpy(char * dest, const char * src, size_t num);`	Copia los primeros `num` caracteres de la cadena `src` en la cadena `dest` incluyendo un carácter nulo para la terminación. Si `num` es mayor a la longitud de `src` se escribirá en dest una cantidad de ceros igual a la diferencia.
`size_t strlen(const char * s);`	Devuelve la longitud de la cadena, menos el carácter de terminación.
`char * strcmp(char * str1, const char * str2,`	Devuelve un puntero a la primera posición de la ocurrencia de `str2` en `str1`, o un puntero nulo en el caso de que `str2` no sea parte de `str1`.

A su vez, la biblioteca `cstlib` cuenta con funciones para realizar conversiones de cadenas a números:

`int atoi(const char * str);`	Interpreta el contenido de la cadena `str` como si fuese un número entero y retorna su valor como `int`.
`long int atoi(const char * str);`	Interpreta el contenido de la cadena `str` como si fuese un número entero y retorna su valor como `long int`.
`double atof(const char * str);`	Interpreta el contenido de la cadena `str` como si fuese un número de punto flotante y retorna su valor como `double`.

Por último, la biblioteca `cstdio` provee varias funciones para entrada/salida generales como `printf` y `scanf` las cuales pueden utilizarse con cadenas de C; así como algunas funciones específicas para cadenas como `gets` y `fgets`, de las cuales se recomienda utilizar la última ya que permite especificar el tamaño de la secuencia, lo que la hace más segura. Además, la biblioteca también incluye algunas funciones para manipular cadenas con formato, que también pueden utilizarse al convertir números a cadenas y viceversa.

`char * fgets(char * str, int num, FILE * stream);`	Lee desde el `stream` (`sdtin`, `stdout`, un archivo, etc) una cadena hasta que se alcance el número máximo de caracteres (`num`-1) o bien un salto de línea o el final del archivo.
`int sprintf(char * str, const char * format, ...);`	Escribe valores en una cadena con formato en forma similar a `printf()`.
`int sscanf(const char * str, const char * format, ...);`	Interpreta los valores de una cadena con formato en forma similar a `scanf()`.

Manejo de archivos

Streams de archivos de C++

C++ provee el mecanismo de *streams* para realizar operaciones de entrada/salida. Los *streams* son una abstracción de los dispositivos en los cuales se puede escribir o extraer datos; su implementación, en C++, los define como una fuente o destino de caracteres de longitud indefinida. Además, pueden estar asociados a diversos dispositivos físicos como la consola y archivos en disco.

Por ejemplo, los objetos `cin` y `cout` son instancias de los tipos `istream` y `ostream` respectivamente, es decir son *streams* de entrada y salida, los cuales están asociados con la consola (en particular, la entrada y salida estándar). En el caso de archivos, existen especializaciones de los *streams* de entrada y salida para poder vincularlos con archivos y así poder leer y escribir en ellos. La forma de trabajar con archivos resulta bastante similar al modo en el que se realiza la entrada y salida por consola (con los objetos `cin` y `cout`) ya que los streams de archivos están definidos a partir de las mismas clases base.

Para manipular archivos a través de *streams* se puede utilizar alguna de las siguientes clases definidas en el encabezado **`fstream`**:

- **`ifstream`**: para realizar operaciones de lectura desde archivos.
- **`ofstream`**: para realizar operaciones de escritura en archivos.
- **`fstream`**: para realizar operaciones de lectura y escritura en archivos.

A continuación se describirán las operaciones específicas para el manejo de archivos, que ofrecen estos tipos *streams*.

Para vincular un *stream* con un archivo se puede utilizar el método `open()` (que se encuentra definido en los tres tipos):

```
void open(const char * filename, ios_base::openmode mode);
```

También es posible asociar el *stream* con un archivo durante la construcción, por ejemplo:

```
explicit fstream(const char * filename, ios_base::openmode mode);
```

- El parámetro `filename` se utiliza para indicar la ruta completa del archivo. Si el archivo se encuentra en el mismo directorio que el programa, simplemente se indica el nombre del archivo deseado.

- El parámetro `mode` es un parámetro opcional, el cual puede contener una combinación de los siguientes valores:

`ios::in`	Abre un archivo existente permitiendo operaciones de entrada (**in**put).
`ios::out`	Abre (o crea en el caso de que no exista) un archivo permitiendo operaciones se salida (**out**put).
`ios::binary`	Configura el stream para que trabaje en modo binario (**binary**), en vez de texto.
`ios::ate`	Configura el stream para que antes de cada operación de salida el indicador de posición se sitúe al final del mismo (**at e**nd).
`ios::app`	Configura el stream para que el indicador de posición se sitúe al final del mismo, de modo que la nueva información se añadirá a la ya existente (**app**end).
`ios::trunc`	Se elimina el contenido del archivo llevando su longitud a 0 (**trunc**ate).

Los valores se pueden combinar utilizando el operador de bits OR (|).

Por ejemplo:

```
ofstream archivo;
archivo.open ("ejemplo.bin", ios::out | ios::app | ios::binary);
```

La función `open()` tiene un modo por defecto que se utiliza cuando se abre un archivo sin el segundo parámetro. El modo por defecto depende de la clase de stream para archivos:

Clase	**Parámetro de modo por defecto**
`ofstream`	`ios::out`
`ifstream`	`ios::in`
`fstream`	`ios::in	ios::out`

Para verificar que el archivo se abrió correctamente se utiliza el método `is_open()`:

```
bool is_open();
```

Para cerrar el archivo y terminar de escribir las salidas pendientes se usa el método `close()`:

```
void close();
```

Para verificar si se llegó al final de un archivo durante la lectura se puede utilizar la función `eof()`:

```
bool eof();
```

Un problema frecuente es que `eof()` no indica que se llegó al final del archivo hasta que efectivamente se intente leer un dato luego de la última lectura correcta. Esto significa que en el caso general, el código siguiente procesará los datos de una última lectura no válida:

```
while (!archivo.eof()) {
    archivo >> datos;
    // Operación sobre los datos
    // ...
}
```

Una forma de reestructurar el código para evitar este problema es realizando la lectura de los datos antes de invocar a `eof()`:

```
archivo >> datos;
while (!archivo.eof()) {
    // Operación sobre los datos
    // ...
    archivo >> datos;
}
```

Cabe aclarar que este comportamiento es independiente del tipo de archivo (de texto o binario) y por lo tanto el mismo se exhibe sin importar que método de lectura o escritura se utilice.

Por último resta mencionar que los streams de archivos cuentan con la posibilidad de ser accedidos en forma aleatoria para leer o escribir datos en cualquier posición. Existen indicadores de posición para la entrada y salida de datos que pueden ser consultadas a través de las funciones `tellp()` y `tellg()` y también pueden modificarse con los métodos `seekp()` y `seekg()`.

Archivos de texto

En el caso de los archivos de texto, la salida se realiza a través del operador <<, del mismo modo que con el objeto `cout`. La lectura de los datos se realiza con el operador >>, de forma equivalente a como se opera sobre la secuencia de entrada de `cin`. También son válidos todos los modificadores y métodos para dar formato a la entrada y la salida de los streams asociados a la consola.

A continuación se presentan las definiciones comunes a una serie de ejemplos que se desarrollarán en lo que resta de esta sección:

```
#include <iostream>
#include <limits>
using namespace std;

const int LONG_STRING = 50;
const int EMPLEADOS = 100;

struct Empleado {
    int id;
    char nombre[LONG_STRING];
    char apellido[LONG_STRING];
    float salario;
};

struct Empleados {
    Empleado empleados[EMPLEADOS];
    int cantidad;
};

void ignorarLinea() {
    cin.ignore(numeric_limits<std::streamsize>::max(), '\n');
}

void cargarEmpleados(Empleados & emp) {
    int id = 1;
    while ((id != 0) && (emp.cantidad < EMPLEADOS)) {
        cout << "Ingresar id (0 para terminar):\n";
        cin >> id;
        ignorarLinea();
        if (id != 0) {
            int pos = emp.cantidad;
            emp.empleados[pos].id = id;
            cout << "Ingresar nombre:\n";
            cin.getline(emp.empleados[pos].nombre, LONG_STRING, '\n');
            cout << "Ingresar apellido:\n";
            cin.getline(emp.empleados[pos].apellido, LONG_STRING, '\n');
            cout << "Ingresar salario:\n";
            cin >> emp.empleados[pos].salario;
            ignorarLinea();
            emp.cantidad++;
        }
    }
}
```

```
void mostrarEmpleados(const Empleados & emp) {
    for (int i = 0; i < emp.cantidad; i++) {
        cout << emp.empleados[i].id << ", ";
        cout << emp.empleados[i].nombre << ", ";
        cout << emp.empleados[i].apellido << ", ";
        cout << emp.empleados[i].salario << "\n";
    }
}
```

Ejemplo de escritura:

```
void guardarEmpleadosTxt(ofstream & arch, const Empleados & emp) {
    for (int i = 0; i < emp.cantidad; i++) {
        arch << emp.empleados[i].id << "\n";
        arch << emp.empleados[i].nombre << "\n";
        arch << emp.empleados[i].apellido << "\n";
        arch << emp.empleados[i].salario << "\n";
    }
}

int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    ofstream archSalida("prueba.txt", ios::trunc);
    if (archSalida.is_open()) {
        cargarEmpleados(empleados);
        guardarEmpleadosTxt(archSalida, empleados);
        archSalida.close();
    } else
        cout << "Error al crear el archivo de salida\n";

    return 0;
}
```

Ejemplo de lectura:

```
void cargarEmpleadosTxt(ifstream & arch, Empleados & emp) {
 int i = 0;
 int id;
    arch >> id;
    while (!arch.eof() && (i < EMPLEADOS)) {
       emp.empleados[i].id = id;
        arch >> emp.empleados[i].nombre;
        arch >> emp.empleados[i].apellido;
        arch >> emp.empleados[i].salario;
        i++;
        arch >> id;
    }
    emp.cantidad = i;
}

int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    ifstream archEntrada("prueba.txt");
    if (archEntrada.is_open()) {
        cargarEmpleadosTxt(archEntrada, empleados);
        mostrarEmpleados(empleados);
        archEntrada.close();
    } else
        cout << "Error al abrir el archivo de entrada\n";

    return 0;
}
```

Notar que en todos los casos el objeto stream se debe pasar por referencia a las funciones, ya que los streams no pueden copiarse.

Archivos binarios

Es posible utilizar los *streams* de archivo para trabajar con archivos binarios. Si bien las secuencias de entrada y salida se siguen manejando como secuencias de caracteres, se proveen una serie de funciones que permiten leer y escribir los datos sin que se aplique ningún formato, lo que permite guardar y recuperar los datos como secuencias de bytes.

Para trabajar con *streams* de archivos binarios se debe utilizar el indicador `ios::binary` para el parámetro de modo, al momento de abrir el archivo.

Para escribir datos en los streams se utiliza el método `write()`:

```
ostream & write(const char * s, streamsize n);
```

Donde:

- "s" es una referencia a una variable o secuencia de cualquier tipo que debe ser convertida a una referencia a secuencia de caracteres. Para la conversión puede utilizarse una conversión explícita de C `(char*)` o bien el operador de conversión `reinterpret_cast<const char*>()` de C++[10] (que se muestra en en los ejemplos);
- "n" indica el tamaño en bytes (o caracteres) de la variable o secuencia a escribir. Para obtener el tamaño en bytes de un tipo o una variable se utiliza el operador `sizeof()`,

Para leer datos se utilizar el método `read()`:

```
istream & read(char * s, streamsize n);
```

Donde:

- "s" es una referencia a una variable o bloque de datos de cualquier tipo;
- "n" indica el tamaño en bytes (o caracteres) de la variable o secuencia a leer.

Ejemplo de escritura:

```
void guardarEmpleadosBin(ofstream & arch, const Empleados & emp) {
        // Primero se guarda la cantidad de empleados
    arch.write(reinterpret_cast<const char *>(&emp.cantidad), sizeof(int));
        // Luego se guarda la parte cargada del arreglo de empleados
    arch.write(reinterpret_cast<const char *>(emp.empleados), sizeof(Empleado) *
      emp.cantidad);
}

int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    ofstream archSalida("prueba.bin", ios::binary|ios::trunc);
    if (archSalida.is_open()) {
        cargarEmpleados(empleados);
        guardarEmpleadosBin(archSalida, empleados);
        archSalida.close();
    } else
        cout << "Error al crear el archivo de salida\n";

    return 0;
}
```

[10] Para ver una explicación y ejemplos de uso de los distintos mecanismos y operadores de conversión de tipos, ver: *http://cplusplus.com/doc/tutorial/typecasting.html*

Ejemplo de lectura:

```
void cargarEmpleadosBin(ifstream & arch, Empleados & emp) {
        // Primero se lee la cantidad de empleados
    arch.read(reinterpret_cast<char *>(&emp.cantidad), sizeof(int));
    // Luego se cargan los empleados en el arreglo
    arch.read(reinterpret_cast<char *>(emp.empleados), sizeof(Empleado) *
      emp.cantidad);
}
int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    ifstream archEntrada("prueba.bin", ios::binary);
    if (archEntrada.is_open()) {
        cargarEmpleadosBin(archEntrada, empleados);
        mostrarEmpleados(empleados);
        archEntrada.close();
    } else
        cout << "Error al abrir el archivo de entrada\n";
    return 0;
}
```

Como puede observarse en ambos ejemplos, tanto para leer como escribir un arreglo o bloque de variables, sólo hace falta realizar el cálculo de **<tamaño celda>*<cantidad celdas>**, para obtener el tamaño total en bytes de la secuencia.

Strings en archivos binarios

El tipo básico *String* de C++, no puede ser almacenado de la misma forma que el resto de los tipos básicos. Esto se debe a que los strings son estructuras dinámicas, si se obtiene el espacio utilizado (mediante `sizeof`) de un string vacio y un string con varias lineas de caracteres, se verá que el resultado es el mismo.

Por lo tanto, para poder almacenar strings en archivos binarios, se debe aplicar el mismo criterio que se utiliza para almacenar información dinámica, es decir, primero se almacena la cantidad de datos a almacenar y luego los datos en sí.

Ejemplo de escritura:

```
void guardarStringBin(ofstream & arch, const string & cadena) {
    int longitud = cadena.length();
    // Primero se almacena la longitud de la cadena
    arch.write(reinterpret_cast<const char *> (&longitud), sizeof(int));
    // Luego se almacena la cadena (en este caso el resultado de
    // cadena.c_str() ya es const char *, por lo tanto se evita la
    // conversión mediante reinterpret_cast<const char *>)
    arch.write(cadena.c_str(), sizeof(char) * longitud);
}
```

Ejemplo de lectura:

```
void cargarStringBin(ifstream & arch, string & cadena) {
    int longitud;
    // Primero se lee la longitud de la cadena
    arch.read(reinterpret_cast<char *> (&longitud), sizeof(int));
    // Se define una cadena de C para leer los caracteres. Se define de
    // longitud + 1, para completar la cadena con '\0'
    char arrCadena[longitud + 1];
    // Se lee la cadena
    arch.read(arrCadena, sizeof(char) * longitud);
    // Se completa el arreglo con '\0' para formar una cadena de caracteres
    // estilo C
    arrCadena[longitud] = '\0';
    // Ahora se puede asignar la cadena de caracters al string
    cadena = arrCadena;
}
```

Streams de archivos de C

C++ provee la implementación de streams original de C, la cual puede utilizarse a través de la biblioteca **cstdio**. Existe un único tipo de stream en C definido por el tipo `FILE`, el cual contiene la secuencia de entrada y salida e información, como la posición a la que se está accediendo, el estado, etc.

Para abrir archivos utilizamos la función `fopen`:

```
FILE * fopen(const char * filename, const char * mode);
```

El parámetro `filename` se utiliza para indicar la ruta hasta el archivo, incluyendo el nombre de dicho archivo. Si el archivo se encuentra en el mismo directorio donde se genera el ejecutable de nuestro programa, simplemente se indica el nombre del archivo deseado.

La cadena de modo (parámetro `mode`) puede contener una combinación de los siguientes indicadores:

r : abre un archivo existente para lectura;

w : crea el archivo o vacía un archivo existente, dejándolo listo para escritura;

a : crea el archivo o mantiene el contenido de uno existente añadiendo los nuevos datos al final.

Seguido de alguno de los modificadores:

+ : agrega escritura al modo `r` o lectura a los modos `w` y `r`.

b : hace que los archivos sean binarios (por defecto son de texto).

Si se pudo abrir el archivo correctamente, la función `fopen()` devuelve una referencia al stream (`FILE*`), o bien un puntero nulo (`0` o `NULL`) en el caso contrario.

Para cerrar un archivo se utiliza la función `fclose()`:

```
int fclose(FILE * stream);
```

Para verificar si se llegó al final de un stream, durante la lectura, se utiliza la función `feof()`:

```
int feof(FILE * stream);
```

Al igual que con los streams de C++ y el método `eof()`, esta función no retornará el valor de fin de archivo (`0` o `EOF`) hasta que se realice una operación de lectura que lo encuentre. Es por eso que, en general, el código que sigue realizará una operación sobre datos no válidos en la última iteración:

```
while (feof(archivo) != EOF) {
    fscanf(arch, "%...", &datos);
    // Operación sobre los datos
    // ...
}
```

Como se comentó anteriormente se puede resolver el problema realizando la lectura de datos antes de llamar a `feof()` y luego procesar los datos. A su vez existe otra solución que consiste en utilizar el valor devuelto por las funciones de lectura de streams de C (`fscanf`, `fread`, etc), el cual indica el número de elementos leídos, para controlar el corte del ciclo (por lo que ya no resulta necesario utilizar `feof()`):

```
while (fscanf(arch, "%...", &datos) != 0) {
    // Operación sobre los datos
    // ...
}
```

Archivos de texto

Cuando se trabaja con archivos de texto se utilizan funciones similares a las ya vistas para el manejo de entrada y salida por consola (`prinf()`, `scanf()`, `fgets()`, etc), varias de las cuales utilizan el mecanismo de la cadena de formato para realizar la interpretación y formateo de los valores de los distintos tipos de información que se lee y escribe. A diferencia de las funciones que se encuentran

asociadas con los streams de la entrada y salida estándar de la consola (`stdin` y `stdout`), estas funciones introducen un parámetro adicional para indicar el stream sobre el cual se desea operar.

Para escribir datos en un stream (de archivo) se utiliza la función `fprint()` que opera igual que `prinf()`:

```
int fprintf(FILE * stream, const char * format, ...);
```

Para leer datos se utiliza la función `fscanf()` que es similar a `scanf()`:

```
int fscanf(FILE * stream, const char * format, ...);
```

Al igual que con los streams de C++ existe la posibilidad de realizar accesos de forma aleatoria a cualquier posición del stream de archivo tanto para escribir como para leer datos. Esto se consigue manipulando el indicador de posición del stream. Las funciones `fgetpos()` y `ftell()` devuelven la posición actual del indicador. Las funciones `fsetpos()` y `fseek()` modifican el indicador llevándolo a la posición deseada.

Ejemplo de lectura:

```
void guardarEmpleadosTxt(FILE * arch, const Empleados & emp) {
    for (int i = 0; i < emp.cantidad; i++) {
        fprintf(arch, "%d\n", emp.empleados[i].id);
        fprintf(arch, "%s\n", emp.empleados[i].nombre);
        fprintf(arch, "%s\n", emp.empleados[i].apellido);
        fprintf(arch, "%f\n", emp.empleados[i].salario);
    }
}
int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    FILE * archSalida = fopen("prueba.txt", "w");
    if (archSalida != 0) {
        cargarEmpleados(empleados);
        guardarEmpleadosTxt(archSalida, empleados);
        fclose(archSalida);
    } else
        cout << "Error al crear el archivo de salida\n";

    return 0;
}
```

Ejemplo de escritura:

```
void cargarEmpleadosTxt(FILE * arch, Empleados & emp) {
    // Se genera dinámicamente el formato para leer las cadenas limitando
    // la cantidad de caracteres
    char stringFormat[50];
    sprintf(stringFormat, "%%%ds\n", LONG_STRING);

    int i = 0;
    int id;
    fscanf(arch, "%d\n", &id);
    while ((feof(arch) != EOF) && (i < EMPLEADOS)) {
       emp.empleados[i].id = id;
       fscanf(arch, stringFormat, &emp.empleados[i].nombre);
       fscanf(arch, stringFormat, &emp.empleados[i].apellido);
        fscanf(arch, "%f\n", &emp.empleados[i].salario);
        i++;
        fscanf(arch, "%d\n", &id);
    }
    emp.cantidad = i;
}
```

```
int main() {
    Empleados empleados;
    empleados.cantidad = 0;

    FILE * archEntrada = fopen("prueba.txt", "r");
    if (archEntrada != 0) {
        cargarEmpleadosTxt(archEntrada, empleados);
        mostrarEmpleados(empleados);
        fclose(archEntrada);
    } else
        cout << "Error al abrir el archivo de entrada\n";
    return 0;
}
```

Archivos binarios

Para manipular archivos binarios hay que incluir el modificador "**b**" a la cadena de modo de la función `fopen()`.

Para leer del archivo se utiliza la función `fread()`:

```
size_t fread(void * ptr, size_t size, size_t count, FILE * stream);
```

donde `ptr` es un puntero a la variable o bloque donde se van a almacenar los datos leídos del archivo, `size` es el tamaño en bytes de cada elemento a leer, `count` es la cantidad de elementos que se van a leer y `stream` es el archivo de donde van a ser leídos.

Esta función devuelve el número de elementos leídos; dicho valor puede ser comparado con el parámetro `count`; sí los valores son distintos, se puede asegurar que ocurrió un error en la lectura.

Para escribir en el archivo se utiliza la función `fwrite()`:

```
size_t fwrite(const void * ptr, size_t size, size_t count, FILE * stream);
```

donde `ptr` es un puntero a la variable donde se encuentran los elementos a guardar en el archivo. En este caso se agrega el modificador `const` para indicar que el contenido de dicha variable no va a ser modificado por la función `fwrite`. El parámetro `size` es el tamaño en bytes de cada elemento a escribir, `count` es la cantidad de elementos que se van a escribir y `stream` es el archivo donde van a ser escritos.

Esta función devuelve el número de elementos escritos en el archivo; dicho valor puede ser comparado con el parámetro `count`, sí los valores son distintos se puede asegurar que ocurrió un error en la escritura.

Ejemplo de escritura:

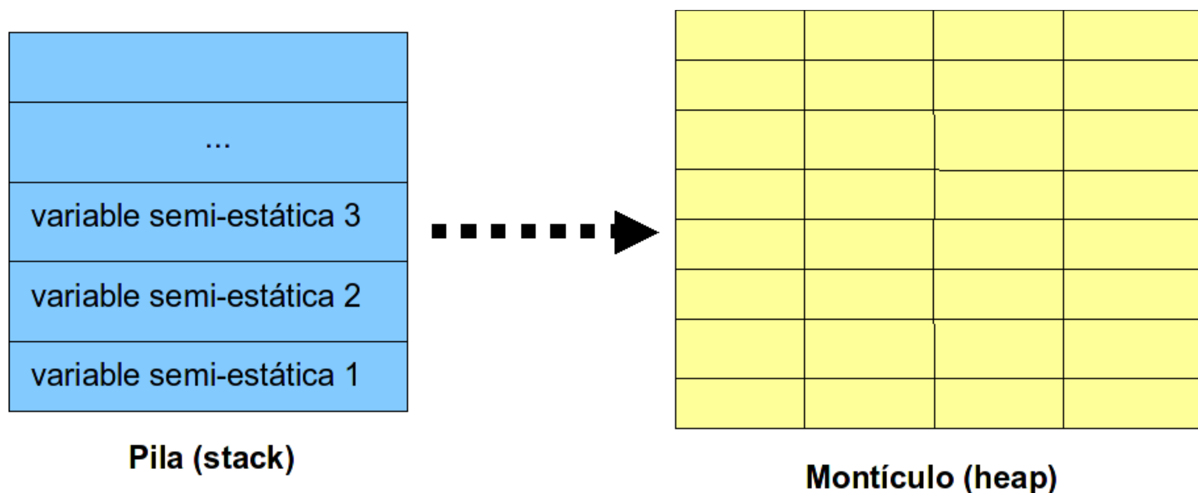
```
void guardarEmpleadosBin(FILE * arch, const Empleados & emp) {
    // Primero se guarda la cantidad de empleados
    fwrite(&emp.cantidad, sizeof(int), 1, arch);
    // Luego se guarda la parte cargada del arreglo de empleados
    fwrite(emp.empleados, sizeof(Empleado), emp.cantidad, arch);
}
int main() {
    Empleados empleados;
    empleados.cantidad = 0;
    FILE * archSalida = fopen("prueba.bin", "wb");
    if (archSalida != 0) {
        cargarEmpleados(empleados);
        guardarEmpleadosBin(archSalida, empleados);
        fclose(archSalida);
    } else
        cout << "Error al crear el archivo de salida\n";
    return 0;
}
```

Ejemplo de lectura:

```
void cargarEmpleadosBin(FILE * arch, Empleados & emp) {
    // Primero se lee la cantidad de empleados
    fread(&emp.cantidad, sizeof(int), 1, arch);
    // Luego se cargan los empleados en el arreglo
    fread(emp.empleados, sizeof(Empleado), emp.cantidad, arch);
}

int main()
{
    Empleados empleados;
    empleados.cantidad = 0;

    FILE * archEntrada = fopen("prueba.bin", "rb");
    if (archEntrada != 0) {
        cargarEmpleadosBin(archEntrada, empleados);
        mostrarEmpleados(empleados);
        fclose(archEntrada);
    } else
        cout << "Error al abrir el archivo de entrada\n";

    return 0;
}
```

Punteros y manejo de memoria

C++ maneja varias regiones de memoria durante la ejecución de un programa. La discusión de esta sección se centra en dos de ellas, en las cuales el programador puede manipular sus variables: la pila y el montículo.



La **pila** se utiliza para almacenar información de las sub-rutinas activas durante la ejecución (funciones, *variables de tamaño fijo*, puntos de retorno, etc). Las variables de la pila cuentan con un nombre y un tipo y pueden declararse en cualquier lugar de un bloque de código. Es el sistema de ejecución el que se encarga de reservar el espacio para estas variables, así como inicializarlas y destruirlas (liberando el espacio reservado) cuando se termina la ejecución del bloque en el que fueron definidas.

El **montículo** se utiliza para almacenar *información de tamaño variable* durante la ejecución. Las variables del heap tienen tipo pero no cuentan con un nombre. Por lo tanto, se manipulan a través de otras variables denominadas punteros. Las variables que se encuentran en el montículo son administradas explícitamente por el programador, por lo que es este quién debe reservar el espacio necesario para las mismas, inicializarlas y destruirlas a través de los punteros y sus operaciones especiales.

Los **punteros** (también denominados “apuntadores” o “referencias”) son un tipo de variable que contiene la dirección de memoria de otras variables en el montículo o en la pila. Los punteros en general se definen de forma que puedan referenciar un tipo determinado de variables. No existe un término preferido para denominar a “lo que apunta” un puntero, aunque suele utilizarse “referente”.

Declaración de los punteros:

```
<tipo_objeto> * <etiqueta_puntero>;
```

Como se dijo anteriormente los valores de los punteros son direcciones de memoria. La asignación es la operación con punteros válida más básica, la cual permite copiar en el puntero la dirección contenida en otro puntero o la obtenida a través de alguno de los operadores de referencia y reserva de memoria que se verán continuación.

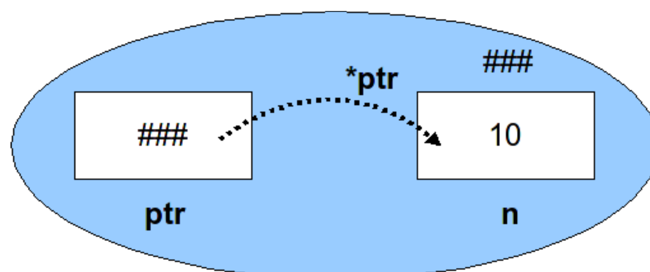
Para iniciar un puntero que no hace referencia a un espacio de memoria válido es necesario asignar el valor constante 0 o bien el valor de la macro `NULL` (definida en la biblioteca `cstdlib`).

Operadores relacionados con punteros

`*<puntero>`	Es el operador de indirección. El mismo permite acceder al valor del objeto referenciado (para leerlo o modificarlo). El operador de indirección también suele llamarse operador de “desreferencia”, y se llama “desreferenciar” a la acción de acceder, a través del mismo, al contenido de un espacio en memoria.
`<puntero>-><miembro>`	Es el operador de acceso a miembros. El mismo realiza una indirección que permite acceder a los campos de una estructura o los miembros de una clase. Es equivalente a escribir `(*<puntero>).<miembro>`.
`&<variable>`	Es el operador de referencia el cual devuelve la dirección en memoria de un objeto.

Por ejemplo:

```
int n;           // Declaramos una variable entera
int * ptr;       // Declaramos un puntero a variables de tipo entero
ptr = &n;        // Asignamos al puntero la dirección de memoria de la variable n (notar
                 que esta dirección se encuentra en la pila)
*ptr = 10;       // Modificamos el contenido de la variable referenciada por el puntero
```



C++ también cuenta con los operadores `new` y `delete` que permiten crear y destruir variables a través de punteros. Estos operadores se explicarán a continuación junto con las funciones de C que cumplen el mismo propósito.

Operaciones para obtener y liberar memoria

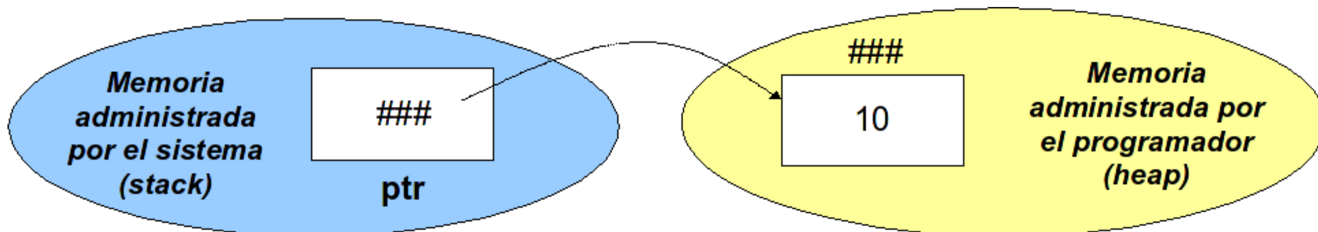
Operaciones para obtener memoria

Existen dos operaciones para obtener memoria: `new` (introducido en C++) y `malloc` (disponible a través de `cstdlib`):

```
<tipo_objeto> * <etiqueta_puntero> = new <tipo_objeto>;

<tipo_objeto> * <etiqueta_puntero> = (<tipo_objeto>*) malloc(<tamaño_bloque>);
```

Por ejemplo:

```
int * ptr = new int;        // Reservamos espacio en el heap para un variable entera e
                            inicializamos un puntero con una referencia al mismo
*ptr = 10;                  // Modificamos el contenido de la variable referenciada
                            por el puntero
```



Aquí se ve un contraste respecto a las regiones de memoria donde se encuentran la variable local y la variable que administra el programador, la cual es creada mediante la instrucción `new`. La memoria administrada por el programador no es liberada hasta que no se realice en forma explícita, a través de la instrucción `delete`.

Operaciones para liberar memoria

Existen dos operaciones para liberar memoria: `delete` (introducido en C++) y `free` (disponible a través de `cstdlib`):

```
delete <etiqueta_puntero>;

free(<etiqueta_puntero>);
```

Por ejemplo:

```
delete ptr;
ptr = 0;
```

Aclaraciones para la operación `delete`:

- Lo que se libera es el espacio en memoria al que referencia el valor del puntero (no el puntero en sí mismo). El valor del puntero al utilizar `delete` debe ser alguno retornado por el operador `new`, es decir, el objeto eliminado debe encontrarse en el montículo (por otra parte, cualquier intento de utilizar `delete` sobre una referencia a una variable de la pila arrojará un error).
- Luego de ejecutar `delete`, la posición de memoria del objeto que referenciaba el puntero queda invalidada (o como suele decirse el puntero queda invalidado o colgado). A continuación, es posible reinicializarlo con una nueva dirección de memoria válida o bien asignarle 0 o `NULL` a fin de evitar errores, ya que si intenta desrefenciar un puntero inválido o volver a ejecutar la operación `delete` sobre el mismo llevará a errores de ejecución.
- No es necesario verificar si el puntero es nulo antes de llamar a `delete`.

Consideraciones sobre la utilización de malloc y free

Es posible y está bien utilizar new y delete, y malloc (así como las variantes calloc y realloc) y free en un mismo programa. Sin embargo, hay que tener cuidado de no utilizar free sobre un espacio de memoria obtenido a través de new. De igual forma, no se debe utilizar delete sobre memoria obtenida a través de malloc. El problema es que las funciones no fueron especificadas por el lenguaje para trabajar en forma conjunta, y no es posible garantizar que vayan a funcionar bien en todas las plataformas.

Sin embargo, para la mayoría de los casos de uso más frecuentes se recomienda utilizar new y delete por las razones que se enumeran a continuación:

- new es una operación de tipado seguro. Es decir, devuelve punteros de un tipo determinado; malloc en cambio, devuelve un puntero de tipo sin valor (void*). Por ejemplo:

  ```
  int * ptr_new = new int;
  int * ptr_malloc = (int*) malloc(sizeof(int));
  ```

- new y delete no solo reservan y liberan el espacio en memoria de los objetos si no que además se encargan de invocar a las funciones constructoras y destructoras respectivamente. Por otra parte, malloc y free no ofrecen soporte para clases, es decir que durante la creación y liberación de los objetos que sean instancias de clases no serán correctamente inicializados ni destruidos.

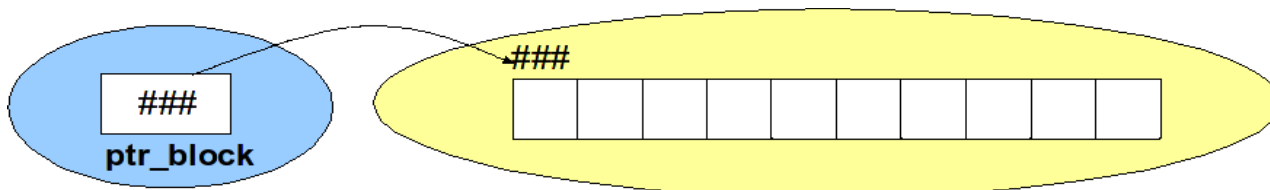
Obteniendo bloques de memoria contigua

Es posible reservar espacio para más de una variable de modo que las mismas queden ubicadas en posiciones de memoria consecutivas:

```
<tipo_objeto> * <etiqueta_puntero> = new <tipo_objeto> [<cantidad_variables>];
```

Por ejemplo:

```
int * ptr_block = new int[10];
```



Para liberar bloques de memoria se debe utilizar la siguiente variante de la operación delete:

```
delete [] <etiqueta_puntero>;
```

Por ejemplo:

```
delete [] ptr_block;
```

No es necesario indicar de ninguna forma ni el tipo ni la cantidad de elementos de los arreglos que se liberan con delete[] (o free). El sistema de tiempo de ejecución se encarga de registrar esa información automáticamente.

No hay que confundir un bloque de memoria contigua con un arreglo. Si bien existen similitudes en la forma de manipularlos (gracias a características propias de los arreglos y a la aritmética de punteros) y pueden considerarse equivalentes en algunos aspectos, no son lo mismo. En las secciones siguientes se profundizará sobre estas equivalencias y diferencias.

Arreglos y punteros

Siempre que un arreglo aparece como parte del lado izquierdo o derecho de una expresión el compilador genera implícitamente una conversión a un puntero constante al primer elemento del arreglo. Esto implica que para toda manipulación de los elementos de los arreglos a través de la notación de subíndices en realidad se está utilizando la expresión de subíndices de la aritmética de punteros.

La expresión de subíndice tiene la forma **`E1[E2]`**, donde **`E1`** es un puntero a un elemento de tipo T y **`E2`** es una expresión de tipo entero. El resultado es un valor de tipo T, equivalente al obtenido a través de la expresión `*`**`(E1+E2)`** (esto es el valor de la variable que se encuentra desplazándose E2 celdas a partir de la posición indicada por E1).

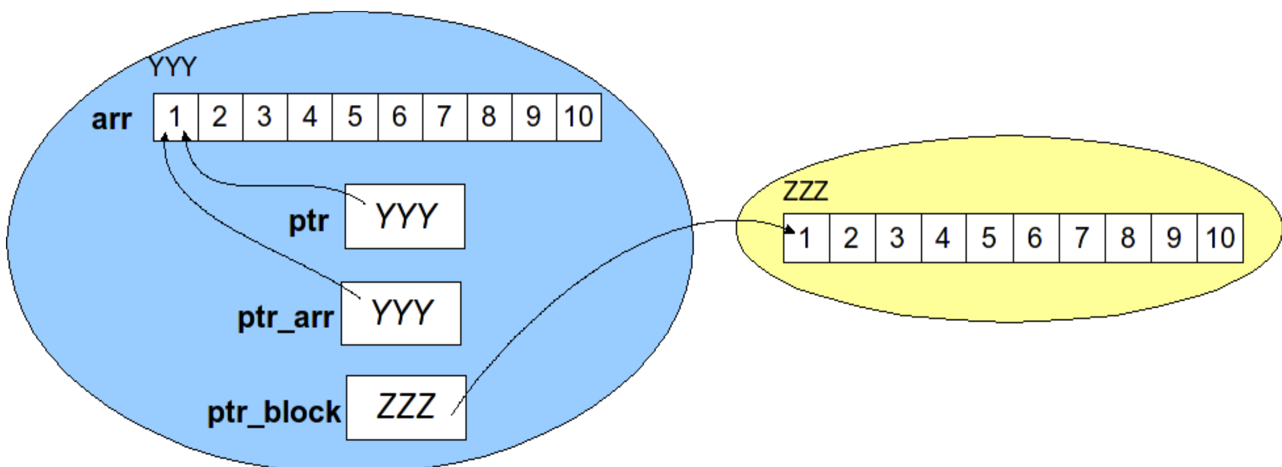
Teniendo en cuenta lo dicho anteriormente veremos que los fragmentos de código que siguen son funcionalmente equivalentes, pero introduciendo pequeñas variantes en cuanto a la forma de acceso a los elementos del arreglo.

	// versión 1	// versión 2	// versión 3
1	`int arr[5] = {1,2,3,4,5};`	`int arr[5] = {1,2,3,4,5};`	`int arr[5] = {1,2,3,4,5};`
2	`for (int i=0; i<5; i++)`	`int * ptr = arr;`	`int * ptr = arr;`
3	`    arr[i] = arr[i]*2;`	`for (int i=0; i<5; i++)`	`for (int i=0; i<5; i++)`
4	`for (int i=0; i<5; i++)`	`    arr[i] = ptr[i]*2;`	`    ptr[i] = arr[i]*2;`
5	`    cout << arr[i];`	`for (int i=0; i<5; i++)`	`for (int i=0; i<5; i++)`
6		`    cout << ptr[i];`	`    cout << arr[i];`

La primera versión declara e inicializa un arreglo, y utiliza la notación de subíndice ya vista para modificar los valores del mismo y luego imprimirlos. Recordando que tanto en las expresiones de la línea 3 como la línea 5, el arreglo es convertido implícitamente a un puntero, es que se introduce el puntero `ptr` en las versiones 2 y 3 del código anterior. Como se ve, `ptr` se inicializa con el valor de `arr` (que en esa asignación, por supuesto, también se convierte en un puntero), lo que lleva a que `ptr` referencie la primera posición de dicho arreglo. Luego en la versión 2 `ptr` es utilizado para leer los valores del arreglo en vez de `arr`, y en la versión 3 `ptr` se utiliza para el acceso de escritura en su lugar.

Los elementos de un arreglo pueden ser accedidos a través de un puntero utilizando una sintaxis similar. Sin embargo, el arreglo y el puntero son objetos distintos. Al mismo tiempo, un bloque en memoria dinámica puede utilizarse para simular un arreglo y es posible manipularlo en forma equivalente pero tampoco es un arreglo tradicional (ya que entre otras cosas debe ser eliminado manualmente a diferencia de los arreglos que son inicializados y destruídos en forma automática). Para ejemplificar, se muestra la declaración e inicialización de un puntero a un elemento del arreglo, un puntero al arreglo[11], y un puntero a un bloque de características similares al arreglo original:

```
int arr[10];
int * ptr = arr;                // Puntero al primer elemento
int (* ptr_arr)[10] = &arr;     // Puntero al arreglo
int * ptr_block = new int[10];  // Puntero a un bloque
```



[11] Hay que tener en cuenta que los punteros a arreglo sólo pueden hacer referencia a arreglos de la dimensión especificada.

Completamos el ejemplo anterior con el código correspondiente a la inicialización e impresión del contenido de las estructuras:

```
for (int i=0; i<10; i++)
    ptr[i] = i+1;
for (int i=0; i<10; i++)
    cout << ptr[i];
```

```
for (int i=0; i<10; i++)
    (*ptr_arr)[i] = i+1;
for (int i=0; i<10; i++)
    cout << (*ptr_arr)[i];
```

```
for (int i=0; i<10; i++)
    ptr_block[i] = i+1;
for (int i=0; i<10; i++)
    cout << ptr_blcok[i];
```

Arreglos como parámetros de funciones

En primer lugar, debido a que no es posible la asignación entre arreglos, no es posible el pasaje por copia de los mismos. Es más, como los nombres de los arreglos se convierten en punteros, cuando pasamos un arreglo a una función, en realidad se pasa un puntero al primer elemento. Por ejemplo, si se quisiera crear una función que manipule arreglos de una dimensión de tamaño variable se podría escribir utilizando la notación de arreglos:

```
void funcion(int arr[], int longitud) {
    ...
}
```

o bien utilizando la notación de punteros que es equivalente:

```
void funcion(int * arr, int longitud) {
    ...
}
```

En las funciones anteriores, aunque el puntero al primer elemento del arreglo se pasa por copia, es posible modificar el contenido del arreglo porque, como no es válida la asignación de arreglos, no se puede realizar una copia del arreglo, si no que la referencia sigue siendo a la secuencia de celdas original. Sin embargo, es posible cambiar la definición de los parámetros anteriores para que las referencias a los elementos del bloque de memoria sean constantes de modo que los elementos de los arreglos no puedan modificarse desde las funciones (es decir, un mecanismo de restricción al acceso de los arreglos para ciertas porciones del código de modo que sólo se pueda leerlos).

Una consideración muy importante es que la conversión del nombre del arreglo a un puntero sucede una sola vez (no es recursiva). Por lo tanto, en el pasaje como parámetro de arreglos de dimensión mayor a uno, la situación es un poco más compleja y la solución involucra el pasaje de un puntero a un arreglo, lo que implica que las funciones podrán definirse para trabajar sobre arreglos de una dimensión específica. La siguiente referencia contiene una discusión muy buena acerca de este tema:

http://c-faq.com/aryptr/pass2dary.html

A raíz de esto último, puede apreciarse que trabajar con arreglos de múltiples dimensiones y tamaños variables puede ser problemático.

Arreglos dinámicos

Una solución al problema expuesto en la sección anterior es trabajar con arreglos dinámicos como se explica en:

http://c-faq.com/aryptr/dynmuldimary.html

Estos "arreglo dinámicos" son los arreglos simulados a través de bloques de memoria contigua que ya se mencionó. Existen varias ventajas al trabajar con arreglos implementados de esta forma:

- La principal es que el tamaño de estos arreglos lo podemos determinar en tiempo de ejecución, a diferencia de los arreglos estáticos los cuales requieren que su tamaño se especifique en tiempo de compilación.
- La definición y manipulación, especialmente el pasaje como parámetro es uniforme y por lo tanto resulta mucho más intuitivo.

- La naturaleza dinámica de estas estructuras lleva a que el código para manipular arreglos de múltiples dimensiones sea inherentemente genérico y en consecuencia reutilizable, en contraposición con las matrices tradicionales que obligan a definir funciones que trabajen con dimensiones específicas.

Lo que sigue es un ejemplo muy simple de como trabajar con un arreglo dinámico de dos dimensiones:

```
/* Es muy importante el pasaje por referencia del puntero a la matriz, porque de
otra forma la matriz será creada pero el puntero original no será modificado de
modo que nos permita acceder a la misma más adelante */

void crearMatriz2D(int ** & matriz, int ancho, int alto) {
    matriz = new int*[alto];
    for (int f = 0; f < alto; f++)
        matriz[f] = new int[ancho];
}

void eliminarMatriz2D(int ** & matriz, int ancho, int alto) {
    for (int f = 0; f < alto; f++)
        delete [] matriz[f];
    delete [] matriz;
    matriz = 0;
}

void cargarMatriz2D(int ** matriz, int ancho, int alto) {
    for (int f = 0; f < alto; f++)
        for (int c = 0; c < ancho; c++)
            matriz[f][c] = f * ancho + c + 1;
}

void mostrarMatriz2D(int ** matriz, int ancho, int alto) {
    for (int f = 0; f < alto; f++) {
        for (int c = 0; c < ancho; c++)
            cout << matriz[f][c] << " ";
        cout << "\n";
    }
}

int main(int argc, char * argv[])
{
    int ** matriz = 0;
    int ancho = 10;
    int alto = 5;
    crearMatriz2D(matriz, ancho, alto);
    cargarMatriz2D(matriz, ancho, alto);
    mostrarMatriz2D(matriz, ancho, alto);
    eliminarMatriz2D(matriz, ancho, alto);
    return 0;
}
```

Arreglos de tamaño variable y el estándar C99

En el estándar de C de 1999 se agrega una extensión a la definición del tamaño de los arreglos. Esta nueva definición permite utilizar una expresión no constante para determinar el tamaño de un arreglo.

Si bien esto permite la declaración de un arreglo cuyo tamaño se determina en tiempo de ejecución, no se está definiendo un arreglo dinámico, ya que la expresión no constante se evaluará únicamente al momento de crear el arreglo. El tamaño del arreglo no podrá modificarse una vez definido. A este tipo de arreglos se los conoce como arreglos de tamaño variable.

Por ejemplo, podríamos definir un arreglo, cuyo tamaño es determinado en tiempo de ejecución, de la siguiente forma:

```
int n;
cin >> n;
int arreglo[n];
// Operaciones sobre el arreglo.
```

Si bien las últimas versiones del compilador GCC han adoptado casi por completo el estándar C99, incluyendo los arreglos de tamaño variable, todavía no han sido adoptados por la mayoría de los compiladores. Por lo tanto, utilizar este tipo de arreglo, comprometería la portabilidad de nuestro código. Por otro lado, el arreglo es almacenado en la pila, con lo cual se puede generar un error si el arreglo es de un tamaño demasiado grande.
Por estas razones, recomendamos la definición de arreglos dinámicos mediante la utilización de bloques de memoria contigua, como se vió anteriormente.

Aritmética de punteros

Es posible obtener un desplazamiento hacia otras posiciones de memoria partiendo de la posición indicada por un puntero.

Una forma es utilizando las operaciones de sumas y restas:

```
<puntero> + <desplazamiento>
<puntero> - <desplazamiento>
```

Otra forma es utilizando la notación de subíndice:

```
<puntero>[<dezplazamiento>]
```

Esta operación implica un desplazamiento y una desreferencia.

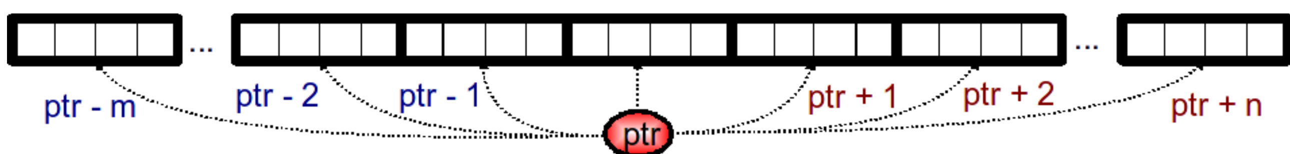
Cuando hablamos de aritmética de punteros, hablamos de las operaciones básicas o elementales que podemos realizar con los punteros.

Hay que notar que las operaciones con subíndices son las que nos permiten trabajar fácil e intuitivamente con punteros y arreglos, dando lugar a la denominada equivalencia que se mencionó anteriormente.

Las operaciones de suma o resta sólo son válidas con punteros con tipo; no están definidas sobre punteros sin tipo: `void *`. Esto se debe a que no es posible determinar el tamaño de los objetos referenciados por un puntero sin tipo.

También es necesario que el operando izquierdo sea del tipo puntero y el operando derecho sea de tipo entero.

Como es evidente, la unidad del desplazamiento no es un byte, sino una cantidad de bytes igual al tamaño del tipo del objeto (`sizeof(<tipo_objeto>)`) almacenado en cada celda de un arreglo o un bloque.



A través de las operaciones de desplazamiento se puede alcanzar virtualmente cualquier posición de memoria. Sin embargo, sólo los desplazamientos sobre posiciones de arreglos estáticos o dinámicos tienen sentido. Hay que tener especial cuidado de no sobrepasar los límites de los arreglos ya que el lenguaje no verifica esto de ninguna forma.

Por último hay que mencionar que también está definida la resta entre punteros:

```
<puntero1> - <puntero2>
```

Dicha operación es posible entre punteros del mismo tipo. Además sólo tiene sentido si ambos punteros hacen referencia a posiciones de un mismo arreglo estático o celdas de un mismo bloque de memoria contigua.

Veamos un pequeño ejemplo:

```
void cargarArreglo (int * arreglo, int limite) {
      for(int i = 0; i < limite; i++)
            arreglo[i] = i;              // *(arreglo + i) = i;
}

void mostrarArreglo (int * arreglo, int limite) {
      for(int i = 0; i < limite; i++)
            printf("%i\n", arreglo[i]);  // printf("%i\n",*(arreglo + i));
}

int main()
{
      int * arr = new int[10];           // int arr[10];
      cargarArreglo(arr, 10);            // cargarArreglo(&arr[0], 10);
      mostrarArreglo(arr + 5, 5);        // mostrarArreglo(&arr[5], 5);
      delete [] arr;
      return 0;
}
```

En este ejemplo vemos algunas de las operaciones que podemos realizar con punteros. La idea es que cualquier línea puede ser reemplazada por el código comentado que le sigue y el programa resultante seguiría comportándose de la misma forma.

Es claro que el cambio más significativo es cambiar el arreglo estático por el dinámico. Pero aún así el programa seguirá funcionando de la misma forma.

Clases

El mecanismo de clases es una de las extensiones principales que incorpora el lenguaje C++, a la programación procedural propia de C. Esto posibilita la orientación a objetos que constituye un paradigma nuevo de programación.

Una clase se puede definir como una estructura que agrupa datos y funciones, permitiendo su encapsulamiento. Las clases representan nuevos tipos de datos.

Las clases están formadas por dos tipos de componentes o miembros: las propiedades o atributos (los datos) y los métodos o funciones. Formalmente se definen como variables y métodos de instancia.

Un objeto es una instancia de una clase determinada. Los objetos son los encargados de procesar los mensajes recibidos, es decir, de responder ante el llamado a cada una de las funciones. Mediante los métodos se podrán manipular los datos propios de los objetos.

Definición de clases - Declaración de la interfaz

Para definir una clase se utiliza la siguiente sintaxis:

```
class <nombre_clase>
{
{<nivel_de_acceso>:}
      <lista_de_miembros>
{<nivel_de_acceso>:}
      <lista_de_miembros>
…
};
```

El nombre de la clase debe ser un identificador válido. Este identificador definirá un nuevo tipo de datos que luego se utilizará para declarar variables (o instanciar objetos) del tipo de la clase.

Dentro del cuerpo de la clase, se deben definir las variables y funciones miembro que la compondrán. Además, opcionalmente, se puede definir un nivel de acceso para cada una de ellas.

El nivel de acceso, modifica los permisos de acceso a los miembros definidos antes de la declaración de un nuevo nivel de acceso. Existen tres alternativas:

- **`public`**: los miembros públicos son accesibles desde cualquier parte donde el objeto es visible.
- **`private`**: los miembros privados sólo son accesibles desde los propios miembros de la clase.
- **`protected`**: los miembros protegidos sólo son accesibles desde los propios miembros de la clase, funciones o clases “friends” y miembros de clases derivadas.

Si no se especifica ningún nivel de acceso, por defecto, los miembros de una clase son privados. Por lo tanto, cualquier miembro declarado antes de la definición de un nivel de acceso es considerado como privado.

Por ejemplo:

```
class Punto
{
    int x;
    int y;

public:
    void cargarCoordenadas(int cx, int cy);
    int coordenadaX() const;
    int coordenadaY() const;
};
```

O lo que es equivalente:

```
class Punto
{
public:
    void cargarCoordenadas(int cx, int cy);
    int coordenadaX() const;
    int coordenadaY() const;
private:
    int x;
    int y;
};
```

En este ejemplo, se define una clase `Punto`, la cual contiene dos atributos privados, de tipo entero, `x` e `y`; y tres métodos de instancia públicos: **`cargarCoordenadas`**, **`coordenadaX`** y **`coordenadaY`**.

Se denomina interfaz a la parte pública de una clase. La interfaz generalmente sólo consta de funciones; rara vez se hacen públicos los atributos. Al quedar ocultos los datos se hace efectiva la abstracción y el encapsulamiento de los tipos de datos abstractos. De esta forma, la interfaz de funciones aísla a los usuarios de los cambios en los tipos y cantidad de atributos.

En general, los cambios en las estructuras de datos de las clases no deberían afectar al resto del código que las utiliza, esto es, suponiendo que la definición de los métodos no varía.

Instanciación y uso

En el fragmento de código siguiente veremos como se declara un objeto de tipo `Punto` y como se invocan sus métodos:

```
Punto punto;
punto.cargarCoordenadas(3,4);
cout << "Coordenada X: " << punto.coordenadaX()
      << " - Coordenada Y: " << punto.coordenadaY();
```

Un objeto es como una variable de registro, donde toda la manipulación de los datos se hace a través de los métodos. Por otra parte, si se quisiera acceder directamente los atributos de las coordenadas `x` e `y` (para consultar o modificar su valor) se produciría un error de compilación:

```
// Cada línea produce un error de compilación: x e y son privadas.
punto.x = 3;
punto.y = 4;
cout << "Coordenada X: " << punto.x << " - Coordenada Y: " << punto.y;
```

Como se mencionó anteriormente, el nombre de la clase corresponde a un nuevo tipo de datos. Por lo tanto, también es posible definir y manipular punteros del tipo de la clase, como se ve en el ejemplo siguiente:

```
Punto * punto = new Punto();
punto->cargarCoordenadas(3,4);
cout << "Coordenada X: " << punto->coordenadaX()
      << " - Coordenada Y: " << punto->coordenadaY();
```

```
delete punto;
```

Es importante notar que la invocación de los métodos de los objetos referenciados es similar a la forma de acceder a los campos de un registro a través de un puntero. El operador -> funciona como una desreferencia y acceso a miembro, que puede utilizarse tanto para atributos como funciones.

Definición/Implementación de la interfaz

Como se puede ver en el ejemplo de la definición de la clase `Punto`, solamente se incluye la declaración de los métodos de instancia, pero no su definición. Para que una clase esté completa es necesario implementar todos los métodos.

Para esto existen dos alternativas. La primer opción es implementar los métodos a continuación de la declaración. Esto es útil cuando el código de cada método es pequeño, de lo contrario suele dificultar su legibilidad.

Por ejemplo:

```
class Punto
{
public:
    void cargarCoordenadas(int cx, int cy) {
        x = cx;
        y = cy;
    }
    int coordenadaX() const {
        return x;
    }
    int coordenadaY() const {
        return y;
    }
private:
    int x;
    int y;
};
```

La segunda alternativa consiste en implementar los métodos fuera de la clase en cuestión. Para esto se utiliza el operador de ámbito (::), el cual nos permite especificar la clase a la cual pertenece el método que se está definiendo:

```
<tipo_retorno> <nombre_clase>::<nombre_función>(<lista_de_parámetros>);
```

Por ejemplo:

```
class Punto
{
public:
    void cargarCoordenadas(int cx, int cy);
    int coordenadaX() const;
    int coordenadaY() const;
private:
    int x;
    int y;
};
void Punto::cargarCoordenadas(int cx, int cy)
{
      x = cx;
      y = cy;
}
int Punto::coordenadaX() const
{
      return x;
}
```

```
int Punto::coordenadaY() const
{
	return y;
}
```

Los atributos son accesibles desde cualquier método asociado a la clase (es decir, definido dentro de su ámbito), como si fueran variables globales a todos ellos.

La implementación de los métodos miembro, por fuera de la clase, se puede incluir en el mismo archivo de encabezado .h, donde se declara la clase, o en un archivo fuente .cpp separado.

Métodos consultores

Como se vió en la declaración y definición de la clase `Punto`, los métodos de instancia **`coordenadaX`** y **`coordenadaY`**, tenían la palabra reservada **`const`** al final de su declaración:

```
int coordenadaX() const;
int coordenadaY() const;
```

Al declarar los métodos de esta forma, se indica que son métodos consultores. Los métodos consultores, son denominados así ya que sólo consultan la información acerca del estado interno de los objetos sin modificarlos.

Cuando un método recibe por parámetro una referencia constante a un objeto sólo podrá invocar a los métodos consultores de dicho objeto.

Por ejemplo:

```
void Punto::copiarA(const Punto & otroPunto)
{
	// Error de compilación: El método invocado no respeta la referencia
	// constante.
	otroPunto.cargarCoordenada(x, y);
}
```

Constructores

Los objetos generalmente necesitan realizar operaciones durante su proceso de creación, como por ejemplo, inicializar variables o asignar memoria dinámicamente.

En el ejemplo anterior, si se define un objeto de tipo `Punto` y se invoca a **`coordenadaX`** antes de determinar las coordenadas de dicho punto (mediante el método **`cargarCoordenadas`**), se obtendrá un resultado indeterminado, ya que las variables `x` e `y` no fueron inicializadas correctamente.

Las clases cuentan con una función especial, denominada **constructor**, que lleva a cabo las operaciones de inicialización necesarias. El constructor es invocado automáticamente cuando se crea un nuevo objeto de una clase determinada; debe tener el mismo nombre que la clase y no retornar ningún valor.

Por ejemplo:

```
class Punto
{
public:
    Punto();
    void cargarCoordenadas(int cx, int cy);
    int coordenadaX() const;
    int coordenadaY() const;
private:
    int x;
    int y;
};
```

```
Punto::Punto()
{
	x = 0;
	y = 0;
}
```

En este caso, el constructor de la clase Punto define un punto en el origen (0,0). De esta forma, cada nuevo objeto creado contendrá coordenadas inicializadas correctamente, evitando resultados indeterminados.

Para crear un objeto utilizando este constructor:

```
Punto punto; //Sin los paréntesis ()
```

Constructores con argumentos

Al igual que todas las funciones del lenguaje, el constructor se puede sobrecargar, modificando el número o tipo de los argumentos. El compilador determinará que constructor llamar de acuerdo a los parámetros pasados en la creación de los objetos. De esta forma, podemos definir un nuevo constructor, con dos argumentos enteros, en reemplazo de la función **cargarCoordenadas**.

```
class Punto
{
public:
    Punto();
    Punto(int cx, int xy);
    int coordenadaX() const;
    int coordenadaY() const;
private:
    int x;
    int y;
};

Punto::Punto()
{
	x = 0;
	y = 0;
}

Punto::Punto(int cx, int cy)
{
	x = cx;
	y = cy;
}
```

Para crear un objeto utilizando este nuevo constructor:

```
Punto p(4,2);
```

Si no se define un constructor, el compilador creará uno por defecto sin argumentos, que no realizará ninguna acción. Sin embargo, si se define un constructor, el compilador ya no proveerá un constructor por defecto; por lo tanto, todas las instanciaciones de objetos deberán utilizar dicho constructor respetando los argumentos esperados.

Constructor por copia

Además del constructor sin argumentos, el compilador define un constructor por copia por defecto si no se ha definido uno.

El constructor por copia crea un objeto a partir de otro existente. Este tipo de constructor sólo tiene un parámetro; una referencia constante a un objeto de la misma clase. El constructor por copia implícito declarado por el compilador hace una asignación simple atributo por atributo.

Para la clase `Punto`, el constructor por copia por defecto es funcionalmente equivalente a:

```
Punto::Punto(const Punto & otroPunto)
{
      x = otroPunto.x;
      y = otroPunto.y;
}
```

Por lo tanto, el siguiente fragmento de código es válido:

```
Punto pA(2,3);
Punto pB(pA);
```

El constructor por copia se puede redefinir, modificando su implementación, siempre que se respete su interfaz. Para el ejemplo del punto esto no es realmente necesario porque todos los atributos son variables simples.
Sin embargo, consideremos el caso de una clase con una estructura dinámica:

```
class Vector
{
public:
    Vector(unsigned int tam);
    …
private:
    int * elementos;
    unsigned int longitud;
};

Vector::Vector(unsigned int tam)
{
    elementos = new int [tam];
    longitud = tam;
}
```

Supongamos que quisiéramos hacer una inicialización de un vector a partir de otro:

```
Vector vA(10);
Vector vB(vA);
```

El estado resultante de la inicialización de `vB` no sería el esperado. Esto es porque el constructor por copia por defecto realiza una asignación simple del atributo elementos, y al tratarse de un puntero esto no resulta adecuado. Veamos como sería el código del constructor por defecto:

```
Vector::Vector(const Vector & otroVector)
{
    elementos = otroVector.elementos;
    longitud = otroVector.longitud;
}
```

El comportamiento por defecto del constructor por copia hace que `vB` quede haciendo referencia al mismo bloque de memoria que `vA`. Esto es un problema grave que se debe resolver implementando el constructor por copia de forma explícita, para que se comporte como esperamos:

```
class Vector
{
public:
    Vector(unsigned int tam);
    Vector(const Vector & otroVector);
    …
private:
    int * elementos;
    unsigned int longitud;
};

Vector::Vector(const Vector & otroVector)
{
    elementos = new int [otroVector .longitud];
    longitud = otroVector.longitud;
    for (int i = 0; i < longitud; i++)
        elementos[i] = otroVector.elementos[i];
}
```

Destructor

El destructor es una función miembro especial, que se utiliza al eliminar un objeto de una clase determinada. Al igual que el constructor, es una función que no se invoca explícitamente, sino que lo hace el compilador automáticamente cuando un objeto es destruido; ya sea porque se terminó el ámbito donde estaba definido o porque es un objeto creado dinámicamente y se ha liberado su memoria, usando el operador `delete`.

El destructor debe tener el mismo nombre de la clase, precedido por el símbolo ~. No recibe argumentos ni retorna valores.

El uso principal de los destructores es liberar la memoria dinámica, que podría haber asignado el objeto durante su tiempo de vida.

Por ejemplo, en el caso de la clase `Vector`, se define un destructor de la siguiente manera:

```
class Vector
{
public:
    …
    ~Vector();
    …
};

Vector::~Vector()
{
    delete [] elementos;
}
```

Sobrecarga de operadores

C++ permite utilizar los operadores clásicos, generalmente definidos para los tipos básicos, para realizar operaciones con objetos.

Por ejemplo, podríamos definir el operador de comparación por igual (==) para la clase `Punto`. Esto permitiría conocer si dos puntos son iguales utilizando la siguiente sintaxis:

```
Punto pA(2,3), pB(7,5);
if (pA == pB) {
    …
}
```

Para sobrecargar un operador es necesario declarar las funciones operador, las cuales son como cualquier otro método, pero cuyo nombre se escribe utilizando la palabra reservada **`operator`** seguida del operador que se quiere sobrecargar. La sintaxis general es:

```
<tipo_retorno> operator <operador> (<lista_de_parámetros>);
```

Algunos de los operadores más comunes que se pueden sobrecargar son:

+, -, *, /, =, <, >. >=, <= +=, -=, *=, /=, <<, >>, ==, !=, ++, –, [], ()

Por ejemplo, para el caso del operador de comparación por igual entre puntos:

```
class Punto
{
public:
    …
    bool operator==(const Punto & punto) const;
    …
};

bool Punto::operator==(const Punto & otroPunto) const
{
      return (x == otroPunto.x) && (y == otroPunto.y);
}
```

A continuación se presenta una tabla con la declaración necesaria para poder sobrecargar los operadores más importantes[12] (el símbolo @ se debe reemplazar por cada operador).

Expresión	Operador	Función miembro	Función global
@a	+ - * ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / < > == != <= >= << >>	A::operator@ (B)	operator@(A,B)
a@b	= += -= *= /= []	A::operator@ (B)	-
a(b, c...)	()	A::operator() (B, C...)	-

Donde **a** es un objeto de la clase **A**, **b** es un objeto de la clase **B** y **c** es un objeto de la clase **C**. La última columna indica que operadores pueden ser definidos como funciones globales, además de como funciones miembro.

Operador de salida de streams

Un ejemplo de un operador que generalmente se sobrecarga como una función global, es el operador de salida de los streams de C++ (<<).

Por ejemplo, para imprimir directamente un objeto del tipo punto, pasándolo como argumento al utilizar el cout, debemos definir la siguiente función que sobrecarga el operador correspondiente:

```
std::ostream & operator << (std::ostream & salida, const Punto & punto)
{
    salida <<"("<< punto.coordenadaX() <<","<< punto.coordenadaY() <<")";
    return salida;
}
```

Lo cual permite escribir lo siguiente:

```
Punto pA(2,3), pB(7,5);
cout << "Punto A " << pA << "\n";
cout << "Punto B " << pB << "\n";
```

[12] Para ver una lista completa de los operadores que pueden ser sobrecargados: http://www.cplusplus.com/doc/tutorial/classes2/

Operador de asignación

El operador de asignación es el único operador que se implementa por defecto. Sin embargo, es una buena práctica siempre redefinir dicho operador cuando se define una clase nueva. Esto es porque la versión implícita del operador se comporta de forma similar al constructor por copia por defecto. El mismo también realiza una asignación simple atributo por atributo, lo cual puede puede llevar al mismo tipo de problemas que ya se indicaron.

Siguiendo el ejemplo de la clase `Vector` definiremos el operador de asignación y lo aprovecharemos para reimplementar el constructor por copia y así evitar la duplicación de código:

```
class Vector
{
public:
    Vector(const Vector & otroVector);
    Vector & operator=(const Vector & otroVector);
    …
};

Vector::Vector(const Vector & otroVector) {
    elementos = 0;
    operator=(otroVector);
}

Vector & Vector::operator=(const Vector & otroVector) {
    delete [] elementos;
    elementos = new int [otroVector.longitud];
    longitud = otroVector.longitud;
    for (int i = 0; i < longitud; i++)
        elementos[i] = otroVector.elementos[i];
    return *this;
}
```

La palabra reservada `this`

La palabra reservada `this` representa un puntero al objeto que está procesando el mensaje. A través de esta referencia se pueden acceder a las variables de instancia de las clases, en forma alternativa a la ya observada. Esto es útil cuando los parámetros ocultan a los atributos al coincidir sus nombres.

Un uso importante que tiene el parámetro implícito `this` es cuando se define el operador de asignación. En primer lugar el operador debe retornar una referencia al objeto que procesa el mensaje (el que está del lado izquierdo de la asignación). La única forma de obtener dicha referencia es a través del puntero `this`, escribiendo `*this`.

En el mismo contexto del operador de asignación, también se puede aprovechar el puntero `this` para evitar asignar un objeto así mismo. En el caso particular de la clase `Vector` esto es fundamental; por lo cual, el código quedaría de la siguiente forma:

```
Vector & Vector::operator=(const Vector & otroVector) {

    if (this != &otroVector) {
        delete [] elementos;
        elementos = new int [otroVector .longitud];
        longitud = otroVector.longitud;
        for (unsigned int i = 0; i < longitud; i++)
            elementos[i] = otroVector.elementos[i];
    }
    return *this;
}
```

Clases parametrizadas

El mecanismo de plantillas o templates de C++ permite definir clases que reciben tipos como argumentos. Podremos utilizar los tipos con los que se parametriza una clase, como si fueran tipos regulares a la hora de definir los métodos y atributos.

La sintaxis general de una clase parametrizada es la siguiente:

```
template <typename <identificador> {, typename <identificador>}>
class <nombre_clase> {
   …
};
```

La palabra clave **`typename`** puede reemplazarse por **`class`**. Ambas son intercambiables.

Por ejemplo, se podría parametrizar la clase Punto, vista anteriormente, para definir el tipo de coordenadas que la componen (discretas o continuas, positivas y negativas, etc).

```
template<typename T> class Punto
{
    public:
        Punto(const T & x, const T & y);
        ~Punto();
        const T & coordenadaX() const;
        const T & coordenadaY() const;
        void trasladar(const T & x, const T & y);
    private:
        T x, y;
};
```

El tipo de datos concreto en el cual se instanciará a la clase se debe incluir en la definición de cada objeto:

```
Punto<unsigned int> pA(1,2);
Punto<float> pB(7.5,5.0);
```

Que un tipo esté parametrizado no significa que se pueda instanciar utilizando cualquier tipo de datos como parámetro. La forma en la que están implementados los métodos impone ciertas restricciones sobre los tipos que se podrán utilizar.

Por ejemplo, el método **`trasladar`** utiliza operaciones de suma y asignación como parte de su implementación, razón por la cual sólo tipos de datos que contengan esas operaciones podrán ser utilizados para instanciar a la clase `Punto`.

```
template <typename T>
void Punto<T>::trasladar(const T & x, const T & y) {
    this->x += x;
    this->y += y;
}
```

Clases parametrizadas y encabezados

Cuando se separa la definición de una clase parametrizada de su implementación, mediante el uso de encabezados, es necesario realizar un paso adicional para que se pueda compilar la clase correctamente.

Por ejemplo, considerando la clase Punto, si se separa la definición en un archivo Punto.h y la implemenentación en Punto.cpp, al momento de definir un objeto de esta clase (Punto<unsigned int> p) se generará un error de tipo "undefined reference...". Esto se debe a que los archivos .h y .cpp son compilados por separado y en ninguno de los existe una definición de la instanciación Punto<unsigned int>.

Existen dos soluciones posibles a este escenario:

1. Ubicar todo el código de la clase (definición e implementación) en el archivo Punto.h:

```
template <typename T> class Punto
{
    public:
        Punto(const T & x, const T & y);
            ~Punto();
            const T & coordenadaX() const;
            const T & coordenadaY() const;
        void trasladar(const T & x, const T & y);
        ...
};
template <typename T>
Punto<T>::Punto(const T & x, const T & y) {
    ...
}
...
template <typename T>
void Punto<T>::trasladar(const T & x, const T & y) {
    this->x += x;
    this->y += y;
}
```

2. Agregar al final del archivo .cpp las posibles instanciaciones de tipos para la clase que se está implementando:

```
template <typename T>
Punto<T>::Punto(const T & x, const T & y) {
    ...
}
...
template <typename T>
void Punto<T>::trasladar(const T & x, const T & y) {
    this->x += x;
    this->y += y;
}
template class Punto<unsigned int>;
template class Punto<float>;
```

Contenedores genéricos

Uno de los escenarios donde es de mayor utilidad la parametrización es cuando se está definiendo tipos de datos cuya función principal es contener otros objectos. Por ejemplo, consideremos el caso de la clase `Vector`, la cual se definió previamente como un contenedor de enteros. Utilizando el mecanismo de plantillas podemos redefinirlo para que almacene objetos de cualquier tipo:

```
template<typename T> class Vector
{
public:
    Vector(unsigned int tam);
    Vector(const Vector & otroVector);
    ~Vector();
    Vector & operator=(const Vector & otroVector);
    T & operator[](unsigned int i);
    bool existe(const T & elem) const;
private:
    T * elementos;
    unsigned int longitud;
};

template<typename T> Vector<T>::Vector(unsigned int tam) {
    elementos = new T [tam];
    longitud = tam;
}
```

```
template<typename T> Vector<T>::Vector(const Vector<T> & otroVector) {
    elementos = 0;
    operator=(otroVector);
}

template<typename T> Vector<T>::~Vector() {
    delete [] elementos;
}

template<typename T> Vector<T> & Vector<T>::operator=(const Vector<T> &
otroVector) {
    if (this != &otroVector) {
        delete [] elementos;
        elementos = new T [otroVector.longitud];
        longitud = otroVector.longitud;
        for (unsigned int i = 0; i < longitud; i++)
            elementos[i] = otroVector.elementos[i];
    }
    return *this;
}

template<typename T> T & Vector<T>::operator[](unsigned int i) {
    return elementos[i];
}

template<typename T> bool Vector<T>::existe(const T & elem) const {
    bool encontrado = false;
    unsigned int i = 0;
     while (!encontrado && (i < longitud)) {
          if (elementos[i] == elem)
               encontrado = true;
          else
               i++;
     }
     return encontrado;
}
```

Ahora podemos definir un vector para que contenga enteros o cadenas de caracteres, a partir de la misma definición, como se puede a continuación:

```
Vector<int> v(10);
Vector<string> w(10);
for(int i=0; i<10; i++)
    v[i] = i;
for(int i=0; i<10; i++) {

   char aux[20];
    sprintf(aux, "%i", i);
    w[i] = aux;
}
cout << v.existe(5) << "\n";
cout << w.existe("5") << "\n";
```

Recordando la importancia de las restricciones que imponen las implementaciones de los métodos al conjunto de tipos de datos posible para instancia la clase, se puede mencionar, por ejemplo, el método `existe` de la clase `Vector`. Este método asume que el tipo utilizado como parámetro contará con el operador de comparación por igual (==). Si esto no es así, se producirá un error de compilación que indicará este requerimiento implícito.

Herencia

La herencia es un mecanismo fundamental de la programación orientada a objetos que posibilita la creación de nuevas clases a partir de clases existentes.

La utilización de la herencia permite construir clasificaciones jerárquicas de clases, en donde cada clase definirá su funcionalidad y sus características propias, las cuales serán compartidas por todas las clases de los niveles inferiores de la jerarquía. Estas clases, además, extenderán la funcionalidad de las clases superiores modificando o agregando comportamiento.

La clase de la cual se hereda se denomina generalmente clase base, clase padre o superclase; mientras que la clase que extiende a una clase base, se conoce como clase derivada, clase hija o subclase. Cada clase derivada puede utilizarse como clase base para obtener una nueva clase derivada, y cada clase derivada puede heredar funcionalidad de una o más clases base (a esto se lo conoce como herencia múltiple).

La principal ventaja asociada a la utilización de herencia es la reutilización de código. Al definir una funcionalidad común en una clase base, todas las clases derivadas compartirán dicha funcionalidad sin duplicar código, y con la posibilidad de modificar ese comportamiento definido. También ayuda a posibilitar que una aplicación sea extensible, al permitir crear clases derivadas con las funcionalidades que se desean añadir o redefinir.

Las principales complicaciones de utilizar herencia se deben al trabajo que hay que realizar para diseñar correctamente las clases base, ya que cada cambio en dichas clases impactarán en todas las clases derivadas.

Creación de una clase derivada

La sintaxis general para definir una clase derivada es la siguiente:

```
class <nombre_clase_derivada> : <nivel_acceso> <nombre_clase_base>
{, <nivel_acceso> <nombre_clase_base>} {
    …
};
```

Como se ve, la herencia múltiple se puede realizar incluyendo más de una clase base para la construcción de una clase derivada.

En cuanto al nivel de acceso, las opciones posibles son:

- **`public`**: los miembros (atributos y métodos) de la clase base conservan el nivel de acceso con el cual fueron definidos.
- **`protected`**: los miembros de la clase base que fueron definidos como públicos son pasados al nivel de acceso protegido. Esto restringe el acceso, desde fuera de la jerarquía, a los miembros públicos de la clase base si se intentan acceder mediante la clase derivada.
- **`private`**: es el nivel de acceso por defecto. En este caso todos los miembros de la clase base pasan a ser privados en la clase derivada. Esto no permite que se acceda a los miembros de la clase base, mediante la clase derivada, desde fuera de la jerarquía ni desde un nivel inferior al de la clase derivada.

Por ejemplo, definiendo la siguiente clase base:

```
class A
{
public:
	int variableA1;
	void metodoA1();
protected:
	int variableA2;
	void metodoA2();
private:
	int variableA3;
	void metodoA3();
};
```

De acuerdo al nivel de acceso que se utilice al hereder de esta clase, se presentarán tres casos distintos de restricciones sobre sus miembros.

public	protected	private

class B : public A
{
La clase B puede acceder a los miembros públicos y protegidos de A.
public:
    void metodoB1(){
        variableA1++;
    }
};
 	
class C : public B
{
La clase C también puede acceder a los miembros públicos y protegidos de A.
    …
};
 	
int main()
{
Una instancia de B puede acceder a los miembros públicos de A y B.
    B objetoB;
    objetoB.metodoA1();
    objetoB.variableA1;
    objetoB.metodoB1();
}
	class B : protected A
	{
	La clase B puede acceder a los miembros públicos y protegidos de A.
	public:
	    void metodoB1() {
	        variableA1++;
	    }
	};
	
	class C : public B
	{
	La clase C también puede acceder a los miembros públicos y protegidos de A.
	    …
	};
	
	int main()
	{
	Una instancia de B no puede acceder a los miembros públicos de A, solo a los métodos públicos de B.
	    B objetoB;
	    objetoB.metodoB1();
	}
		class B : private A
		{
		La clase B podrá acceder a los miembros públicos y protegidos de A.
		public:
		    void metodoB1() {
		        variableA1++;
		    }
		};
		
		class C : public B
		{
		La clase C no podrá acceder a ningún miembro de A.
		    …
		};
		
		int main()
		{
		Una instancia de B no puede acceder a los miembros públicos de A, solo a los métodos públicos de B.
		    B objetoB;
		    objetoB.metodoB1();
		}

Después de ver los ejemplos, se puede concluir que el nivel de acceso utilizado para heredar de la clase base, es el nivel que se utilizará para restringir el acceso a todos los miembros de dicha clase, cuando se invocan desde la clase derivada.

Constructores y destructores de las clases derivadas

Cuando se instancia un objeto de una clase derivada, primero se invoca al constructor de la clase base de la cual extiende la clase en cuestión. Si la clase pertenece a una jerarquía de herencia de más de un nivel de extensión, los constructores de las clases base se invoca desde la clase de nivel superior en orden descendente hasta llegar a la clase derivada. Por último se invoca el constructor de la clase derivada.

En el caso del ejemplo visto anteriormente, al instanciar un objeto de la clase C, primero se llamará al constructor de la clase A, luego al constructor de la clase B y por último al constructor de la clase C.

En el caso que el constructor de una clase base requiera el pasaje de algún parámetro para su invocación, estos parámetros deberán ser pasados antes de inicializar los atributos de la clase derivada. Para esto se utiliza la siguiente sintáxis en la definición del constructor de la clase derivada:

```
<nombre_clase_derivada>({<lista_de_parámetros>}) :
    <nombre_clase_base>(<lista_de_parámetros>)
    {, <nombre_clase_base>(<lista_de_parámetros>)} {
        …
};
```

Por ejemplo, se define un constructor para la clase A que requiere un parámetro **int**. Si se desea invocar a este constructor desde un constructor de la clase B, se debe realizar de la siguiente forma:

```
class B : public A
{
public:
    B(int parametro1, int parametro2) : A(parametro1) {
    }
};
```

En el caso de los destructores, el orden de invocación es inverso al de los constructores. Es decir, primero se invoca al destructor de la clase derivada, luego a los destructores de los objetos que formen parte de la clase y, por último, a los destructores de las clases base desde el nivel más bajo al nivel superior.

Los constructores, los destructores y el operador de asignación (=) son métodos que no se herendan desde la clase base a las clases derivadas; es decir, deben ser definidos específicamente en cada clase derivada o sino el compilador se encargará de crearl sus versiones por defecto. La razón de esta diferencia con el resto de los métodos, que si se heredan automáticamente, es porque se encargan de construir y destruir un objeto, tareas que son específicas a cada clase particular.

Redefinición de miembros en las clases derivadas

La definición de un método en una clase derivada, con igual nombre que algún otro método existente en una clase base, oculta la definición de ese método y todos los que tengan el mismo nombre. Esto se conoce como redefinición o superposición (*overriding*) de un método.

La única forma de acceder al método de la clase base que quedó oculto es utilizando su nombre completo (siempre que el nivel de acceso de la herencia lo permita):

```
<nombre_objeto>.<nombre_clase_base>::<nombre_método>;
```

Por ejemplo:

```
class A
{
public:
    void procesar() {cout << "Procesar A";};
    void procesar(int valor);
};
class B : public A
{
public:
    void procesar(){cout << "Procesar B";};
};
int main () {
    B objetoB;
    objetoB.procesar();
    //objetoB.procesar(10); no es una invocación válida.
    objetoB.A::procesar();
    objetoB.A::procesar(10);
}
```

Esta lógica de redefinición no solo es válida para los métodos de una clase base sino también para todos los atributos de clase que contenga.

Polimorfismo

El polimorfismo es una de las propiedades más importantes de la programación orientada a objetos. Esta propiedad indica que un programa puede trabajar con un objeto perteneciente a una clase sin importar de qué clase concreta se trate.

En C++ el polimorfismo se aprovecha principalmente al utilizarlo junto a los mecanismos de punteros y referencias. El lenguaje nos permite acceder a objetos de una clase derivada mediante un puntero o referencia a una clase base. Obviamente, sólo se podrá acceder a los atributos y métodos que hayan sido definidos en la clase base.

Por ejemplo, si se define una función que reciba como parámetro una referencia a la clase base `A`, se podrá invocar con cualquier instancia de clases que hayan heredado de dicha clase:

```
void proceso(A & a) {
    a.procesar();
}

int main() {
    B objetoB;
    C objetoC;

    proceso(objetoB);
    proceso(objetoC);
}
```

En todos los casos, el método invocado es el que está definido en la clase base, con lo cual la salida sería:

```
> Procesar A
> Procesar A
```

Una mejor forma de aprovechar el polimorfismo y la jerarquía de clases sería si, al invocar al método definido en la clase base, se ejecutaran las redefiniciones realizadas en las clases derivadas. De esta forma, el comportamiento de un programa dependería de una instancia concreta que podría definirse en tiempo de ejecución, lo cual brindaría una flexibilidad muy importante para el desarrollo de aplicaciones. Esto se puede conseguir mediante lo que se conocen como métodos virtuales.

Métodos virtuales

Un método virtual es básicamente un método de clase pero con la palabra reservada **`virtual`** al comienzo de su declaración.

```
virtual <tipo_retorno> <nombre_función>(<lista_de_parámetros>){};
```

Como se mencionó, la característica principal de los métodos virtuales es que, cuando son invocados mediante un puntero o referencia a una clase base, es la clase derivada la que responde el llamado mediante su definición del método invocado.

Por ejemplo, si se cambia la definición de la función vista en el ejemplo anterior:

```
class A
{
public:
    virtual void procesar() {cout << "Procesar A";};
};

int main() {
    B objetoB;
    C objetoC;

    proceso(objetoB);
    proceso(objetoC);
}
```

Ahora la salida sería:

```
> Procesar B
> Procesar C
```

Este ejemplo también se podría haber realizado con un puntero (*) en lugar de una referencia (&).

Una vez que un método es declarado como virtual esta propiedad será heredada en las clases derivadas, por lo tanto, el método seguirá siendo virtual en estas clase. Si el método definido como virtual en la clase base, no se declara en las clases derivadas con el mismo tipo de valor de retorno, el mismo número y tipo de parámetros que en la clase base, no se considerará como el mismo método, sino como una redefinición o superposición.

Una diferencia que es importante mencionar en este sentido, es que la sobrecarga se resuelve en tiempo de compilación utilizando los nombres de los métodos y los tipos de sus parámetros; mientras que el polimorfismo se resuelve en tiempo de ejecución del programa en función de la clase a la que pertenece el objeto en cuestión.

Con respecto a los **constructores**, para construir un objeto el constructor debe ser del mismo tipo que dicho objeto, razón por la cual los constructores no pueden ser definidos como virtuales. Además, si un constructor llama a un método virtual, éste siempre será el de la clase base, debido a que el objeto de la clase derivada aún no ha sido creado.

Esta restricción no se aplica a los **destructores.** Por el contrario, si una clase tiene como parte de su interfaz métodos virtuales, el destructor debe ser declarado virtual. Esto es debido a que cuando un puntero a una clase base es destruido, se debe asegurar que se invoque el destructor de la clase derivada correspondiente para que se realice la liberación de recursos adecuado. Si un destructor no es declarado como virtual, siempre que se destruyan objetos de clases derivadas, a través de la clase base, sólo se realizarán las tareas definidas en el destructor de la clase base.

Clases abstractas

La creación de métodos virtuales posibilita la definición de un nuevo tipo de clases denominadas clases abstractas o incompletas. La utilización de este tipo de clases permite proveer un mecanismo, similar al encontrado en otros lenguajes, conocido como interfaces.

La característica distintiva de una clase abstracta es que no puede ser instanciada, sí se pueden definir punteros y referencias. Para definir una clase abstracta solamente es necesario definir uno de sus métodos como virtual puro.

```
virtual <tipo_retorno> <nombre_función>(<lista_de_parámetros>) = 0;
```

Por ejemplo:

```
class A
{
public:
    virtual void procesar() = 0;
};
```

Todas las clases que hereden de la clase `A` estarán obligadas a definir el método **`procesar`** para poder ser instanciadas, sino seguirán siendo clases abstractas.