INF-134 Estructuras de Datos

Departamento de Informática, Universidad Técnica Federico Santa María

Tutorial del Lenguaje C++

Diego Arroyuelo Roberto Díaz

1 Introducción al Lenguaje

El lenguaje C++ fue definido por Bjarne Stroustrup en 1985, como una evolución del lenguaje C. Tiene la carácterística de ser flexible y comparte la habilidad de C para manejar eficientemente el hardware a nivel de bits, bytes, direcciones, etc. Pero, además, incluye carácterísticas de suficiente alto nivel como para simplificar el desarrollo de software a mediana y gran escala. Este apunte no pretende ser un manual completo de C++, ya que sólo estudiará los conceptos básicos necesarios para un correcto desarrollo del curso. No abordaremos temas como orientación a objetos (por ejemplo, herencia y polimorfismo), sobrecarga de operadores, templates, así como ninguna de las carácterísticas introducidas en los últimos estándares del lenguaje. Sin embargo, el subconjunto del lenguaje abordado es suficiente para nuestro estudio de estructuras de datos en secciones posteriores del curso.

2 El Primer Programa

Comenzaremos el estudio de C++ con un ejemplo típico que suele ser usado como primer pograma al aprender un nuevo lenguaje: mostrar el mensaje "Hola Mundo" por pantalla. En C++, el programa es el siguiente:

```
#include <iostream>

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

Aunque éste es un programa muy simple, podemos notar lo siguiente:

- Los programas en C++ en general se construyen a partir de funciones. Cada una de las funciones que conforman un programa tiene un nombre elegido por el programador (al momento de definirlas) para identificarlas y usarlas. En general, hay libertad en el nombre que las funciones pueden tener (siguiendo algunas reglas básicas de sintaxis especificadas por el lenguaje). Sin embargo, la función `main` que aparece en el programa anterior es una función especial: la ejecución del programa comienza en esa función, por lo que todo programa debe tener una función `main`. El encabezamiento de la definición de la función "`int main()`" indica que la función `main` no tiene argumentos [1]. Esto es porque los paréntesis en la definición

[1] A lo largo del curso utilizaremos *argumentos* o *parámetros* para referirnos a lo mismo.

no encierran a ninguna declaración de parámetros. El encabezamiento también indica que la función retorna un valor entero (`int` en la terminología de C++). Aquí bien cabe la siguiente pregunta: Si la función `main` es el programa principal, ¿Cómo es posible que retorne un valor o tome argumentos? Los argumentos al programa principal, o función `main` serán tratados en una sección posterior. El valor retornado, un entero en este caso, es un valor que es pasado por el programa al sistema operativo al finalizar su ejecución. Este valor es utilizado por el sistema operativo para determinar si el programa finalizó en forma correcta o se produjo algún error. Por convención, el valor de retorno que indica que el programa finaliza exitosamente es `0`. Un valor distinto de `0` indica que el programa no pudo terminar correctamente dado que se produjo algún tipo de error. Este valor es retornado al sistema operativo a través de la sentencia `return 0`.

- Analicemos ahora la línea `std::cout << "Hola Mundo" << std::endl`, cuyo objetivo es mostrar el mensaje en la pantalla. Para manejar la entrada y salida de datos en dispositivos secuenciales (como una pantalla o teclado), C++ usa una abstracción llamada *streams* [2]. Un stream es una entidad donde un programa puede insertar o extraer carácteres. Esta abstracción permite ocultar los detalles de cómo se realizan dichas acciones a nivel de los dispositivos, por lo que sólo es necesario saber que los streams son fuente/destino de carácteres, y que los carácteres son provistos/recibidos secuencialmente, uno después de otro. La biblioteca estándar del lenguaje define una variedad de streams, como por ejemplo:

  - `std::cin`: stream de entrada estándar.
  - `std::cout`: stream de salida estándar.
  - `std::cerr`: stream de salida de errores estándar, utilizada para mostrar mensajes de error.

  En adelante, usaremos `cin` y `cout` con mayor frecuencia.

  Volviendo a nuestro programa, el operador `<<` es usado para enviar carácteres a `cout`, para ser impresos en pantalla. En este caso, el mensaje a imprimir es el string de carácteres "Hola Mundo", indicado por las comillas dobles. Finalmente, `std::endl` imprime una nueva línea.

- Al comienzo del programa aparece una línea con el siguiente contenido: `#include <iostream>`. Ésta es una instrucción para el preprocesador (se detallará más adelante), en la que se indica que se deben incluir las definiciones de la biblioteca estándar de entrada/salida `iostream`, la cual permite trabajar con streams. Note que los streams son un concepto definido en una biblioteca estándar, no siendo parte de la definición del lenguaje en sí.

3 Espacios de Nombres

Podemos simplificar un poco la escritura de nuestro primer programa si hacemos uso del concepto de *espacios de nombres* del lenguaje C++. Como habrá notado, para imprimir un mensaje por la pantalla hemos usado `std::cout`. Aquí, `std` es el espacio de nombres que contiene al stream `cout`. Lo mismo ocurre con `std::endl`. Para evitar usar el espacio de nombre cada vez, se puede incluir el espacio de nombres antes de la función `main` usando:

[2] Una posible traducción es *flujo*, aunque seguiremos usando stream en adelante.

```
using namespace std;
```

Eso indica que se usará el espacio de nombres `std`, por lo que luego podemos hacer:

```
cout << "Hola Mundo" << endl;
```

Aunque esto permite incluir todos los espacios de nombres que usemos en el programa, hay que tener cuidado, en particular en grandes proyectos de desarrollo de software. El problema es que puede haber alcance de nombres en distintos espacios de nombres, lo cual puede llevar a conflictos. Es por eso que para grandes proyectos se desaconseja usar `using namespace`.

4 Compilando y Ejecutando un Programa

A diferencia de lenguajes como Python, C++ es un lenguaje compilado. Eso significa que luego de escribir nuestro programa (conocido también como código fuente), debemos usar un compilador del lenguaje C++ para poder generar un programa ejecutable (o código objeto). Dicho ejecutable puede ser ejecutado directamente por el computador subyacente, a través del sistema operativo. En el curso usaremos principalmente el compilador `g++` de Linux, que permite compilar desde la línea de comando de un terminal usando:

```
g++ test-fuente.cpp -o test-objeto
```

El compilador generará un código objeto (ejecutable) llamado `test-objeto`, a partir del código fuente `test-fuente.cpp`.

Luego, podemos ejecutar nuestro programa desde la terminal usando:

```
./test-objeto
```

5 Tipos de Datos

Un tipo de datos está formado por un conjunto de valores de un cierto tipo, junto a un conjunto de operaciones para manipularlos. Por ejemplo, podemos pensar en valores de tipo entero junto con las operaciones aritméticas para manipularlos (por ejemplo, suma, resta, multiplicación, división, entre otras). Revisaremos en esta sección los tipos de datos simples primitivos de C++, junto con el conjunto de operadores que provee el lenguaje para manipular variables de dichos tipos.

5.1 Nombres de Variables y Declaraciones

En C++ (al igual que en C), es obligatorio declarar las variables antes de su uso. Esta es una de las carácterísticas distintivas del lenguaje. Una sentencia de declaración tiene la forma:

$$\langle \textit{tipo de datos} \rangle \ \textit{nombre de variable};$$

en donde $\langle \textit{tipo de datos} \rangle$ debe ser reemplazado por el tipo deseado para la variable, y *nombre de variable* es el nombre de la variable. Una vez declarada una variable de un cierto tipo, éste no va a cambiar durante el ciclo de vida de la variable. Por ejemplo, la sentencia

```
int x;
```

declara una variable `x` de tipo entero (denotado `int` en C++). El compilador usa esta información para reservar el espacio de almacenamiento en memoria para la variable `x`. El punto y coma final indica el final de la sentencia. Es posible además declarar varias variables del mismo tipo en una única sentencia de declaración, tal como a continuación:

```
int x, y;
```

la cual declara dos variables `x` e `y` de tipo entero. El lenguaje C++ define varios tipos primitivos, a partir de los cuales se pueden construir cosas más complejas y que estudiamos a continuación.

5.2 Los Tipos Primitivos Fundamentales

Estudiamos ahora los tipos primitivos fundamentales provistos por C++. Es importante notar que cada tipo fundamental corresponde a alguna representación en el hardware del computador, que mencionaremos brevemente en cada caso.

5.2.1 El Tipo `int` y sus Variantes

El tipo entero básico de C++ es `int`, que ya usamos en un ejemplo anterior. Aunque el estándar del lenguaje establece que los compiladores deben usar al menos 16 bits para implementar una variable de este tipo, es común que este tipo se implemente usando 32 bits (es decir, 4 bytes). Eso significa 32 dígitos binarios para representar un número entero. Eso limita los valores que una variable de este tipo puede tomar al intervalo $[-2^{31}, 2^{31} - 1]$. Para ilustrar por qué ocurre esto, piense en un caso más típico. Si a usted le permiten escribir números enteros positivos usando, por ejemplo, sólo 3 dígitos decimales, entonces podrá representar los valores $000 = 0$, $001 = 1$, $002 = 2$, y así siguiendo hasta el 999. Es decir, el rango de valores válidos en ese ejemplo es $[0, 999]$.

El siguiente ejemplo declara tres variables enteras. Notar que la última es inicializada en la misma declaración:

```
int i,j;
int var = 12;
```

La variable `var` es creada e inicializada con valor 12.

Es posible usar los calificadores `short` y `long` para obtener variables enteras con menor o mayor rango de valores, respectivamente. Por ejemplo:

```
short int a;
long int b;
```

Aunque nuevamente depende de la implementación del compilador, usualmente las variables `short int` se implementan usando 16 bits, mientras que las de tipo `long int` usan 64 bits [3]. Para simplificar la escritura, `short int x` es equivalente a `short x`. Lo mismo ocurre con el modificador `long`.

Una manera de saber cuántos bytes dedica un compilador para implementar un tipo dado es mediante el operador `sizeof`. Entonces, se podría hacer:

```
cout << "Tipo int: " << sizeof(int) << " bytes" << endl;
cout << "Tipo short int: " << sizeof(short int) << " bytes" << endl;
cout << "Tipo long int: " << sizeof(long int) << " bytes" << endl;
```

[3] Cuidado: usualmente ocurre eso, pero no está definido explícitamente en el estándar del lenguaje.

El operador `sizeof` puede emplearse con cualquier tipo de datos, y siempre devolverá la cantidad de bytes con las que se está implementando ese tipo. Es importante notar que la manera en que se implementa un tipo dado puede variar con los distintos compiladores y arquitecturas que existen. No es seguro asumir que un tipo va a ser siempre implementado con una cantidad de bytes dada.

En caso de ser necesario asegurar la cantidad de bits usados en la implementación de una variable entera, la biblioteca `cstdint` provee los tipos `int8_t`, `int16_t`, `int32_t`, e `int64_t`, correspondientes a enteros de 8, 16, 32, y 64 bits, respectivamente. Un ejemplo de uso es el siguiente:

```
#include <cstdint>

int16_t x;
int64_t y;
```

También es posible declarar variables enteras que pueden tomar sólo valores positivos, ampliando de esta manera el rango de valores positivos que pueden tomar:

```
unsigned int x;
unsigned short int y;
```

Si las variables enteras se implementan con 32 bits, entonces una variable de tipo `unsigned int` puede tomar valores en el rango $[0, 2^{32} - 1]$. Tal como antes, no es necesario usar la palabra `int` cuando se realizan estas declaraciones. Las declaraciones anteriores podrían haberse escrito:

```
unsigned x;
unsigned short y;
```

En `cstdint` también están las versiones sin signo especificadas anteriormente, es decir, `uint8_t`, `uint16_t`, `uint32_t`, y `uint64_t`.

5.2.2 El Tipo `char`

El tipo `char` permite representar carácteres individuales. Por ejemplo, si declaramos:

```
char v;
```

luego podemos asignar:

```
v = 'A';
```

Sin embargo, en C++ no existe una diferencia real entre los carácteres y los enteros. De hecho, el tipo `char` es un caso particular de un entero de 1 byte (8 bits). La asignación anterior podría perfectamente haberse escrito:

```
v = 65;
```

dado que el código ASCII del carácter '`A`' es 65. Se puede notar la equivalencia si se imprime el valor de la variable `v` como entero y como carácter, usando:

```
cout << "El código ASCII del carácter " << v << " es " << (int)v << endl;
```

lo cual mostrará por pantalla el mensaje:

```
El codigo ASCII del carácter A es 65
```

Aquí, la notación `(int)v` es conocida como cast en C++, y su función es la de convertir un tipo de datos en otro en una expresión [4]. En este caso, se indica que el valor de la variable `v` se interprete como un entero en esa expresión.

[4] Al aplicar un cast sobre una variable como en el ejemplo, el tipo original de la variable no cambia, sólo se interpreta como lo indica el cast.

5.2.3 Constantes Enteras

También es posible definir constantes `long`, las que deben ser utilizadas cuando el valor que se desea especificar está fuera del rango soportado por el entero común. Se escribe con una letra `l` (ele) o `L` siguiendo el número. Por ejemplo, suponiendo que en la implementación particular un `int` tiene 32 bits y un `long` 64 bits:

```
long x;
x = 1000;           /* en el rango entero */
x = 10834766975L;   /* fuera del rango entero */
x = 1000L;          /* la L no es necesaria, pero tampoco es un error */
```

También existen constantes `unsigned`, las que se especifican con una `u` o `U` siguiendo al número. Por ejemplo:

```
unsigned y;
y = 455u;
```

Las constantes enteras se pueden escribir en distintas notaciones. Una secuencia de dígitos que no comienza con 0 es considerada un entero decimal, si comienza con un 0 (cero) es considerada una constante octal, y si comienza con la secuencia 0x (cero equis) una constante hexadecimal. Por ejemplo:

```
int i;
i = 255;     /* 255 decimal */
i = 0xff;    /* ff hexadecimal = 255 decimal */
i = 0xf3;    /* f3 hexadecimal = 243 decimal */
i = 027;     /* 27 octal = 23 decimal */
```

5.2.4 Combinando Enteros

Al existir varias clases de enteros, existen reglas para determinar cuál es el tipo y el valor del resultado de operaciones que los combinen. Ellas son:

1. Los carácteres (`char`) y los enteros cortos (`short`) son convertidos automáticamente a enteros comunes (`int`) antes de evaluar.
2. Si en la expresión aparece un entero largo (`long`) el otro argumento es convertido automáticamente a entero largo (`long`).
3. Si aparece un entero sin signo (`unsigned`) el otro argumento es convertido automáticamente a entero sin signo (`unsigned`).

Suponga que se tienen las siguientes definiciones:

```
int a = 5;
long b = 50000L;
char c = 'A';
unsigined d = 33;    // o 33u, es lo mismo
```

Las siguientes expresiones se evalúan de la siguiente forma:

expresión	resultado	tipo	reglas usadas
`a + 5`	10	`int`	–
`a + b`	50005	`long`	2
`a + c`	70	`long`	1
`a + b + c`	50070	`long`	1, 2
`a + d`	38	`unsigned`	3

5.2.5 El Tipo `float`

El tipo `float` permite almacenar números reales. Existen tres variantes de `float` en C++. El tipo estándar se denomina `float`, mientras que las versiones `double` y `long double` permiten representar valores reales con mayor precisión. Al igual que con los enteros, la precisión de cada uno depende de la implementación. Sólo se asegura que un `double` no tiene menor precisión que un `float`, y que un `long double` no tiene menor precisión que un `double`. Un ejemplo de uso de flotantes es el siguiente:

```
float f, g = 5.2;

f = g + 1.1;
cout << f << endl;    /* imprime 6.3 */
```

5.2.6 Combinando Flotantes con Enteros

Se debe tener cuidado al mezclar enteros con flotantes, ya que se pueden obtener resultados inesperados en algunos casos. Considere el siguiente ejemplo, el cual imprime 3 y no 3.5 como podría esperarse:

```
float f;
f = 7 / 2;
cout << f << endl;    /* imprime 3 */
```

La razón es que ambos argumentos de la división son enteros, por lo que se aplica la operación de división entera, produciendo 3 como resultado. Una posible solución a este problema es forzar a que uno de los argumentos sea flotante. C++ interpretará entonces que se desea realizar una división de flotantes, por ejemplo:

```
float f;
f = 7 / 2.0;
cout << f << endl;    /* imprime 3.5 */
```

5.2.7 El Tipo `bool`

El tipo booleano `bool` permite almacenar valores de verdad lógicos. Esto es, una variable de tipo `bool` puede almacenar uno de dos valores posibles: `true` (verdadero) o `false` (falso). Cabe notar que en C++ cualquier valor numérico *igual* a 0 se considerará falso, mientras que cualquier valor *distinto* de 0 se considerará verdadero.

En el siguiente ejemplo, al mostrar por pantalla el valor de la variable `b` de tipo `bool`, se imprime 1 ya que es el que corresponde al valor `true`:

```
bool b = true;
cout << b << endl;  /* imprime 1 */
```

5.2.8 Conversión de Tipos

Tal como se ha visto en las Secciones 5.2.4 y 5.2.6, se debe tener especial cuidado cuando se combinan operandos de tipos distintos. Como regla general, si un argumento "más chico" debe ser operado con uno "más grande", el "más chico" es primero transformado y el resultado corresponde al tipo "más grande". Por ejemplo, si un `int` es sumado a un `long`, el resultado será `long`, dado que es el tipo "más grande". C++ permite que valores "más chicos" sean asignados a variables "más grandes" sin objeción, por ejemplo:

```
float f = 3;      /* 3 es un entero y f float */
double g = f;     /* f es float y g double */
```

También se permite que valores de un tipo "más grande" sean asignados a variables "más chicas", aunque dado que esta situación podría dar lugar a errores, el compilador emite un mensaje de advertencia (*warning*). Realmente no se recomienda realizar este tipo de operación.

```
long x = 10;
int i = x;       /* permitido, pero peligroso */
```

A fin de que el programador tenga una herramienta para indicar la clase de conversión de tipo que desea, el lenguaje provee una construcción para forzar un cambio de tipo. Como ya lo hemos mencionado anteriormente, se denomina *cast*, y consiste en especificar entre paréntesis el tipo al que se desea convertir una expresión. Por ejemplo, para que el compilador no emita una advertencia en el ejemplo anterior se puede hacer:

```
long x = 10;
int i = (int)x;
```

El cast (`int`) fuerza el valor de `x` a tipo entero. Si bien aun podría producirse una pérdida de información, el programador demuestra que es consciente de ello.

Para el caso de la división planteado en la Sección 5.2.6, el problema podría ser resuelto con un *cast* como sigue:

```
float f;
f = (float)7 / 2;
cout << f << endl;     /* imprime 3.5 */
```

Al ser el 7 convertido a `float` por el *cast*, la operación de división se realiza entre flotantes, y el resultado es 3.5

5.3 Operadores

Al igual que C, el lenguaje C++ provee un conjunto extremadamente rico de operadores. Es una carácterística que lo distingue de la mayoría de los lenguajes. Una desventaja, sin embargo, es que contribuyen a aumentar la complejidad. Estudiamos a continuación los operadores de uso más común.

5.3.1 Operadores Aritméticos

Los operadores aritméticos son los siguientes:

operador	descripción
`+`	suma
`-`	resta
`*`	multiplicación
`/`	división
`%`	módulo

Se debe tener en cuenta que las operaciones entre enteros producen enteros, tal como fue mostrado en los ejemplos de la Sección 5.2.6.

Cuando un argumento es negativo, la implementación del lenguaje es libre de redondear hacia arriba o hacia abajo. Esto significa que con distintos compiladores es posible obtener resultados distintos para la misma expresión 2. Por ejemplo, la siguiente expresión puede asignar a `i` el valor -2 o el valor -3:

```
i = -5 / 2;
```

Algo similar ocurre con la operación de módulo, que puede asignar a `i` el valor 1 o -1, quedando el signo del resultado dependiente de la implementación particular.

```
i = -5 % 2;
```

5.3.2 Operadores Relacionales

Los operadores relacionales son los siguientes:

operador	descripción
`<`	menor
`>`	mayor
`<=`	menor o igual
`>=`	mayor o igual
`==`	igual
`!=`	distinto

En C++ las expresiones relacionales entregan un valor booleano. Por ejemplo las siguientes expresiones tienen los siguientes valores:

`1 < 5` → `true` (verdadera)
`4 != 4` → `false` (falsa)
`2 == 2` → `true` (verdadera)

Como se indicó en la Sección 5.2.7, cualquier valor igual a cero es considerado falso y un valor distinto de cero es considerado como verdadero. La siguiente condición:

```
if (i != 0) ...
```

es entonces equivalente a:

```
if (i) ...
```

ya que ambas condiciones son consideradas verdaderas cuando `i` es distinto de cero, y serán ambas falsas solo cuando `i` sea igual a cero.

El resultado de una expresión relacional se transforman a enteros si se usan en expresiones aritméticas. Por ejemplo:

`10 + (3 < 4)` → `11`
`40 - (4 == 5)` → `40`

5.3.3 Operadores Lógicos

Los operadores lógicos son los siguientes:

operador	descripción
`&&`	and (binario)
`||`	or (binario)
`!`	not (unario)

Los operadores `&&` y `||` funcionan en "corto circuito": se garantiza que el segundo operando sólo es evaluado si es necesario. Es decir, el segundo operando de `&&` es evaluado sólo si el primero es verdadero, ya que si fuera falso el resultado de todo el `&&` no depende de la evaluación del segundo. Algo similar ocurre con `||`, en donde el segundo operando se evalúa sólo si el primero es falso. Considere el siguiente ejemplo:

`false && ...` → `false`
`true || ...` → `true`

Por ejemplo,

```
if(i != 0 && j/i > 5) ...
```

tiene esta carácterística. Si el valor de `i` es cero, no se evalúa la segunda condición, que podría ser problemática (ya que dividiría por cero).

5.3.4 Operadores de Asignación

En C++ la asignación es una operación y no una sentencia: el operador de asignación retorna el valor que es asignado. Por ejemplo, si tenemos:

```
int a = 2, b, c;
c = (b = a + 1) + 4;
```

a la variable `b` se le asigna el valor 3. Este valor es retornado por la asignación interna, el cual sumado a 4 y es asignado finalmente a la variable `c` (que recibe el valor 7).

Al ser combinado con otros operadores, da lugar a toda una familia de operadores de asignación, que permite escribir las operaciones en forma mucho más compacta, por ejemplo:

normal	compacta
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`

5.3.5 Operadores de Incremento y Decremento

Para incrementar y decrementar el valor de una variable de tipo entero (o alguna de sus variantes) se pueden utilizar respectivamente los operadores de incremento `++` y de decremento `--`. Por ejemplo las siguientes expresiones son equivalentes:

normal	compacta
`i = i + 1`	`i++`
`i = i - 1`	`i--`

Estos operadores se pueden utilizar en forma prefijo (`++i`) o postfijo (`i++`), con significados distintos. En forma prefijo primero se realiza la operación de incremento (o decremento, según corresponda) y luego se entrega el resultado (es decir, la expresión retorna el valor de la variable incrementada). En forma postfijo, primero se entrega el resultado de la variable y luego se incrementa (o decrementa) el valor de la variable (es decir, la expresión retorna el valor de la variable sin incrementar). Por ejemplo, las siguientes expresiones son equivalentes:

compacta	normal
`a = i++;`	`a = i;` `i = i + 1;`
`a = ++i;`	`i = i + 1;` `a = i;`
`a = --i + 5;`	`i = i - 1;` `a = i + 5;`
`a = i++ + 5;`	`a = i + 5;` `i = i + 1;`

5.3.6 Operador Condicional

C++ provee un operador muy útil denominado operador condicional, cuya sintaxis es `?:`, que permite expresar operaciones similares a la selección (`if`) pero dentro de expresiones. El operador toma tres expresiones como argumentos. La primera es una condición, la cual una vez evaluada determinará cual de las otras expresiones será evaluada para calcular el resultado. Si la condición es verdadera, la segunda es evaluada, en caso contrario la tercera es evaluada. Por ejemplo,

```
a = (i < j) ?  i : j;
```

asigna a la variable `a` el valor de la variable `i` si `i < j`, y el de `j` en caso contrario. Es decir, en otras palabras, asigna el menor valor. También se puede utilizar en la parte izquierda de la asignación:

```
(i != j) ? a : b = 0;
```

asigna el valor `0` a la variable `a` si `i != j` es distinto al valor de la variable `j`, en otro caso asigna `0` a la variable `b`.

6 Control de Secuencia

Como todo lenguaje de programación, C++ provee de diversas construcciones que permiten controlar el flujo de ejecución de un programa. Estudiamos a continuación las más típicas.

6.1 La Sentencia de Selección `if`

La forma general de la sentencia básica de selección `if` es la siguiente:

```
if (<condición>)
    <sentencia>
```

Si la condición es verdadera, la sentencia es ejecutada. Por ejemplo, asumiendo que la variable `ganancia` ha sido adecuadamente declarada y se le ha dado un valor:

```
if (ganancia < 0)
    cout << "no se gana nada" << endl;
```

En lugar de una sentencia, se puede poner un bloque formado por sentencias encerradas entre llaves. Por ejemplo:

```
if (ganancia < 0) {
    veces++;
    cout << "no se gana nada" << endl;
}
```

También se puede incorporar una sentencia a ser ejecutada en caso que la condición sea falsa, con la siguiente forma general:

```
if (<condición>)
    <sentencia1>
else
    <sentencia2>
```

Por ejemplo:

```
if (ganancia < 0)
    cout << "no se gana nada" << endl;
else
    cout << "se gana " << ganancia << " pesos" << endl;
```

6.2 La Sentencia de Selección `switch`

La sentencia `switch` permite la selección con múltiples ramas. Su forma general es la siguiente:

```
switch(<expresión>) {
    case <cte1>: <sentencias>
    case <cte2>: <sentencias>
    ...
    default: <sentencias>
}
```

Primero se evalúa `<expresión>`, obteniendo un valor v. Luego se ejecuta el grupo de sentencias asociado a la constante v. Si el valor no coincide con ninguna constante, se ejecuta el grupo de sentencias asociadas a la cláusula `default`. La cláusula `default` es opcional, por lo que si no se ha especificado y el valor de la expresión no coincide con ninguna constante, la sentencia `switch` no hace nada. Es importante aclarar que una vez que una alternativa es seleccionada, las siguientes se continúan ejecutando secuencialmente, independientemente del valor de su constante. Por ejemplo, el siguiente trozo de programa imprime `uno`, `dos` y `tres` si el valor de la variable `i` es 1, imprime `dos` y `tres` si el valor es 2, e imprime `tres` si el valor es 3:

```
switch (i) {
    case 1: cout << "uno" << endl;
    case 2: cout << "dos" << endl;
    case 3: cout << "tres" << endl;
}
```

Para evitar este efecto se puede utilizar la sentencia `break`, que fuerza al flujo de ejecución salir de la sentencia `switch`, por lo que las siguientes alternativas no serán ejecutadas. Por ejemplo, el siguiente trozo de programa imprime `uno` si el valor de la variable `i` es 1, imprime `dos` si el valor es 2, e imprime `tres` si el valor es 3.

```
switch (i) {
    case 1: cout << "uno" << endl;
            break;
    case 2: cout << "dos" << endl;
            break;
    case 3: cout << "tres" << endl;
}
```

No se permiten múltiples valores para una alternativa, pero su efecto se puede simular aprovechando el hecho que la ejecución continúa cuando se ha encontrado la constante adecuada y no existen sentencias `break`. Por ejemplo:

```
switch (ch) {
    case ',':
    case '.':
    case ';': cout << "signo de puntuacion" << endl;
              break;
    default : cout << "no es signo de puntuacion" << endl;
}
```

Las expresiones deben ser de tipo entero, carácter o enumeración (explicada más adelante).

6.3 La Sentencia de Iteración `while`

Su forma básica es la siguiente:

```
while (<condición>)
    <sentencia>
```

La `<sentencia>` se ejecutará mientras la `<condición>` sea verdadera. Si la `<condición>` es falsa inicialmente, no se ejecuta la sentencia y la ejecución continúa luego del fin del `while`.

Por ejemplo, el siguiente programa recibe un número entero por entrada estándar y luego muestra todos los números hasta llegar a 0.

```
int main() {
    int a;
    cin >> a;
    while (a >= 0) {
        cout << a << " ";
        a--;
    }
    cout << endl;
    return 0;
}
```

El programa muestra el valor de la variable `a` y luego la decrementa en uno. Al inicio de cada iteración del `while` verifica si es mayor o igual a cero. Esta condición será falsa apenas `a` sea negativo, en cuyo caso saldrá del (o evitará entrar nuevamente al) `while` e imprimirá un salto de línea antes de finalizar.

6.4 La Sentencia de Iteración `do-while`

Su forma básica es la siguiente:

```
do
    <sentencia>
while (<condición>)
```

La sentencia se ejecuta hasta que la condición sea falsa. Notar que se ejecuta al menos una vez, a diferencia del `while` visto en la subsección anterior, cuyas sentencia pueden no ejecutarse nunca.

El mismo ejemplo anterior reescrito con la sentencia `do-while` es:

```
int main() {
    int a = 0;
    cin >> a;
    do {
        cout << a << " ";
        a--;
    } while(a >= 0);
    cout << endl;
    return 0;
}
```

En este caso, siempre se mostrará por salida estándar al menos el número inicial sin importar su valor.

6.5 La Sentencia de Iteración `for`

La sentencia `for` es una sentencia de iteración que permite reescribir en forma más compacta expresiones comunes del tipo:

```
<sentencia inicial>
while (<condicion>) {
    <sentencia cuerpo>
    <sentencia iteracion>
}
```

de la siguiente forma:

```
for (<sentencia inicial>; <condicion>; <sentencia iteracion>)
    <sentencia cuerpo>
```

en las que la sentencia inicial prepara el comienzo de la iteración (inicializa variables, etc), la condición controla si se debe o no realizar una nueva iteración, y la sentencia de iteración es la que realiza las acciones necesarias para entrar en una nueva iteración. Por ejemplo, la siguiente iteración `while` imprime los enteros de 1 a 10:

```
i = 1;
while (i <= 10) {
    cout << i << endl;
    i++;
}
```

Se puede reescribir utilizando `for` de la siguiente manera:

```
for (i = 1; i <= 10; i++)
    cout << i << endl;
```

La primera expresión del `for` es ejecutada primero, luego la segunda expresión (o condición) es evaluada. Si es verdadera, se ejecuta el cuerpo y luego la tercera expresión. El proceso continúa hasta que la segunda expresión se vuelva falsa.

Las expresiones del `for` pueden ser omitidas, por ejemplo, todos los siguientes trozos de programa son equivalentes al ejemplo anterior:

```
i = 1;
for (; i <= 10; i++)
    cout << i << endl;

for (i = 1; i <= 10;)
    cout << i++ << endl;

i = 1;
for (; i <= 10;)
    cout << i++ << endl;
```

Al omitirse la condición se asuma verdadera siempre.

El operador coma (`,`) es muy útil cuando se combina con la sentencia `for`. El operador coma toma dos expresiones como argumento, evalúa las dos, y sólo retorna el valor de la segunda. Es útil para colocar dos expresiones, donde sólo se admite una. Por ejemplo:

```
for (i = 0, j = 5; i < j; i++,j--)
    cout << i << " " << j << endl;
```

imprimirá:

```
0 5
1 4
2 3
```

Notar que la primera y la última expresión del `for` son reemplazadas por dos expresiones cada una.

6.6 Las Sentencias `break` y `continue`

La sentencia `break` tiene dos usos. Uno ya fue explicado en la Sección 6.2 en relación con el `switch`. Cuando se utiliza dentro una sentencia de iteración (`for`, `do-while` o `while`), causa que se salga de la iteración.

Por ejemplo, el siguiente programa lee un número desde entrada estándar y lo divide a la mitad hasta alcanzar 1. Se puede utilizar el `break` para salir inmediatamente si se entrega un número no positivo):

```
int main() {
    int a;
    cin >> a;
    while (a != 1) {
        cout << a << " ";
        if (a <= 0) break;
        a--;
    }
    cout << endl;
```

```
    return 0;
}
```

La sentencia `continue`, que puede ser utilizada únicamente dentro de una iteración, causa que se abandone la iteración actual, y se comience una nueva.

Por ejemplo el siguiente programa recibe un número por entrada estándar y lo decrementa hasta alcanzar un número negativo, mostrando por pantalla solo los números pares.

```
int main() {
    int a;
    cin >> a;
    for(; a >= 0; a--){
        if( a % 2 != 0 ) continue;  // se salta los impares
        cout << a << " ";
    }
    cout << endl;
    return 0;
}
```

Notar que la sentencia `a--` es ejecutada dado que siempre se ejecuta al terminar una vuelta de la iteración, ya sea en forma normal, o porque la terminación ha sido forzada por el `continue`.

Una buena utilización de las sentencias `break` y `continue` evita el uso de ramas muy grandes en selecciones y el uso de *flags* en las iteraciones.

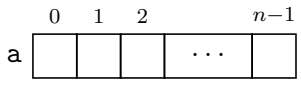
7 Objetos de Datos Estructurados

Estudiamos a continuación cómo manejar datos estructurados en C++, los cuales permiten definir tipos de datos mas complejos (o elaborados), que son necesarios en la práctica.

7.1 Arreglos

Los arreglos son quizás las estructuras de datos más básicas (y fundamentales) que estudiaremos. Son provistos como tipo de datos por la mayoría de los lenguajes de programación, y permiten almacenar un conjunto de datos en memoria principal. Una carácterística importante es que todos los datos almacenados en un arreglo (conocidos como los *elementos* o las *componentes* del arreglo) deben ser del mismo tipo, conocido como el *tipo base* del arreglo. Llamaremos *tamaño* de un arreglo a la cantidad de componentes que tiene.

Otra carácterística distintiva de los arreglos es que sus componentes son almacenadas de manera contigua en la memoria. Por lo tanto, podemos hablar de que existe una primera componente, una segunda componente, y así siguiendo, cada uno de las cuales se identifica usando un índice entero. De forma gráfica, un arreglo `a` de n componentes tiene la forma:



Cada una de las casillas corresponde a una componente del arreglo, las cuales en C++ son identificadas con los subíndices 0 a $n-1$.

Un arreglo `a` de tamaño 3 y tipo base `int` se define en C++ como a continuación:

```
int a[3];
```

Esta definición reserva lugar para tres componentes enteras, las cuales se pueden acceder mediante el operador `[]` de la siguiente manera: el primer elemento del arreglo es `a[0]`, el segundo es `a[1]`, y el tercero `a[2]`. Cada uno de esos es, por sí mismo, un entero, por lo que se puede operar con ellos de la forma habitual. Como ya lo dijimos, los subíndices de los arreglos en C++ comienzan siempre desde 0, por lo que el índice de la última componente de un arreglo de tamaño n es $n-1$.

El siguiente programa inicializa todos los elementos de un arreglo `a` con `0`:

```
int main() {
    int a[3], i;
    for (i = 0; i < 3; i++)
        a[i] = 0;
    return 0;
}
```

Un arreglo puede ser inicializado en la misma definición, por ejemplo:

```
int a[3] = {4, 5, 2};
float b[] = {1.5, 2.1};
char c[10] = {'a', 'b'};
```

El arreglo `a` tiene tres componentes enteras inicializadas con los valores `4`, `5` y `2` respectivamente. El arreglo `b` tiene dos componentes flotantes inicializadas con los valores `1.5` y `2.1`, respectivamente. El tamaño del arreglo `b`, que no fue especificado en la declaración se obtiene de los datos. El arreglo `c` tiene 10 componentes de tipo carácter, de las cuales sólo la primera y la segunda han sido inicializadas, con los valores `'a'` y `'b'` respectivamente.

7.2 Estructuras

Las estructuras son conjuntos de una o más variables que pueden ser de igual o distinto tipo, agrupadas bajo un solo nombre. Las estructuras ayudan a organizar datos complejos, en particular dentro de programas grandes, debido a que permiten que a un grupo de variables relacionadas se les trate como una unidad en lugar de como entidades separadas.

Por ejemplo, se podría necesitar mantener en un programa los datos de puntos en el plano, en donde a cada punto le corresponden las coordenadas x e y representadas como números flotantes, y además un peso almacenado como un valor entero. Entonces, en lugar de declarar una variable por cada uno de esos atributos, se puede definir una estructura en C++ que englobe todos estos valores en una única estructura que represente (de alguna manera) el concepto de punto.

A diferencia de un arreglo, en el que todos los elementos son del mismo tipo y se acceden a través de un índice entero, en la estructura los elementos pueden ser de tipos distintos y se acceden a través de un nombre. La forma general de la declaración de una estructura es la siguiente:

```
struct <nombre-estructura> {
    <tipo> <nombre-campo>;
    <tipo> <nombre-campo>;
    ...
};
```

La palabra clave `struct` es la que permite definir una estructura. Entonces, para el ejemplo de los puntos antes mencionado, se podría hacer:

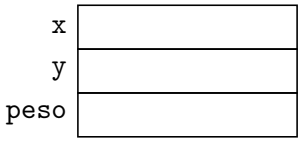
```
struct punto {
    float x;
    float y;
```

```
    int peso;
};
```

El identificador `punto` es el nombre de la estructura. En C++, esta definición introduce un nuevo tipo de datos (en este caso, el tipo `punto`), que puede ser utilizado posteriormente para declarar variables, como por ejemplo:

```
punto p1;
```

que declara una variable de tipo `punto` llamada `p1`. La variable `p1` tiene la estructura definida más arriba, con sus dos coordenadas de tipo `float` y el peso de tipo `int`. Gráficamente, podemos representar a esta variable de la siguiente manera:



Cada elemento de una estructura se conoce como *miembro* o *campo*. El acceso a los campos o miembros de una variable de tipo estructura se hace de la siguiente manera:

```
<nombre_estructura>.<nombre_campo>
```

Aquí, '`.`' es el operador conocido como *selector de campo*. Por ejemplo:

```
p1.x = 3.6:
p1.y = 2.1;
p1.peso = 7;
```

Las variables de tipo `struct` también pueden inicializarse en la misma declaración, como por ejemplo:

```
punto p1 = {3.6, 2.1, 7};
```

lo que le asigna el valor indicado a cada uno de los campos de la estructura, respetando el orden en que han sido definidos.

7.3 Combinación

Los arreglos y las estructuras pueden ser combinadas de forma arbitraria, permitiendo definir tipos de datos complejos que pueden aparecer en la práctica. Por ejemplo, un arreglo de 10 estructuras `punto` se define como sigue:

```
punto p[10];
```

asumiendo la definición del `struct punto` dada anteriormente. Las componentes `p[0]`, `p[1]`, ..., `p[9]` del arreglo son puntos, por lo que se pueden acceder de la forma esperada:

```
p[0].x = 3.6;
p[0].y = 2.1;
p[0].peso = 7;
p[1].x = 6.3;
p[1].y = 2.9;
p[1].peso = 3;
```

De esta forma se le asignan valores (arbitrarios) a los campos de los puntos `p[0]` y `p[1]`.

Las `struct` también pueden contener arreglos. El siguiente ejemplo define un punto en un espacio de 10 dimensiones, asumiendo además que cada punto tiene asociado un peso entero:

```
struct punto_10D {
    float coord[10];
    int peso;
};
```

Si luego declaramos una variable:

```
punto_10D p2;
```

entonces podemos hacer:

```
p2.peso = 7;
p2.coord[0] = 3.6;
...
p2.coord[9] = 4.9;
```

A continuación estudiamos un ejemplo más complejo: una base de datos (aunque muy primitiva) de alumnos. La idea es almacenar 1000 alumnos, y para cada uno de ellos se busca mantener su rol, su fecha de nacimiento, y la lista de cursos que ha tomado (junto con la cantidad de estos). Eso podría representarse en C++ de la siguiente manera:

```
struct {
    int rol;
    struct {
        int dia;
        int mes;
        int agno;
    } nacimiento;
    struct {
        int sigla;
        int nota;
    } cursos[50];
    int tomados;
} bd_alumnos[1000];
```

Aquí, `bd_alumnos` es un arreglo de 1000 alumnos, cada uno de los cuales está representado por los siguientes 4 campos: `rol`, `nacimiento`, `cursos`, y `tomados`. A su vez, el campo `nacimiento` es un `struct` de 3 campos (`dia`, `mes`, y `agno`), mientras que el campo `cursos` es un arreglo de 50 `structs`, cada uno de los cuales tiene dos campos (`sigla` y `nota`). Para ilustrar cómo manipular este tipo de datos, a continuación le asignamos valores al alumno `bd_alumnos[4]`:

```
bd_alumnos[4].rol = 20182436;
bd_alumnos[4].nacimiento.dia = 4;
bd_alumnos[4].nacimiento.mes = 9;
bd_alumnos[4].nacimiento.agno = 1986;
bd_alumnos[4].tomados = 2;  // el alumno tomó 2 cursos
bd_alumnos[4].cursos[0].sigla = 134;
bd_alumnos[4].cursos[0].nota = 65;
bd_alumnos[4].cursos[1].sigla = 221;
bd_alumnos[4].cursos[1].nota = 59;
```

8 Funciones

8.1 Definición y Utilización

Una función es un conjunto de declaraciones, definiciones, expresiones y sentencias que realizan una tarea específica. Para la definición (o creación) de una función en C++, el formato general a seguir es el siguiente:

```
<especificador-de-tipo> <nombre-de-función> (<lista-de-parámetros>) {
    <variables locales de la función>
    <código de la función>
}
```

El `<especificador-de-tipo>` indica el tipo del valor que la función devolverá (o) mediante el uso de la sentencia `return`. También puede especificar que no se devolverá ningún valor utilizando el tipo `void`. El valor, si fuera diferente a `void`, puede ser de cualquier tipo válido. Si no se especifica, entonces el procesador asume por defecto que la función devolverá un resultado entero.

No siempre se deben incluir parámetros en una función. En consecuencia, se asume que la `lista-de-parámetros` puede estar vacía.

Las funciones terminan y regresan automáticamente a la función invocante cuando se encuentra la última llave }, o bien, se puede forzar el regreso anticipado usando la sentencia `return`. Esta sentencia permite, además, devolver un valor a la función invocante. El siguiente ejemplo muestra el cálculo del promedio de dos números enteros mediante el uso de funciones:

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

float calcProm(int num1, int num2) {
    float promedio;
    promedio = (num1 + num2) / 2.0;
    return promedio;
}

int main() {
    int a = 7, b = 10;
    cout << "Promedio = " << calcProm(a, b) << endl;
    return 0;
}
```

8.2 Parámetros Formales y Parámetros Reales (o Actuales)

En el ejemplo anterior, la función `calcProm` emplea dos parámetros identificados como `num1` y `num2` ambos de tipo entero (`int`). Dichas variables se denominan *parámetros formales*. Al igual que las variables definidas en el cuerpo de la función, los parámetros formales representan variables locales a la misma, cuyo uso está limitado a la propia función. Por lo tanto, no existe problema alguno al utilizar nombres duplicados de variables en otras funciones.

Es importante notar que el lenguaje C++ requiere que cada parámetro formal vaya precedido por su tipo en el encabezado de la función. Será un encabezado no válido si se escribe:

```
float calcProm(int num1, num2)
// ^ No válido!
```

En el ejemplo de la Sección 8.1, cuando se invoca a la función `calcProm` desde la función `main`, las variables `a` y `b` representan los *parámetros reales* o *actuales*. Los contenidos de estas variables se copian a los correspondientes parámetros formales de `calcProm`, en este caso `num1` y `num2`.

Así, un parámetro formal actúa de forma similar a una variable en la función en donde ha sido declarado. Dicha "variable" es inicializada al momento de la invocación con el valor del correspondiente parámetro real. En otras palabras, todo parámetro real es evaluado en la invocación de la función, y su resultado es pasado a la función mediante los parámetros formales. Los parámetros reales pueden ser una constante, variable o incluso una expresión más elaborada que forma parte de la invocación.

Por ejemplo, consideremos la invocación

```
calcProm(a+5, b);
```

Aquí, la expresión del primer parámetro es evaluada y su resultado es asignado al parámetro formal `num1`. El valor de la variable `b`, entregada como segundo parámetro, es asignada al parámetro formal `num2`.

8.3 Devolución de un Valor desde una Función

Tal como hemos dicho, a partir de los parámetros se puede comunicar información desde una función que invoca, a otra que recibe la llamada (en el ejemplo anterior, sería desde la función `main` a la función `calcProm`). Para enviar información en la dirección contraria, se utiliza el *valor de retorno* de la función. La sentencia `return` hace que un valor se transmita como resultado de una función. En el ejemplo anterior, el valor a retornar es el de la variable `promedio`. Una función que posea valor de retorno deberá declararse con el mismo tipo que tiene dicho valor. En el ejemplo, el tipo de la variable `promedio` es `float`, por lo tanto el tipo de la función `calcProm` también deberá ser `float`.

Es importante tener en cuenta que la declaración de tipo forma parte de la definición de la función, refiriéndose a su valor de retorno y no a sus parámetros, así en el encabezado siguiente:

```
float calcProm(int num1, int num2)
```

se está definiendo una función que acepta dos parámetros de tipo `int`, y que devuelve un valor de tipo `float`. En ese caso diremos que la función es de tipo `float`, refiriendo al tipo de su valor de retorno. Al momento de invocar a una función que retorna un valor, es posible asignar el valor que retorna la función a alguna variable. Dicha variable deberá ser del mismo tipo que el de la función.

8.4 Funciones de Tipo `void`

Algunas funciones no necesitan retornar valores a la función que la invoca, sino que simplemente llevar a cabo algún proceso que no produce resultados que deban ser devueltos. En ese caso, el tipo de la función debe ser indicado como `void`. A continuación se muestra un ejemplo con una función de tipo `void` cuya única tarea es la de imprimir el resultado obtenido al ejecutar otra función.

8.5 Un Ejemplo más Complejo

A continuación estudiamos un ejemplo más elaborado:

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

```
int doble(int x) {
    return 2*x;
}

void mostrar(int x, int y) {
    cout << "El doble de " << x << " es " << y << endl;
}

int main() {
    int result;

    result = doble(5);
    mostrar(5, result);

    return 0;
}
```

Este programa incluye las funciones `doble` y `mostrar`, además de la función `main`. Dentro de la definición de la función `main`, aparece en la primera línea la declaración de una variable local llamada `result`, de tipo entero. Notar que las declaraciones de las variables locales aparecen dentro del cuerpo de la función (es decir dentro de las llaves). Esta variable, por ser local a la función `main`, sólo podrá ser utilizada dentro de ella. Cuando la función finalice su ejecución, todas sus variables locales son automáticamente destruidas por el sistema de ejecución.

La función `doble` se define a través de:

```
int doble(int x) { ... }
```

Esta definición indica que la función `doble` retorna un valor entero y toma como parámetro un entero `x`.

La función `mostrar` se define como:

```
void mostrar(int x, int y) { ... }
```

Esta definición indica que la función `mostrar` no retorna ningún valor (`void`) y toma como parámetros dos valores enteros `x` e `y`.

La función `doble` es invocada desde el programa principal (o función `main`) en la sentencia:

```
result = doble(5);
```

Aquí, se asigna a la variable `result` el valor entero retornado por la función `doble`, la cual es invocada con parámetro real 5. Tanto la variable `result` como la función `doble` son del tipo `int`. Al invocarla, el control es transferido a la función `doble`, por lo que el valor 5 es asignado al parámetro formal `x`. La función `doble` tiene una única sentencia en su cuerpo:

```
return 2 * x;
```

en donde el valor del parámetro formal `x` es multiplicado por 2, y ese resultado es retornado por la función. Al retornar a la función `main`, el resultado de la invocación es asignado a la variable local `result`. A continuación la función `mostrar` es invocada, y los parámetros formales `x` e `y` toman los valores de los parámetros reales: la constante 5 y el valor contenido en `result`, respectivamente. La función mostrar tiene como única sentencia en su cuerpo la de imprimir a través del stream `cout` de la siguiente forma:

```
std::cout << "El doble de " << x << " es " << y << std::endl;
```

obteniéndose en la pantalla:

```
El doble de 5 es 10
```

8.6 Parámetros por Referencia

En las funciones vistas anteriormente, al pasar como parámetro actual una variable, el valor de esta es copiada al parámetro formal. Esto es conocido como *paso de parámetros por copia* y cualquier cambio de valor sobre el parámetro formal no afecta al parámetro real. Esto ofrece comunicación desde la función invocante a la invocada. Para la dirección inversa solamente se tiene el valor de retorno, pero éste permite retornar solamente un único valor.

Para permitir que la función invocada modifique los valores de las variables usadas como parámetros reales, los parámetros formales deben ser declarados como *parámetros por referencia*. Con esto logramos comunicación bidireccional usando los parámetros.

Para definir un parámetro por referencia en C++, en la declaración del parámetro formal se debe agregar `&` antes del nombre del parámetro. En el siguiente ejemplo se muestra una función que intercambia los valores de dos variables pasadas por referencia, la típica operación swap:

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

void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int v1 = 2, v2 = 5;

    cout << "Pre-swap  v1 = " << v1 << ", v2 = " << v2 << endl;
    swap(v1, v2);
    cout << "Post-swap v1 = " << v1 << ", v2 = " << v2 << endl;
    return 0;
}
```

Los parámetros formales `a` y `b`, de la función `swap`, actúan como *aliases* de las variables `v1` y `v2`, respectivamente. Esto permite que el intercambio de valores que hace `swap` se refleje en las variables `v1` y `v2` de `main`.

8.7 Arreglos como Parámetros

En C++ es posible pasar arreglos como parámetros a una función, agregando simplemente `[]` al declarar el correspondiente parámetro formal. En el siguiente ejemplo, la función `duplicarElems` permite duplicar el valor de los elementos de un arreglo.

```
#include <iostream>

void duplicarElems(int array[], int length) {
    for(int i = 0; i < length; i++)
        array[i] *= 2;
}
```

```
int main() {
    int a[3] = {1, 3, 5};
    int b[5] = {2, 4, 8, 16, 32};

    duplicarElems(a, 3);
    duplicarElems(b, 5);
    return 0;
}
```

En C++ los arreglos se pasan automáticamente como referencia, por lo que cualquier modificación de sus elementos afectará al arreglo original.

Notar que `duplicarElems` recibe dos parámetros: el arreglo `array` de enteros y un entero `length` que indica el número de componentes del arreglo. Este segundo parámetro es necesario, ya que una vez que un arreglo es pasado como parámetro, no es posible conocer su número de componentes. Toda función en C++ que recibe uno o más arreglos como parámetros, debe recibir también un parámetro de tipo entero que especifique el tamaño de cada uno de los arreglos.

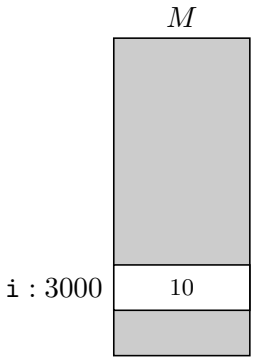
9 Punteros

La memoria de un computador está conformada por un conjunto de celdas, cada una de las cuales es capaz de almacenar un valor. Cada celda tiene una *dirección* que la identifica, que es simplemente un identificador entero. Por lo tanto, a nivel de hardware el procesador utiliza esas direcciones para almacenar o leer valores en las celdas de la memoria. Pare entender la idea de mejor manera, se podría pensar que la memoria es un gran arreglo conceptual de celdas $M[0..N-1]$, en donde N es la capacidad de la memoria. La dirección de memoria de la celda $M[i]$ es i.

Toda variable de un programa es almacenada en una (o más) celda de la memoria cuando el programa está en ejecución. Si una variable x (de un tipo dado) ha sido almacenada en la celda $M[i]$ en tiempo de ejecución, entonces diremos que x tiene dirección i. Por ejemplo, si se define una variable entera como sigue:

```
int i = 10;
```

en algún lugar de la memoria, por ejemplo en la dirección 3000, se reserva un lugar para almacenar los valores de esta variable. La situación en memoria es la siguiente:



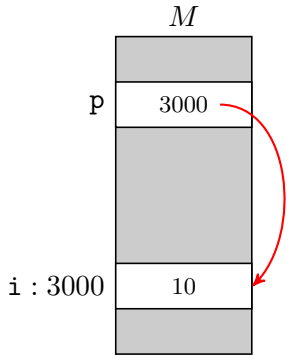
Un *puntero* (o *apuntador*) p es una variable que puede almacenar la dirección de memoria de otra variable i. En ese caso, diremos que p apunta a i. En C++, una variable puntero debe declarar el tipo de los valores a los que puede apuntar. Por ejemplo, un puntero a valores de tipo entero se define de la siguiente manera:

```
int *p;
```

En este caso, `*` indica que la variable que estamos declarando es un puntero. Para hacer que `p` apunte a la variable entera `i` declarada anteriormente, se debe hacer:

```
p = &i;
```

en donde `&` es el operador que permite obtener la dirección de memoria en la que está almacenada una variable. La situación en la memoria se muestra en la siguiente figura, donde se puede apreciar que el puntero `p` almacena la dirección de la variable `i`, o lo que es lo mismo, apunta a la variable `i`.



El valor apuntado por un puntero se puede obtener usando el operador de desreferenciación `*`, como se ilustra en el siguiente ejemplo:

```
cout << "El valor de la variable i es: " << i << endl; // Imprime 10
cout << "El valor apuntado por p es: " << *p << endl;  // Imprime 10
*p = 5;   // A la variable apuntada por p le asigna 5
cout << "El valor apuntado por p es: " << *p << endl; // Imprime 5
cout << "Que es igual al valor de la variable i: " << i << endl; // Imprime 5
```

Note cómo la variable `i` cambia de valor sin haberle asignado uno nuevo directamente: la asignación se hizo a través del puntero que la apunta. Aunque puede no quedar clara su utilidad en este momento, veremos más adelante que esto agrega gran poder al lenguaje.

Es posible también imprimir en pantalla la dirección de memoria de una variable, como a continuación:

```
cout << "La dirección de memoria de la variable i es: " << &i << endl;
```

Es importante notar que la dirección de memoria en la que se almacena una variable depende del sistema operativo, y puede cambiar de ejecución en ejecución. Por lo tanto, no es correcto ni seguro asumir que una variable siempre estará almacenada en una misma dirección de memoria.

Existe un puntero especial que no apunta a nada, y que es utilizado en general como marca (por ejemplo, será el que indica el final de un camino dentro de una estructura de datos, como estudiaremos más adelante en el curso). Dicho puntero es conocido como puntero nulo y se llama `NULL` en C++. Este valor puede asignarse a punteros de cualquier tipo, para indicar que aún no apunta a ningún valor, como por ejemplo:

```
int *p = NULL;
```

La constante `NULL` es definida en varias bibliotecas, como por ejemplo `cstddef`, `cstdio`, y `cstdlib`, por nombrar algunas de las más comúnmente utilizadas. Por lo tanto, alguna de esas debe ser incluida para poder usar la constante `NULL`.

9.1 Punteros y Arreglos

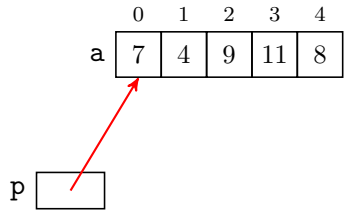
Los punteros y los arreglos están muy relacionados en C y C++. Todas las operaciones que se pueden realizar con arreglos se pueden realizar también usando punteros. Las siguientes sentencias declaran un arreglo de enteros y un puntero a enteros:

```
int a[5] = {7, 4, 9, 11, 8};
int *p;
```

Luego, la asignación

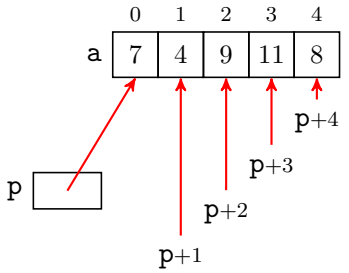
```
p = &a[0];
```

hace que `p` apunte a la primera componente del arreglo, `a[0]`. Dado que las componentes del arreglo están almacenadas de forma contigua en la memoria, esa asignación permite que se pueda acceder al arreglo completo usando a `p`. La situación en la memoria es la siguiente:



Algo importante en C y C++ es que el nombre de un arreglo usado en una expresión del lenguaje se evalúa a la dirección base del arreglo. Esto es, la dirección de la primera componente del arreglo. Por lo tanto, `p = &a[0]` es equivalente a `p = a`.

Por definición, si un puntero apunta a una componente de un arreglo, entonces `p+i` apuntará a `i` componentes más adelante.



Entonces, podemos hacer:

```
int a[5] = {7, 4, 9, 11, 8};
int *p;
int x;

p = a;
x = a[3];
cout << x << endl;  // imprime 11
x = *(p+3);
cout << x << endl;  // imprime 11
```

Esto es, `*(p+i)` es equivalente a `a[i]` cuando `p` apunta a la base del arreglo `a`. Además, para simplificar la notación `*(p+i)` es equivalente a `p[1]`, por lo tanto:

```
x = p[3];
cout << x << endl;  // imprime 11
```

Es decir, `*(p+i)` es equivalente a `p[i]`.

9.2 Punteros a Estructuras

También es posible apuntar a estructuras. Por ejemplo, si tenemos:

```
struct punto {
    float x;
    float y;
};
```

y luego declaramos la variable

```
punto s = {4.3, 2.8};
```

entonces podemos declarar un puntero a `punto` de la siguiente manera:

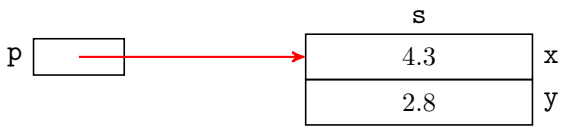
```
punto *p;
```

Luego, hacemos que el puntero `p` apunte a la estructura `s`:

```
p = &s;
```

A diferencia de los arreglos, el nombre de un `struct` no se evalúa a su dirección base, por lo que es necesario el operador `&` para apuntar a una estructura.

Luego de ejecutar las anteriores sentencias, la situación en la memoria es la siguiente:



Para modificar un campo de la estructura a través del puntero podemos hacer:

```
(*p).x = 5.3;
```

Los paréntesis en esta expresión no pueden omitirse, dado que el operador de selección de campo (`.`) tiene mayor precedencia que el operador de desreferenciación (`*`). El mismo comportamiento de estos dos operadores combinados se puede obtener por medio del operador `->`, de la siguiente forma:

```
p->x = 5.3;
```

lo cual es más simple y gráfico.

9.3 Aritmética de Punteros

Una de las características distintivas de C++ (y que hereda del lenguaje C) es la variedad de operaciones que permite hacer con punteros. Anteriormente hemos visto cómo es posible sumar un valor entero a un puntero para acceder a elementos de un arreglo. De la misma forma, es posible restar un valor entero a un puntero. Si un puntero `p` apunta a una componente de un arreglo `a`, el puntero `p-i` apunta a una componente que está `i` componentes más atrás.

Por ejemplo, declaremos un arreglo de estructuras y dos punteros a estructuras como a continuación:

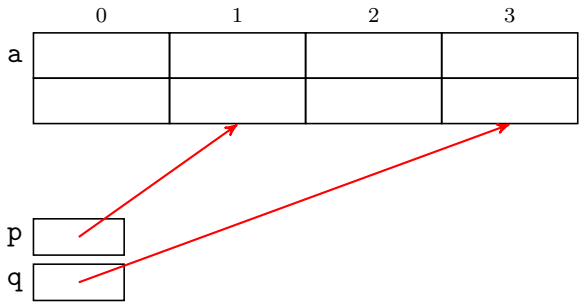
```
struct punto {
    int x;
    int y;
} a[4];           // declara un arreglo de 4 puntos

punto *p, *q;
```

Esos punteros pueden apuntar a componentes del arreglo `a`, ya que estos últimos son puntos:

```
p = &a[1];
q = &a[3];
```

lo cual se ilustra a continuación:



Se puede acceder a las componentes de la estructura apuntada por `p` usando el operador `->`, como por ejemplo:

```
p->x = 2;
```

que, en este caso, es equivalente a `a[1].x = 2`. Además podemos, por ejemplo, acceder a la primera componente del arreglo restando 3 al puntero `q`:

```
*(q-3).y = 8;
```

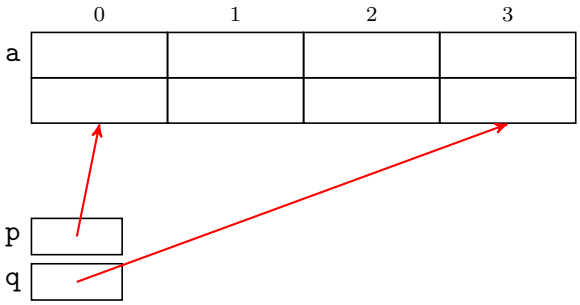
Los punteros que apuntan a componentes de un mismo arreglo pueden ser comparados. Un puntero será menor que otro si apunta a una componente anterior, serán iguales si apuntan a la misma componente, y mayor si apunta a una posterior. Por ejemplo, si tenemos en cuenta el código anterior, las dos primeras de las siguientes condiciones son verdaderas, y la última es falsa:

```
p < q
p != q
p >= q
```

Los punteros también pueden ser modificados para que apunten a otras componentes en base a la componente a la que apunta actualmente. Por ejemplo, si hacemos:

```
p = p - 1;
```

el resultado es que `p` ahora apunta a la componente previa del arreglo, tal como se muestra a continuación:



La resta de punteros también es una operación válida, y produce un entero que representa el número de componentes entre las apuntadas por los dos punteros. Por ejemplo, luego de la última línea de código se tiene que `q - p` produce 3, mientras que `p-q` produce -3.

Finalmente, para inicializar todos los campos del arreglo de estructuras `a` a 0, podemos usar un puntero `p` como a continuación:

```
for (p = a; p <= &a[3]; p++) {
    p->x = 0;
    p->y = 0;
}
```

Aquí el puntero `p` es inicializado en la primera componente del arreglo, y es avanzado en cada iteración a la siguiente con `p++`.

9.4 Punteros a `void`

Un puntero a `void` es llamado un *puntero genérico*, y puede apuntar a objetos de cualquier tipo. Anteriormente, siempre hemos declarado punteros que pueden apuntar a objetos de un tipo fijo predefinido. Consideremos el siguiente ejemplo:

```
int i;
float f[5];

void *p, *q;
p = (void *)&i;
q = (void *)&f[3];
```

Aquí se definen dos punteros genéricos, `p` y `q`. La primera asignación hace que `p` apunte a `i`, mientras que la segunda asignación hace que `q` apunte a la componente `f[3]`. Note que se debe usar el operador de cast para efectivamente convertir los tipos antes de la asignación. De esa forma, ambos lados de la asignación son del mismo tipo: `void *`.

Como ya dijimos, los punteros genéricos pueden apuntar a objetos de cualquier tipo. Sin embargo, no soportan algunas operaciones. Por ejemplo, no es posible sumarle un valor entero a un puntero genérico. La razón es que el compilador no conoce el tamaño del objeto de dato apuntado por el puntero, por lo cual no puede determinar cuántos bytes debe sumar al puntero para que apunte a la componente correspondiente dentro del arreglo. Por ejemplo, no se puede hacer `q++`, siendo `q` definido como antes. Sin embargo, con el cast apropiado es posible hacer ese incremento:

```
(float *)q++;
```

Note cómo el cast transforma a `q` en un puntero a float, el que luego es incrementado.

De forma similar, para imprimir el valor apuntado por un puntero genérico se debe usar un cast, como por ejemplo:

```
cout << *(int *)p << endl;
```

La parte `(int *)p` convierte el puntero genérico `p` en un puntero a `int`. Luego, ese puntero es desreferenciado por el operador `*` que está al comienzo de la expresión, obteniendo el entero correspondiente.

10 Administración Dinámica de Memoria

Aunque no lo hemos mencionado, hasta el momento hemos trabajado con variables estáticas y locales. Eso significa que son variables declaradas por el programador, y cuyo espacio de almacenamiento es reservado por el compilador usando la información brindada por la declaración de la variable. Además, el compilador genera código para destruir dichas variables cuando se alcanza el final del bloque que contiene su declaración.

Sin embargo, en muchos casos no es posible saber de antemano cuánta memoria necesitará un programa en tiempo de ejecución, dado que dependerá del escenario en que esté ejecutando. Por ejemplo, un programa podría necesitar leer una cantidad arbitraria de datos desde la entrada estándar, para almacenarlos en memoria principal. Aunque uno podría pensar en usar un arreglo para almacenar dichos valores, distintas ejecuciones podrían necesitar almacenar una cantidad distinta de ellos. El hecho de no conocer la cantidad de datos a almacenar hace difícil declarar un arreglo de la forma conocida (en donde el tamaño del arreglo es una constante conocida al escribir el programa).

El lenguaje C++ permite declarar arreglos de tamaño variable, como en el siguiente ejemplo:

```
{  // inicio del bloque
    int n, i;   // variables locales al bloque

    cout << "Indique la cantidad de datos a leer: ":
    cin >> n;

    int a[n];   // declara un arreglo de tamaño variable

    for (i = 0; i < n; i++) {
        cout << "Leyendo dato " << i+1 << "/" << n << ": ";
        cin >> a[i];
    }
    ... // procesa el arreglo de datos leído
} // fin del bloque, las variables n, i, y a son destruidas automáticamente
```

Esto resuelve la situación planteada, aunque parcialmente. El problema es que esto funciona sólo para valores de `n` relativamente pequeños, ya que la memoria disponible para ese tipo de variables suele ser limitada. Por lo tanto, esto deja de ser una solución factible en la mayoría de los casos, y recomendamos no usarla.

La solución a esta situación es la administración dinámica de memoria, que permite solicitar y liberar cantidades arbitrarias de memoria en cualquier punto durante la ejecución de un programa. En C++, la memoria solicitada dinámicamente se provee desde un almacén de memoria conocido como *heap*. Usualmente, este almacén tiene una mayor disponibilidad de memoria que el mencionado en el ejemplo anterior, lo que permite resolver de mejor manera la situación planteada (y las situaciones que necesiten memoria dinámica, en general). C++ implementa la administración de memoria dinámica mediante los operadores `new` y `delete`, que manipulan el heap. El ejemplo anterior se resolvería de la siguiente manera usando dichos operadores:

```
{  // inicio del bloque
    int n, i;   // variables locales al bloque
    int *p;

    cout << "Indique la cantidad de datos a leer: ":
    cin >> n;

    p = new int[n];   // pide memoria para un arreglo de enteros de tamaño n

    for (i = 0; i < n; i++) {
        cout << "Leyendo dato " << i+1 << "/" << n << ": ";
        cin >> p[i];
    }
    ... // procesa el arreglo de datos leído
    delete [] p; // libera el arreglo dinámico, DEBE hacerlo el programador
    ...
} // fin del bloque, las variables n, i y p son destruidas automáticamente
```

Este programa solicita un valor `n` desde la entrada estándar, para luego asignar un arreglo de tamaño `n` usando `new`. Luego, solicita `n` valores, los cuales son almacenados en el arreglo. Antes de finalizar el bloque, el arreglo pedido dinámicamente es destruido usando `delete`. Estudiamos a continuación estos operadores.

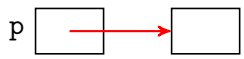
10.1 Operador `new`

Este operador permite solicitar memoria dinámica desde el heap. Las solicitudes de memoria van consumiendo la memoria disponible en el heap. Si hay suficiente memoria para satisfacer el pedido, el operador `new` inicializa la memoria (si así se lo requiere) y retorna un puntero a la memoria asignada. Para poder usar la memoria solicitada, se la debe apuntar con un puntero. Por ejemplo, a continuación se pide memoria para almacenar un entero:

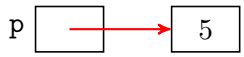
```
int *p;

p = new int;   // pide memoria dinámica para almacenar un entero
```

Aquí, el puntero retornado por `new` es asignado a `p`, por lo que ahora éste apunta a la memoria dinámica recién asignada. Gráficamente, el resultado es el siguiente:



Luego, es posible manipular la memoria asignada usando el puntero `p`, tal como hemos visto anteriormente. Por ejemplo, se puede almacenar un valor en la memoria dinámica recién asignada haciendo, por ejemplo, `*p = 5`, y produciendo como resultado:



Note que la memoria asignada dinámicamente es, en realidad, una variable anónima: es un espacio de memoria que permite almacenar valores, pero que no ha sido declarado, y por lo tanto no tiene nombre. Entonces, la única manera de acceder a la memoria dinámica es mediante punteros.

La solicitud de memoria del ejemplo anterior puede hacerse en la misma declaración, como sigue:

```
int *p = new int;   // pide memoria dinámica para almacenar un entero
```

Inicialización de la Memoria Dinámica. La memoria dinámica puede ser inicializada al momento de solicitarla, como mostramos a continuación:

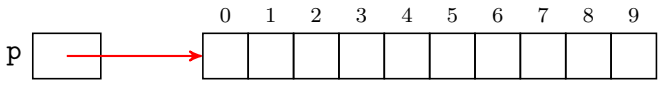
```
int *p = new int(25);
float *q = new float(75.25);
```

Aquí, no sólo se pide memoria dinámica, sino que también se la inicializa con el valor indicado entre paréntesis.

Asignación Dinámica de Arreglos. Uno de los principales usos de la memoria dinámica es solicitar memoria en tiempo de ejecución para almacenar arreglos. De esta manera, es posible asignar memoria para arreglos cuyo tamaño no es necesariamente conocido al escribir el programa (como en el ejemplo estudiado anteriormente). A continuación asignamos memoria dinámica para un arreglo de 10 enteros:

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

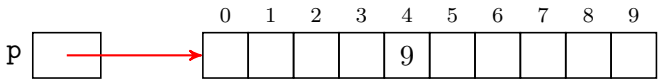
El valor entre corchetes es el tamaño del arreglo que se está solicitando. El operador `new` retorna un puntero a la primera componente del arreglo. Gráficamente, la situación en memoria es la siguiente:



A partir de este punto, el puntero `p` es usado para manipular el arreglo que ha sido asignado dinámicamente, tal como lo estudiamos en secciones anteriores. Por ejemplo, se podría asignar un valor a alguna de las componentes del arreglo de la siguiente manera:

```
p[4] = 9;
```

lo que produce como resultado:



Respecto al tamaño del arreglo solicitado, en el ejemplo anterior hemos usado la constante 10. Sin embargo, en la mayoría de los casos interesantes se usa una variable, tal como en el ejemplo anterior (en donde el tamaño del arreglo dependía del valor `n` introducido por el usuario desde la entrada estándar).

Si la memoria disponible no es suficiente para satisfacer el pedido hecho con `new`, el operador indica este error lanzando una excepción del tipo `std::bad_alloc`.

10.2 Operador `delete`

La memoria dinámica es manejada completamente por quien programa, por lo que es su responsabilidad devolver toda la memoria que ha sido asignada dinámicamente. Para esto, C++ provee el operador `delete`, que se usa de la siguiente manera:

```
int *p = new int;
...
delete p;   // libera la memoria dinámica apuntada por p
```

Para liberar arreglos dinámicos se usa la versión `delete []` del operador:

```
int *p = new int[100];
...
delete [] p;   // libera la memoria dinámica apuntada por el arreglo p
```

10.3 Manejo Responsable de la Memoria Dinámica

El manejo de la memoria dinámica debe hacerse de forma responsable, ya que de otra manera puede llevar a perder demasiado espacio de memoria. Este efecto es conocido como *memory leak* [5]. Esto eventualmente puede llevar a consumir toda la memoria disponible, y provocar que el sistema operativo fuerce la finalización del programa por hacer mal uso de los recursos del computador (en este caso, la memoria). La recomendación es liberar toda la memoria asignada dinámicamente que ya no sea necesaria para el programa. De hecho, cada `new` **debe** tener su correspondiente `delete` en alguna parte del programa.

10.4 Funciones que Retornan Arreglos

Ciertas funciones necesitan retornar un arreglo como resultado. En C++, esto se logra asignando un arreglo de forma dinámica dentro de la función, y retornando un puntero a su dirección base. Por lo tanto, el tipo de retorno de la función debe ser un puntero al tipo base del arreglo que se retorna. Por ejemplo, si necesitamos retornar un arreglo de enteros, el tipo de retorno de la función debe ser `int *`. Otro ejemplo es el siguiente:

```
struct punto {
    float x;
    float y;
};

punto *f(int n) {
    punto *p = new punto[n];
    ...   // lee y procesa n puntos de alguna forma
    return p;
}
```

En este ejemplo, la función `f` recibe un valor entero `n` como parámetro, y retorna un puntero a un arreglo de puntos de tamaño `n`. En este ejemplo, el tamaño del arreglo retornado depende de un parámetro de la función. En otras palabras, la función que invoca a `f` conoce que el arreglo retornado es de tamaño `n`. Un ejemplo de invocación es el siguiente:

```
int i;
punto* q;
...
q = f(100);
// En este punto se sabe que q es un arreglo dinámico de tamaño 100
for (i = 0; i < 100; i++)
    // hace algo con el punto q[i]

delete [] q;   // el arreglo debe liberarse cuando ya no sea necesario mantenerlo
```

Sin embargo, no siempre se conoce el tamaño del arreglo que será retornado por una función. En esos casos, la función debe informar, además, el tamaño del arreglo que retorna. De otra manera, la función que invoca (y que recibe el arreglo como resultado) no sabrá el tamaño del arreglo

[5] O goteo de memoria.

retornado, y por lo tanto no podrá procesarlo. Una manera simple de informar dicho tamaño es agregando un parámetro a la función, el cual es pasado por referencia.

Para ilustrar esto, pensemos en una función `filtro`, que recibe como parámetros un arreglo de `float` y un valor `float c`. La función debe retornar un arreglo con todos los elementos del arreglo original cuyo valor sea menor a `c`. El código de la función es el siguiente:

```
float *filtro(float a[], int n, float c, int& m) {
    int i;
    // primero cuenta la cantidad de elementos del arreglo a retornar
    for (m = i = 0; i < n; i++)
        if (a[i] < c) m++;

    float* p = new float[m];
    int j;

    for(j = i = 0; i < n; i++)
        if (a[i] < c) {
            p[j] = a[i];
            j++;
        }

    return p;
}
```

Note cómo la función declara un parámetro `n` correspondiente al tamaño del arreglo `a`, así como un parámetro por referencia `m` en donde la función informará el tamaño del arreglo que retorna. Luego, la función que invoca a `filtro` podría ser como a continuación:

```
int m;
float a[10] = {3.2, 6.9, 2.1, 7.3, 9.3, 8.5, 3.9, 5.3, 7.5, 9.9};
float *q;
...
q = filtro(a, 10, 5.0, m);
// en este punto sabemos que el arreglo q tiene tamaño m
for (int i = 0; i < m; i++)
    cout << q[i] << endl;
...
delete [] q;
```

Siempre que se necesite retornar un arreglo desde una función, éste deberá ser asignado dinámicamente usando `new`. No es correcto retornar punteros a arreglos declarados localmente en la función que lo retorna. Por ejemplo, el siguiente es un uso incorrecto:

```
int* f() {
    int a[5];
    ....
    return a;
}
```

Otro uso incorrecto es el siguiente:

```
int* f(int n) {
    int a[n];
    ....
    return a;
}
```

El error en ambos ejemplos es que el arreglo `a` existe mientras se esté ejecutando la función `f`. Apenas la función alcance el `return`, esos arreglos se destruyen automáticamente. Sin embargo, la función retorna la dirección base de ese arreglo, el cual no existe más. Esto es, sin dudas, incorrecto.

11 Strings

Los *strings* o *cadenas de caracteres* pueden representarse en C++ usando:

- Un arreglo de caracteres.
- Usando el tipo (o clase) `std::string` de la biblioteca estándar `string`.

11.1 Strings como Arreglos de Caracteres al Estilo C

La representación de strings heredada del lenguaje C usa un arreglo de caracteres para almacenar un string, al cual debe agregarse (al final) el carácter especial '\0', conocido como carácter nulo y cuyo código ASCII es 0. Por ejemplo, el string "`hola`" se puede representar usando:

```
char str[5];

char str[0] = 'h';
char str[1] = 'o';
char str[2] = 'l';
char str[3] = 'a';
char str[4] = '\0';   // indicador de fin de string

cout << str << endl;  // imprime hola
```

La secuencia de asignaciones almacena los caracteres deseados en el string. La última asignación corresponde al carácter de fin de string. Una notación más compacta para inicializar un string es la siguiente:

```
char str[5] = "hola";
```

que tiene exactamente el mismo efecto que el ejemplo anterior (incluyendo la asignación del carácter de fin de string).

La existencia de un carácter nulo para indicar el fin de string se debe a que los strings tienen longitud variable, pudiendo ser ésta menor a la del arreglo base, como por ejemplo:

```
char str[100] = "hola";    // 95 componentes del arreglo quedan sin usar
```

Para strings representados de esta forma, existen muchas funciones definidas en la biblioteca `cstring`, la cual debe incluirse para poder usarlas. Algunas de ellas son las siguientes:

- `unsigned strlen(char *s)`: esta función retorna el número de caracteres en el string `s`, sin contar el carácter nulo. El siguiente ejemplo

  ```
  #include <cstring>
  #include <iostream>
  using namespace std:

  int main() {
      char str[] = "C++ es lindo";
  ```

```
    cout << strlen(s) << endl;
    return 0;
}
```

imprimirá 12.

- `void strcpy(char *s1, char *s2)`: esta función permite copiar todos los caracteres del string `s2` en el arreglo apuntado por `s1`, incluyendo el carácter nulo. Se asume que hay suficiente espacio en `s1` como para contener todos los caracteres de `s2`. Si esto no ocurre, la copia se hará en una zona inválida de la memoria, con resultado indeterminado para el programa. Por ejemplo:

```
char a[100];
char b[] = "C++ es lindo";

strcpy(a, b);
cout << a << endl;
```

lo cual imprimirá "C++ es lindo".

- `void strcat(char *s1, char *s2)`: esta función concatena los caracteres del string `s2` al final del string `s1`. Debe haber suficiente espacio en `s1` para agregar los caracteres de `s2`. En otro caso, el resultado es indeterminado, pudiendo llevar a situaciones de error difíciles de detectar. Por ejemplo:

```
char a[100] = "C++ es ";
char b[] = "lindo";

strcat(a, b);
cout << a << endl;
```

imprime "`C++ es lindo`".

- `int strcmp(char *s1, char*s2)`: esta función permite comparar lexicográficamente los strings `s1` y `s2`. Retorna 0 si ambos strings son iguales, un valor < 0 si `s1` es lexicográficamente menor que `s2`, y un valor > 0 si `s1` es lexicográficamente mayor que `s2`. Por ejemplo:
  - `strcmp("hola", "hola")` retorna 0
  - `strcmp("hola","hello")` retorna un valor positivo
  - `strcmp("hello","hola")` retorna un valor negativo
- `char *strchr(char *s, char c)`: retorna un puntero a la primera aparición del carácter `c` en el string `s`.

11.2 La Clase `string`

Una forma más limpia (y, en general, más conveniente) de manejar strings en C++ la provee el tipo `std::string` definido en la biblioteca `string`. Este tipo oculta todos los detalles de implementación de los strings, como por ejemplo el carácter nulo (que ya no es necesario) y la existencia de espacio suficiente al usar las operaciones, entre otros. Un string puede ser declarado e inicializado de la siguiente manera:

```
#include <string>
using namespace std:

int main() {
    string s = "hola";

    cout << s << endl;
    return 0;
}
```

Además, si `s1` y `s2` son strings, las siguientes funciones y operadores están disponibles (entre muchos otros):

- `s1 = s2`: le asigna al string `s1` el valor del string `s2`.
- `s1.length()`: retorna el largo del string `s1`.
- `s1.empty()`: verifica si `s1` está vacío.
- `s1[i]`: accede al i-ésimo carácter de `s1`.
- `s1 + s2`: concatena los strings `s1` y `s2`.
- `s1 += c`: concatena el carácter `c` al final del string `s1`.
- `s1 == s2`: retorna el valor booleano `true` si los strings son iguales, retorna `false` en otro caso.
- `s1.find(c)`: retorna la posición de la primera ocurrencia en `s1` del carácter `c`.
- `s1.find(s2)`: retorna la posición de la primera ocurrencia del string `s2` en `s1`.
- `s1.c_str()`: retorna un `char *` apuntando a un string del estilo del lenguaje C (es decir, un arreglo de `char` finalizado con el carácter nulo).

12 Archivos

Un archivo es una colección de datos relacionados, usualmente almacenados en memoria secundaria (como, por ejemplo, en disco). C++ considera a un archivo simplemente como una secuencia de bytes. El lenguaje implementa el manejo de archivos a través de variables de tipo stream y ofrece la biblioteca estándar `<fstream>`, que contiene rutinas, funciones y objetos que permiten su manejo. Es necesario incluirla antes de cualquier tarea que involucre el uso de archivos:

```
#include <fstream>
```

Tipos de Stream. A través de un *stream* se pueden leer o escribir los datos del archivo, dependiendo si permite operaciones de entrada o salida, respectivamente. Existen tres tipos de stream:

- `ifstream`: que solo soporta operaciones de entrada (leer datos).
- `ofstream`: que solo soporta operaciones de salida (escribir datos).

- `fstream`: que soporta operaciones tanto de entrada como de salida.

Dado que los stream son variables, es necesario declararlos antes de usarlos (como cualquier otra variable). Por ejemplo, para declarar un `stream` de salida se usa una sentencia como la siguiente:

```
ofstream fp;
```

Luego de esto, la variable `fp` podrá ser el nexo entre nuestro programa y un archivo en disco.

Abriendo un Archivo. Para manipular un archivo (escribiendo o leyendo datos), primero se debe abrir usando la función `open`, usando su nombre externo (es decir, el nombre que se le ha dado al archivo en el sistema operativo subyacente). Luego de abrirlo adecuadamente, podrá ser usado para lectura o escritura, según sea el caso. Al finalizar su uso, el archivo debe cerrarse usando la función `close`. Un ejemplo de uso es el siguiente:

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

int main() {
    fstream fp;
    fp.open("archivo.txt", <modo>);

    if (!fp.is_open()) {
        cerr << "No se pudo abrir el archivo" << endl;
        return 1; // error
    }
    // realizar operacion de entrada o salida según <modo>

    fp.close();  // cierra el archivo cuando ya no se necesita usarlo
    return 0;
}
```

La operación `open` recibe dos parámetros: un string que representa el nombre de archivo a abrir, y el modo de apertura del archivo. Para esto último, existen varios *flags* que pueden ser utilizadas según sea necesario:

- `ios::in`: abre el archivo para lectura de datos. Se puede obviar si la variable se declara usando `ifstream` en lugar de `fstream`.
- `ios::out`: abre el archivo para escritura de datos. Si el archivo ya existe, destruye sus contenidos. En caso contrario, crea uno nuevo. Se puede obviar si se usa `ofstream`.
- `ios::app`: abre el archivo permitiendo operaciones de escritura sólo al final del archivo, permitiendo agregar nuevos datos al final del mismo.
- `ios::ate`: permite comenzar operando al final del archivo.

Estos *flags* pueden mezclarse según sea necesario, permitiendo combinar modos de trabajo. Por ejemplo, para permitir leer y escribir en un archivo, se puden combinar `ios::in` y `ios::out` al abrir el archivo, usando el operador `|` (que es un *or* a nivel de bits) tal como a continuación:

```
fstream fp;
fp.open("archivo.txt", ios::in|ios::out);
```

Esta es la forma recomendada cuando se desea:

- modificar el contenido de un archivo (ya que `ios::out` usado por sí solo destruye el archivo existente al abrirlo),
- realizar operaciones de lectura y escritura sobre el archivo.

Después de la operación `open`, se puede verificar si el archivo fue abierto correctamente con la operación `is_open`, que retorna verdadero si es el caso.

Cerrando un Archivo. Al terminar de usar un archivo se debe cerrar con un llamado a la operación `close`. Esto es necesario si el archivo debe ser usado por otro programa o se desea usar el *stream* para operar con otro archivo, o con el mismo archivo pero en otro modo.

El Cursor de un Stream. Cada stream tiene asociado un *cursor* que puede ser movido a lo largo del archivo. En general, las operaciones sobre el archivo se hacen sobre la posición en la que está el cursor. Al momento de abrir un archivo, la posición en que se ubica el cursor depende del tipo de apertura que se use, tal como se indica a continuación:

- `ios::in`: el cursor del stream es colocado en la posición inicial del archivo.
- `ios::out`: el cursor del stream es colocado en la posición inicial del archivo, ya que en este modo el archivo siempre estará vacío al abrirlo.
- `ios::app`: antes de cada operación de escritura, el cursor es posicionado automáticamente al final del archivo.
- `ios::ate`: coloca el cursor al final del archivo inmediatamente después de abrirlo.

Las operaciones de lectura y escritura mueven el cursor de forma automática, avanzando a la posición que está a continuación de los datos leídos/escritos. Además, existen funciones que permiten ubicar el cursor en alguna posición arbitraria dentro del archivo. Por ejemplo, la operación `seekg` avanza el cursor una cierta cantidad de bytes dentro del archivo. Existen dos versiones de esta operación:

- `seekg(pos)`: recibe un entero `pos` con la posición respecto del inicio del archivo, a la cual mover el cursor. Las posiciones se miden en bytes.
- `seekg(offset, base)`: recibe un entero `offset` con la diferencia relativa respecto de `base`, a la cual desea moverse el cursor. `base` puede ser cualquiera de las siguientes constantes:
  - `ios::beg`: el inicio del archivo,
  - `ios::end`: el fin del archivo,
  - `ios::cur`: la posición actual del cursor.

Por ejemplo, para mover el cursor al inicio del archivo, se hace:

```
fp.seekg(0);
```

Para mover el cursor al quinto byte dentro del archivo, se puede hacer:

```
fp.seekg(5);
```

o, alternativamente:

```
fp.seekg(5, ios::beg);
```

Ambos usos son equivalentes. Por otro lado, para mover el cursor al final del archivo se hace:

```
fp.seekg(0, ios::end);
```

Esto indica que el cursor se mueva 0 bytes desde el fin del archivo. Para retroceder un byte hacia atrás y continuar las operaciones desde ese punto se hace:

```
fp.seekg(-1,cur);
```

Se puede obtener la posición actual del cursor, respecto al inicio del archivo, con la operación `tellg`, invocándola de la siguiente manera:

```
fp.tellg();
```

Al alcanzar el final del archivo, futuras operaciones de entrada no se realizarán o retornarán valores especiales. La operación `eof` retornará verdadero cuando se haya alcanzado el final del archivo.

12.1 Formatos de Archivos

La manera de realizar las operaciones de entrada y salida sobre un archivo dependerán del formato del mismo. En C++ existen dos formatos de archivos:

- Archivos ASCII.
- Archivos binarios.

Por defecto, la operación `open` asume que los archivos están en formato ASCII.

12.1.1 Archivos ASCII

Un archivo en este formato tendrá su contenido en codificación ASCII (es decir, caracteres textuales) y, por tanto, corresponde a aquellos que son en general legibles por un humano.

Lectura y Escritura con Formato. Para escribir en un archivo ASCII se ocupa el operador `<<`, de forma similar al stream `cout` usado para imprimir en la pantalla. En el siguiente ejemplo se escribe `Hola mundo` y el número entero 2021 en el archivo de texto `archivo.txt`:

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

int main() {
    ofstream fp;
    fp.open("archivo.txt");
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // error
    }
    // escribe "Hola mundo" y un salto de línea
```

```
    fp << "Hola mundo" << endl;
    // escribe el entero 2021
    fp << 2021;

    fp.close();
    return 0;
}
```

Luego de compilar y ejecutar este programa, el contenido del archivo de texto `archivo.txt` es el siguiente:

```
Hola mundo
2021
```

Para leer desde un archivo ASCII se ocupa el operador `>>`, similar a `cin`. El siguiente ejemplo lee los contenidos del archivo `archivo.txt`, escritos por el programa anterior:

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

int main() {
    ifstream fp;
    string s1, s2;
    int a;
    fp.open("archivo.txt");
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // error
    }
    // lee "Hola" desde el archivo y lo almacena en s1
    fp >> s1;
    // lee "mundo" desde el archivo y lo almacena en s2
    fp >> s2;
    // lee el entero 2021 desde el archivo y lo almacena en a
    fp >> a;

    fp.close();
    return 0;
}
```

La lectura desde un archivo ASCII usando el operador `>>` procede hasta encontrar el primer espacio en blanco (es decir, un espacio, tabulación, o salto de línea). El cursor queda en la posición posterior al espacio en blanco, y el valor leído es almacenado en la variable indicada en la operación de lectura, con el formato correspondiente a dicha variable. En el anterior ejemplo, al hacer `fp >> s1`, el string leído desde `fp` es almacenado con formato de string en la variable `s1`. Por otro lado, cuando se hace `fp >> a`, el string "2021" que se lee desde el archivo es almacenado como un entero en la variable `a`. Por esta razón es que se llama lectura y escritura con formato a este tipo de operación.

Si se intenta leer con este operador al final del archivo, no habrá cambios en la variable y se retornará un valor evaluable como falso.

Lectura sin formato. También es posible leer datos desde un archivo ASCII simplemente como strings o caracteres, sin darles ningún tipo de formato. La función `getline` definida en `<iostream>` permite leer líneas completas desde un archivo ASCII, y almacenarlas como strings. Consideremos el siguiente ejemplo:

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

int main() {
    ifstream fp;
    string s;
    fp.open("archivo.txt");
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // finaliza la ejecución del programa retornando error
    }
    //lee una línea desde el archivo
    getline(fp, s);
    fp.close();
    return 0;
}
```

En este caso, el llamado `getline(fp, s)` provoca que se lea hasta alcanzar el primer salto de línea del archivo, almacenando en el string `s` la frase "`Hola mundo`". Notar que el cursor quedará antes del primer `2` de `2021`. Intentar leer con esta función en el final del archivo tendrá un efecto similar al observable con el operador `>>`.

Otra manera de leer archivos ASCII es carácter a carácter. Para ello se puede usar la operación `get`, que lee el siguiente carácter desde el archivo (de acuerdo a la posición del cursor) y lo retorna, sin importar si es un espacio en blanco o no. Al intentar leer el final de archivo, retornará un carácter especial de fin de archivo, denominado `EOF` (por "End Of File"). El siguiente programa muestra todos los caracteres del archivo junto a su posición:

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

int main() {
    ifstream fp;
    fp.open("archivo.txt");
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; //
    }

    char c;
    int i = 0;
    while((c = fp.get()) != EOF) {
        cout << "(" << i++ << ":" << c << ") ";
    }
    cout << endl;

    fp.close();
}
```

El llamado a la operación `get` se encuentra dentro de la condición del `while`, y mientras el valor asignado a `c` sea distinto del valor especial `EOF` se mostrarán por pantalla los caracteres junto con sus posiciones. Los paréntesis rodeando la asignación `c = in.get()` son obligatorios, ya que la operación de asignación (`=`) tiene menor precedencia que la de desigualdad (`!=`). Al compilar y ejecutar este programa se mostrará por pantalla lo siguiente:

```
(0:H), (1:o), (2:l), (3:a), (4: ), (5:m), (6:u), (7:n), (8:d), (9:o), (10:
), (11:2), (12:0), (13:2), (14:1),
```

Notar cómo este programa muestra por pantalla que en la posición 4 se encuentra un carácter de espacio, mientras que en la posición 10 se encuentra un carácter de salto de línea.

12.1.2 Archivos Binarios

Un archivo binario es uno cuyo contenido estará en el mismo formato binario que tienen los objetos en la memoria. Por ejemplo, el valor entero 2021 que en un archivo ASCII era escrito como el string "2021", en un archivo binario sería representado por su codificación binaria de 4 bytes (asumiendo enteros de 4 bytes). Usualmente, este tipo de archivos no es legible a simple vista.

Abrir un Archivo Binario. Para abrir un archivo de formato binario, se debe usar el flag `ios::binary`, combinado con el flag correspondiente a la lectura o escritura, según sea neceario. Por ejemplo, para abrir un archivo para escritura en formato binario se hace:

```
fstream fp;
fp.open("archivo.dat", ios::out|ios::binary);
```

Esto abre un archivo binario para escritura. Lo mismo puede hacerse con `ios::in` para lectura. Incluso, se puede hacer:

```
fp.open("archivo.dat", ios::in|ios::out|ios::binary);
```

para abrir un archivo binario para lectura y escritura.

Escribir un Archivo Binario. Para escribir en un archivo binario se debe utilizar la operación `write`, que recibe dos parámetros: (1) un puntero (o *dirección de memoria*) de tipo `char *` al área de memoria que contiene los datos que se desean escribir, y (2) el tamaño del bloque de memoria que será escrito. En el siguiente ejemplo, se almacena en el archivo `archivo.dat` un único entero en formato binario:

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

int main() {
    ofstream fp;
    int i = 5;
    fp.open("archivo.dat", ios::binary);
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // error
    }

    fp.write((char*)&i, sizeof(int));   // escribe el valor de i en el archivo

    fp.close();
    return 0;
}
```

Los dos parámetros de `write` son los siguientes:

- `(char*)&i`: es un puntero a la variable `i`, lo que le indica a `write` que debe escribir al archivo el valor de la variable `i`. La conversión a `char*` es un requerimiento de la implementación de la función.
- `sizeof(int)`: que indica la cantidad de bytes que deben escribirse en el archivo. En este caso, se quiere escribir un único entero, por lo tanto `sizeof(int)` es la cantidad de bytes a escribir.

Leer un Archivo Binario. Para leer datos desde un archivo binario, se usa la operación `read`, que usa parámetros similares a `write`. Por ejemplo, para leer el contenido del archivo `archivo.dat` generado anteriormente, hacemos lo siguiente:

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

int main() {
    ifstream fp;
    int i;
    fp.open("archivo.dat", ios::binary);
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // error
    }

    fp.read((char*)&i, sizeof(int));

    cout << i << endl;

    fp.close();
    return 0;
}
```

En este caso, los parámetros de `read` son: (1) un puntero a la zona de memoria en donde se almacenará el valor leído, y (2) la cantidad de bytes que usa el dato leído.

Almacenar Arreglos en Archivos. El hecho de estar almacenados en una zona contigua de memoria hace que los arreglos puedan ser almacenados de forma simple en un archivo binario, lo cual es muy útil en general para hacer volcado de datos directamente desde la memoria a un archivo. El siguiente ejemplo muestra cómo escribir un arreglo de 10 enteros en un archivo:

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

int main() {
    ofstream fp;
    int array[10] = {0,1,2,3,4,5,6,7,8,9};

    fp.open("archivo.dat", ios::binary);
    if (!fp.is_open()) {
        cerr << "Error el abrir el archivo" << endl;
        return 1; // error
    }
```

```
    fp.write((char*)array, 10*sizeof(int));

    fp.close();
    return 0;
}
```

Respecto a los parámetros de `write`, note lo siguiente:

- dado que el nombre del arreglo es su propia dirección base, el primer parámetro no lleva `&` como en el caso de otros tipos de variables.
- en el segundo parámetro se multiplica al tamaño de cada una de las componentes del arreglo (en este caso, (`sizeof(int)`) por la cantidad de componentes, que en este caso son 10. Ése es el tamaño total del arreglo, en bytes.

Para mover el cursor dentro de un archivo binario, se pueden utilizar las operaciones `seekg` y `tellg` vistas anteriormente. Sólo hay que tomar en consideración que el movimiento será por una cierta cantidad de bytes. Por ejemplo, si se quiere avanzar hasta la componente 2 del arreglo anteriormente almacenado en el archivo, hay que hacer `fp.seekg(2*sizeof(int))`. En general, se puede reemplazar el 2 por la posición a la que se necesita acceder, y `sizeof(int)` por el tipo base del arreglo correspondiente.

Almacenar Structs en Archivos. Los archivos binarios también permiten almacenar structs, tal como muestra el siguiente ejemplo, que almacena un arreglo de structs en un archivo binario:

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

struct punto {
    int x;
    int y;
};

int main() {
    punto pts[4] = {{2,3},{1,8},{5,6},{6,4}};
    ofstream out;
    out.open("arch.dat",ios::binary);
    if(!out.is_open()) {
        cerr << "Error: no se pudo abrir el archivo arch.dat" << endl;
        exit(1);
    }

    out.write((char*)pts, 4*sizeof(punto));

    out.close();
    return 0;
}
```

Como se puede ver, el programa almacena un arreglo de 4 puntos en el archivo. Notar que se debe realizar un *cast* del arreglo `pts` en el llamado a `write`. Como siempre, el segundo parámetro indica el tamaño total del arreglo en bytes, en este caso 4 elementos de tipo `punto`.

El siguiente programa lee (uno a uno) los structs del archivo generado con el programa anterior, y muestra su contenido por pantalla:

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

struct punto {
    int x;
    int y;
};

int main() {
    punto pt;
    ifstream in;
    in.open("arch.dat",ios::binary);
    if(!in.is_open()) {
        cerr << "Error: no se pudo abrir el archivo arch.dat" << endl;
        exit(1);
    }

    while(in.read((char*)&pt, sizeof(punto)))
        cout << "x = " << pt.x << ", y = " << pt.y << endl;

    in.close();
    return 0;
}
```

Notar que aunque todos los elementos de tipo `struct punto` fueron almacenados juntos, se pueden leer uno por uno. La operación `read` retorna un valor booleano, que es `false` cuando la lectura alcanza el fin del archivo. Dicho valor es usado para iterar mientras haya puntos en el archivo.

Este último programa demuestra como escribir al final del archivo generado anteriormente, y luego leer todas las componentes en un mismo programa:

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

struct punto {
    int x;
    int y;
};

int main() {
    punto pt = {4, 5};
    fstream f;
    f.open("arch.dat");
    if(!f.is_open()) {
        cerr << "Error: no se pudo abrir el archivo arch.dat" << endl;
        exit(1);
    }
    if(!f.seekg(0, ios::end)) {
        cerr << "Error: no se pudo ir al final del archivo" << endl;
        exit(1);
    }
    if(!f.write((char*)&pt, sizeof(punto))) {
```

```
        cerr << "Error: no se pudo escribir el nuevo punto" << endl;
        exit(1);
    }
    if(!f.seekg(0, ios::beg)) {
        cerr << "Error: no se pudo ir al inicio del archivo" << endl;
        exit(1);
    }
    while(f.read((char*)&pt, sizeof(punto)))
        cout << "x = " << pt.x << ", y = " << pt.y << endl;

    f.close();
    return 0;
}
```

12.2 Ejemplos Adicionales

Ejemplo 1: Lectura y Escritura con Formato. El siguiente programa lee enteros desde el archivo `num.txt`, y escribe sus valores absolutos en el archivo `abs.txt`:

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

int main() {
    ifstream in;
    ofstream out;
    in.open("num.txt");
    if(!in.is_open()) {
        cerr << "Error: no se pudo abrir el archivo num.txt" << endl;
        exit(1);
    }

    out.open("abs.txt");
    if(!out.is_open()) {
        cerr << "Error: no se pudo abrir el archivo abs.txt" << endl;
        exit(1);
    }

    int num;
    while(in >> num)
        out << abs(num) << " ";

    in.close();
    out.close();
    return 0;
}
```

Notar que la condición del `while` es la operación `in >> num`, de lectura desde el stream `in` al entero `num`. Dicha operación retorna un valor booleano que es `false` cuando se alcanza en fin de archivo.

Ejemplo 2: Lectura y Escritura sin formato. El siguiente programa permite copiar un archivo en otro. Asume que los nombres de los archivos han sido pasados por entrada estándar:

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main() {
    ifstream in;
    ofstream out;
    string namein, nameout;

    cout << "Ingrese el nombre de archivo de entrada:";
    cin >> namein;
    in.open(namein);
    if(!in.is_open()) {
        cerr << endl << "Error: no se pudo abrir el archivo de entrada " << namein
             << endl;
        exit(1);
    }

    cout << endl << "Ingrese el nombre de archivo de salida:";
    cin >> nameout;
    out.open(nameout);
    if(!out.is_open()) {
        cerr << endl << "Error: no se pudo abrir el archivo de salida " << nameout
             << endl;
        exit(1);
    }

    char c;
    while((c = in.get()) != EOF)
        out.put(c);

    in.close();
    out.close();
    return 0;
}
```

13 Tipos de Datos Abstractos

Los *tipos de datos abstractos* (TDAs) son un concepto fundamental para el desarrollo de software a mediana y gran escala. La idea es manipular conceptos prescindiendo de los detalles del mismo, centrándonos simplemente en la funcionalidad que proveen. Para entender qué es un TDA, pensemos en el siguiente ejemplo doméstico: aprender a conducir un auto. Durante el aprendizaje, se le presentará el tablero del auto, que incluye todo lo necesario para su conducción: la llave de encendido, un volante, pedales de freno, acelerador y embrague, palanca de cambios, y palancas de luces, entre otros. Todo eso le permite interactuar con el auto sin preocuparse de cómo se implementan sus funcionalidades. El tablero del auto es, en definitiva, la *interfaz* entre el conductor y el auto. Por ejemplo, usted aprenderá que el auto viaja a mayor velocidad a medida que presiona más el acelerador, pero no se preocupa de cómo está implementado ese mecanismo. De esa forma, usted puede cambiar de modelo de auto y aún sabrá cómo acelerar: presionando el pedal. Lo mismo ocurre con el volante, y las demás funcionalidades.

Imagine si tuviese que aprender a acelerar un auto directamente, manipulando los mecanismos. Eso no sólo complica la conducción, sino que además tendría que aprender a acelerar cada vez que cambie el auto por uno de otra marca, o incluso si cambia de un auto de gasolina a uno

eléctrico: seguramente los mecanismos son totalmente distintos a los que conocía. Extendiendo el razonamiento al resto de las funcionalidades, usted tendría que aprender todos los mecanismos cada vez que cambia de auto, lo cual no es práctico. De alguna manera, los detalles de implementación del auto están ocultos detrás de la interfaz con la que interactuamos: el tablero. Parece quedar claro que esa es una mejor manera de interactuar con un concepto como es el auto.

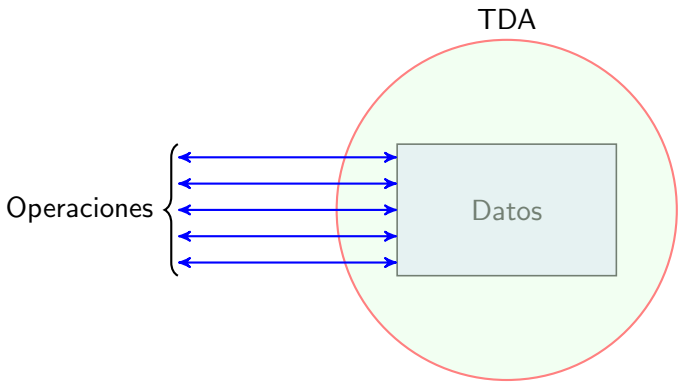
Pensando de forma similar sobre el desarrollo de software, en donde usualmente los sistemas están construidos en base a TDAs, se podría construir software en base a componentes que proveen funcionalidades que no necesariamente sabemos cómo están implementadas. Eso es mucho más práctico en general, y permite concentrarse en los detalles del software que se está construyendo, en lugar de los detalles de los componentes que van a conformarlo.

13.1 Definición Formal

Formalmente, un TDA consiste de:

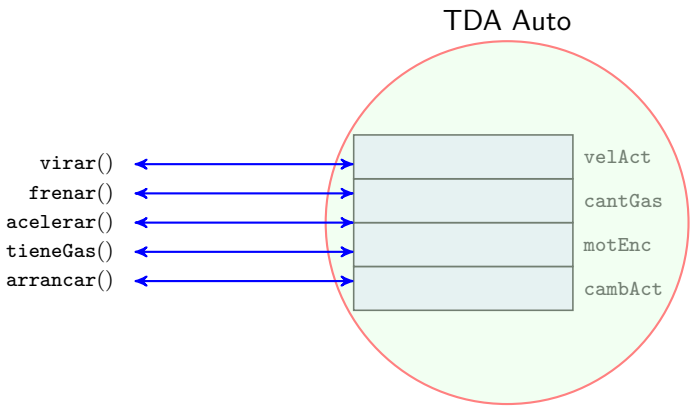
1. Un tipo de datos, compuesto por:
   - Un conjunto de datos, a los que llamaremos *los datos del TDA*,
   - Un conjunto de operaciones sobre esos datos, llamada *la interfaz del TDA*.
2. Una cápsula que encierra a los datos del TDA, de manera que la única manera de acceder a ellos es mediante las operaciones de la interfaz.

Los datos del TDA permiten implementar *su estado actual*, mientras que las operaciones implementan *su comportamiento*. Gráficamente, un TDA puede ser representado como a continuación:



Lo importante es que el TDA encapsula sus datos mediante la cápsula conceptual de color rojo de la figura anterior: las operaciones son la única manera de acceder al estado (datos) del TDA.

Por ejemplo, un auto pensado como un TDA sería algo similar a lo siguiente:



Aquí, los datos del TDA consisten de 4 variables, necesarias para modelar su estado: una variable para almacenar la velocidad actual, otra para la cantidad de gasolina que tiene actualmente el auto, otra para indicar si el auto está encendido o no, y una para almacenar el cambio actual. Respecto a las operaciones, se permite virar, frenar, acelerar, chequear si el auto tiene gasolina, y arrancar el auto. De esta manera, la única manera de modificar el estado actual de un auto es a través de las operaciones de la interfaz.

13.2 Implementación de TDAs en C++

Un TDA en C++ puede implementarse usando el concepto de *clases*, denotados con la palabra clave `class` en el lenguaje. Por ejemplo, el TDA Auto del ejemplo anterior podría definirse de la siguiente manera:

```
class Auto {
  private:
    // datos del TDA, encapsulados por la clausula private
    int velAct;
    float cantGas;
    bool motEnc;
    int cambAct;
  public:
    // operaciones del TDA, visibles gracias a la clausula public
    Auto() {
        motEnc=false; velAct=0; cantGas=0.0; cambAct=0;
    };
    ~Auto() {;};
    void arrancar();
    void acelerar();
    void frenar();
    bool tieneGas();
    void cargarGas(float cant);
    void subirCambio();
    void bajarCambio();
    void neutro();
};
```

Una clase está compuesta de:

1. Variables miembro (que son similares a los campos de un `struct`), las cuales implementan los datos del TDA, y

2. funciones miembro, o métodos, los cuales implementan las operaciones del TDA.

Para encapsular los datos del TDA, y que no puedan ser accedidos por ningún programa desde afuera de la clase, se usa la clausula `private`. Por otro lado, para que las funciones miembro puedan ser usadas desde afuera del TDA, se usa la clausula `public`. Las funciones miembro de la clase pueden acceder a las variables miembros de la misma, siendo éstas las únicas que pueden modificar los datos del TDA, tal como se espera. El siguiente código declara una variable de la clase `Auto`, e intenta acceder (erróneamente) a una de sus variables miembro:

```
int main() {
    Auto a;   // declara una variable de la clase Auto

    a.velAct = 100;   // error
    return 0;
}
```

Esto produce un error de compilación, ya que se está intentando acceder a uno de los datos del TDA `Auto` directamente, sin usar una de las funciones del TDA. Un posible uso correcto del TDA `Auto` lo muestra el siguiente programa:

```
int main() {
    Auto a;
    a.arrancar();
    a.cargarGas(50.0);
    while (a.tieneGas()) {
        a.acelerar();
        /* ... */
    }
    return 0;
}
```

Aquí, `a.arrancar()` invoca al método `arrancar()` de la clase `Auto`, para el auto `a`. El resultado de dicha operación es que el auto `a` pasará a estar encendido.

13.3 Constructores y Destructores

En la definición de las clase `Auto` dada anteriormente, hay dos funciones especiales, que no declaran tipo de retorno y se llaman de la misma forma que la clase. En este caso, nos referimos a las funciones `Auto()` y `~Auto()`, que reciben el nombre de *constructor* y *destructor* de la clase, respectivamente, y que estudiamos en detalle a continuación.

Constructores. Los constructores de una clase permiten inicializar variables de una clase de manera automática, al momento de declarar la clase. En otras palabras, la declaración de una variable invoca automáticamente a un constructor de la clase. De esta forma, la variable es creada e inicializada. El constructor debe definir las acciones específicas de inicialización a realizar sobre los datos de la clase. En la clase `Auto`, el constructor `Auto()` asigna valores iniciales a las 4 variables miembro de la clase. Entonces, al declarar un `Auto`, sus variables miembro son inicializadas de la manera indicada en el constructor. En el siguiente programa, luego de la declaración del auto `a`, su estado es el indicado en el constructor:

```
int main() {
    Auto a;
    // Aquí, a.motEnc==false, a.velAct==0, a.cantGas==0.0, y a.cambAct==0.
```

```
    return 0;
}
```

Aunque en este caso el constructor de la clase `Auto` ha sido definido sin parámetros, en general podemos necesitar agregarlos. Además, es posible definir varios constructores para una misma clase, siempre y cuando se distingan entre sí respecto al número y tipo de sus parámetros. Por ejemplo, podría ser necesario tener un constructor de la clase `Auto` que inicialice un auto con una cierta cantidad de gasolina. Entonces, deberíamos agregar un segundo constructor a la clase, que sea de la forma:

```
Auto(float cG) {
    motEnc=false; velAct=0; cantGas=cG; cambAct=0;
}
```

Luego, para usar dicho constructor debe declararse una variable de la siguiente forma:

```
int main() {
    Auto a(25.0);
    // en este punto, el auto a ha sido construido con 25.0 litros de gasolina
    return 0;
}
```

Así, el constructor que es invocado es aquel que corresponda a los parámetros usados en la declaración de la variable. Si no se usan parámetros en la declaración (es decir, se declara `Auto a;`), el constructor invocado es el que no tiene parámetros (en este caso, `Auto()`).

Destructores. El destructor de una clase es la contraparte de los constructores, y se invocan automáticamente al alcanzar el final del bloque que declara a una variable de la clase. En C++, un destructor tiene el mismo nombre que la clase, precedido por $\sim$. Por ejemplo, el destructor de la clase `Auto` se llama $\sim$`Auto()`.

En general, un destructor se encarga de liberar la memoria dinámica pedida en la clase (si fuese el caso), y cerrar los archivos abiertos dentro de la clase, entre otras tareas. Hay que tener en cuenta que las variables miembro de la clase se destruyen automáticamente al llegar al final del bloque que contiene la declaración de la variable, por lo tanto no necesitan ser destruidos por el destructor. En el ejemplo de la clase `Auto`, ninguna de sus variables miembro es de tipo puntero o archivo, por lo que su destructor $\sim$`Auto()` es vacío. A diferencia de los constructores, cada clase puede tener un único destructor, el cual nunca tiene parámetros.

13.4 Implementación de las Funciones de una Clase

Hasta ahora hemos estudiado cómo se define una clase que representa a un TDA, junto con sus constructores y destructor. Mostramos a continuación cómo implementar las funciones miembro de la clase. En general, hay que tener en cuenta que las funciones miembro son, en definitiva, funciones, y por lo tanto se implementan de la misma forma estudiada en secciones anteriores de este documento. Planteamos a continuación dos maneras alternativas de agregar el código correspondiente a las funciones de una clase.

Implementación de las Funciones dentro de la Clase. Una manera de implementar las funciones de una clase es dentro la misma clase. Para el ejemplo de la clase `Auto` estudiada anteriormente, eso correspondería a lo siguiente:

```
class Auto {
  private:
    // datos del TDA, encapsulados por la clausula private
    int velAct;
    float cantGas;
    bool motEnc;
    int cambAct;
  public:
    // operaciones del TDA, visibles gracias a la clausula public
    Auto() {motEnc=false; velAct=0; cantGas=0.0; cambAct=0;};
    Auto(float cG) {motEnc=false; velAct=0; cantGas=cG; cambAct=0;};

    ~Auto() {;};

    void arrancar() {motEnc = true; velAct = 0;};

    void Auto::acelerar() {velAct++;};

    void frenar() {velAct--;};

    bool tieneGas() {return cantGas > 0.0;};

    void cargarGas(float cant) {cantGas += cant;};

    void subirCambio() {cambAct++;};

    void bajarCambio() {cambAct--;};

    void neutro() {cambAct=0;};
};
```

Ésta suele ser la manera recomendada cuando las funciones miembro son simples, como en este ejemplo. Cuando las funciones son mas complejas de implementar, sin embargo, esta manera complica la lectura de la clase, y puede complicar el entendimiento de las operaciones que provee el TDA. En esos casos, se aconseja usar la siguiente forma de implementar las funciones.

Implementación de las Funciones fuera de la Clase. En esta alternativa separamos la definición de la clase de la implementación de sus funciones. Usualmente, la definición de la clase se hace en un archivo de extensión `.hpp`, mientras que las funciones se implementan en un archivo de extensión `.cpp`. Dicho archivo debe incluir al `.hpp` que define la clase, usando `#include`. Por ejemplo, para la clase `Auto` podríamos tener un archivo `auto.hpp` que contenga la definición de la clase, como a continuación:

```
// Archivo auto.hpp
class Auto {
  private:
    // datos del TDA, encapsulados por la clausula private
    int velAct;
    float cantGas;
    bool motEnc;
    int cambAct;
  public:
    // operaciones del TDA, visibles gracias a la clausula public
    Auto();
```

```
    Auto(float cG);

    ~Auto();

    void arrancar();

    void Auto::acelerar();

    void frenar();

    bool tieneGas();

    void cargarGas(float cant);

    void subirCambio();

    void bajarCambio();

    void neutro();
};
```

Por otro lado, el archivo `auto.cpp` contendrá la implementación de las funciones que define la clase, como a continuación:

```
// Archivo auto.cpp
#include "auto.hpp"

Auto::Auto(){
    motEnc=false;
    velAct=0;
    cantGas=0.0;
    cambAct=0;
}

Auto::Auto(float cG){
    motEnc=false;
    velAct=0;
    cantGas=cG;
    cambAct=0;
}

Auto::~Auto(){}

void Auto::arrancar() {
    motEnc = true;
    velAct = 0;
}

void Auto::acelerar() {
    velAct++;
}

void Auto::frenar() {
    velAct--;
}

bool Auto::tieneGas() {
```

```
    return cantGas > 0.0;
}

void Auto::cargarGas(float cant) {
    cantGas += cant;
}

void Auto::subirCambio() {
    cambAct++;
}

void Auto::bajarCambio() {
    cambAct--;
}

void Auto::neutro() {
    cambAct=0;
}
```

Note que es necesario agregar "`Auto::`" precediendo el nombre de cada función, indicando a qué clase corresponde cada una de ellas. Esta alternativa permite, en general, un desarrollo de software más ordenado.

13.5 Clases y Administración de Memoria Dinámica

El constructor de una clase no sólo es invocado automáticamente al momento de declarar una variable de esa clase, sino que también al momento de pedir memoria para una variable de la clase. Por ejemplo, en el siguiente programa se pide memoria dinámica para un puntero de la clase `Auto`:

```
int main() {
    Auto *a = new Auto(25.0);  // crea un Auto de forma dinámica, y lo inicializa
                               // con 25.0 de gasolina

    /* ... se usa el puntero a ... */

    delete a;                  // se llama automáticamente al destructor
    return 0;
}
```

Al inicializar el puntero con el operador `new` se puede invocar el constructor de la clase. Luego, al usar `delete`, se invoca automáticamente al destructor de la clase.