Resolviendo Problemas

Con el Lenguaje de Programación C++

M.Sc. Jorge Terán P.

Auspica:

Prologo

El presente de texto de programación tiene su origen el texto *Conceptos y ejercicios de Programación* del M.Sc. Jorge Terán Pomier. Este texto esta orientado a estudiantes de primeros años de universidad y últimos años de colegio. Contiene muchos ejercicios que no han sido incluidos en el presente texto. Aún cuando está orientado al lenguaje de programación Java, es complemento ideal para iniciarse en la solución de problemas utilizando un lenguaje de programación para computadoras.

El presente texto ha sido escrito pensando el lenguaje C++. No pretende ser un manual de referencia del lenguaje, de los cuales hay muchos disponibles. El presente texto esta escrito para presentar los conceptos necesarios para que pueda resolver sus primeros problemas. Luego con ayuda de texto de referencia del lenguaje C++ podrá buscar y entender otras funcionalidades disponibles en el lenguaje.

Para un uso efectivo de este material, se recomienda, primero instalar el compilador C++ y el entorno de desarrollo, descritos en el capitulo 1. Segundo, se recomienda que se escriban los programas que se presentan en los diferentes capítulos estudiando su funcionamiento. Tercero, buscar ejercicios adicionales para resolver, hay muchos en el texto ya mencionado. Para aquellos estudiantes más avanzados se recomienda el texto *Fundamentos de Programación* del Ms. Jorge Terán que los iniciará en los jueces virtuales para la evaluación automática de problemas y a ejercicios de mayor dificultad, cuenta con un numero importante de ejercicios catalogados por temas.

Para le elaboración del presente texto se ha contado con la valiosa colaboración de Waldo Edgar Callisaya Monzon quien tradujo los programas y el texto original escrito para el lenguaje de programación Java al lenguaje C++.

Agradecimiento especial:
Crear un texto nuevo siempre es una tarea que demanda tiempo y esfuerzo, quiero agradecer a Gustavo Rivera, gerente de Code Road por toda la colaboración prestada.

Atentamente

M.Sc. Jorge Terán Pomier.
Email: teranj@acm.org

Capítulo 1

Introducción

Aprender programación no es solamente conocer un lenguaje de programación. También hay que conocer metodologías para resolver problemas en forma eficiente.

En este curso de programación usamos como base el lenguaje ANSI/ISO C++ para el desarrollo de la programación por dos motivos. El primero porque es un lenguaje que hace una verificación fuerte de los programas detectando una gran variedad de errores y actualmente es un estándar. Segundo porque es utilizado mayoritariamente en las universidades y colegios para enseñar programación.

C++ es un lenguaje orientado a objetos que fue desarrollado por Bjarne Stroustrup. Permite la ejecución de un mismo programa en múltiples sistemas operativos, sin necesidad de recompilar el código. Provee soporte para trabajo en red. Es fácil de utilizar.

1.1. El lenguaje y compilador

Para poder programar en C++ es necesario tener un compilador y un entorno de ejecución. Estos pueden descargarse del sitio:

```
http://www.codeblocks.org/downloads/26
```

Esta versión se denomina Codeblocks en este momento estamos en la versión 10.05, viene integrado con el compilador MinGW, anteriormente conocido como MinGW32, este es una implementación de los compiladores de GCC para la plataforma Win32, que permite migrar la capacidad de este compilador en entorno Windows.

Existen varios compiladores entre los cuales se pueden mencionar GNU GCC Compiler, Borland C++ Compiler, etc. Para diferentes tipos de soluciones se puede encontrar compiladores apropiados.

Cuando revise la literatura encontrara textos de C++ anteriores al ANSI/ISO DE C++, en esta versión se incluye cambios en el lenguaje, en los cuales muchas partes fueron reescritas siguiendo la filosofía orientada a objetos, y los nuevos estándares.

1.1.1. Instalación de MinGW

Para instalar el software en el sistema operativo Windows debe descargar el lenguaje y compilador del sitio indicado.

Al ejecutar el instalador de Codeblocks (codeblocks-10.05mingw-setup) también se estará instalando el compilador MinGw.

La instalación en Linux es más fácil. En las distribuciones basadas en Debian, tales como Ubuntu, el gestor de paquetes ya incluye el compilador para C++. En la línea de comando escribir:

```
sudo apt-get install g++
```

1.2. Construir y compilar un programa

Para construir un programa ANSI/ISO C++ es suficiente utilizar el editor de textos del sistema operativo, o uno de su preferencia. Los pasos a seguir son los siguientes:

1. Copiar el programa en el editor de textos.
2. Guardar el mismo con extensión `cpp`.
3. Compile el programa.
4. Para hacer correr el programa.

Para ejemplificar el proceso construyamos un programa básico y expliquemos las partes que lo constituyen.

1. Todo en C++ necesita archivos de encabezado de la biblioteca estándar. Un encabezado de biblioteca se puede definir como:

   ```
   #include < nombre_biblioteca >
   ```

2. El código fuente se guarda en un archivo ascii, y puede tener cualquier nombre con la extencion `.cpp`.
3. El compilador genera un archivo con extensión `.o` y un archivo ejecutable en Windows.
4. Para que un programa se ejecute de forma independiente y autónoma, deben contener la función `main()` esta función es parte de todos los programas de C++.

   ```
   #include< nombre_biblioteca >
   using namespace std;
   int main (){
   }
   ```

5. Vea que después de colocar la biblioteca que utilizaremos se comienza a escribir el programa.
6. La palabra clave `int` a la izquierda del `main` indica que `main` devuelve un valor entero.
7. Las palabras `using namespace std` especifica que estamos usando el espacio de nombres `std`, que es una característica relativamente nueva de C++. Los espacios de nombres `std` se diseñaron para ayudar a que los programadores desarrollen componentes de software nuevos sin generar conflictos de nombres con los componentes de software ya existentes.
8. Las instrucciones de nuestro código se ingresan en la función `main()`.
9. Una instrucción para mostrar un texto en pantalla es:

   ```
   cout << "texto a mostrar";
   ```

10. La línea:

    ```
    cout << "hola" << endl;
    ```

    le indica a la computadora que imprima en la pantalla la *cadena* contenida ente comillas. La línea

completa incluyendo `cout`, el operador <<, la cadena `" hola"`, el manipulador de flujo `endl` y el punto y coma (;), se llama instrucción. Cada instrucción debe terminar con un punto y coma (*también conocido como terminador de instrucción*).
El manipulador `endl` (abreviatura en ingles de fin de linea) envía a la salida un salto de línea.

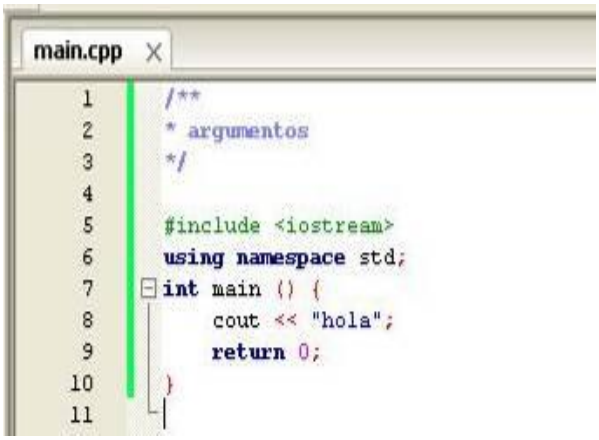


Figura 1.1: Estructura de un programa C++

El programa terminado queda como sigue:

```
#include <iostream>
using namespace std;
int main (){
      cout<< "hola";
      return 0;
}
```

11. Es muy importante aclarar que las letras en mayúsculas y minúsculas se consideran diferentes.

Una descripción de las partes que constituyen un programa se ven en la figura 1.1.

1.3. Herramientas de desarrollo

Existen una variedad de herramientas de desarrollo integrado (IDE) para programas C++. Son ambientes de edición con facilidades para el programador. Las características de una herramienta de desarrollo en relación a un editor de textos convencional son:

1. Más fácil al momento de escribir el programa.
2. Ayuda sobre las diferentes clases y métodos del lenguaje.
3. Depuración de programas en forma sencilla.
4. Es más fácil probar los programas.
5. Es más fácil el imponer normas institucionales.
6. Menos tiempo y esfuerzo.
7. Administración del proyecto.

Para el desarrollo elegimos la herramienta `Codeblocks`. Las razones para esto son las siguientes:

1. Es gratuito y disponible bajo los términos de la GNU versión 3.

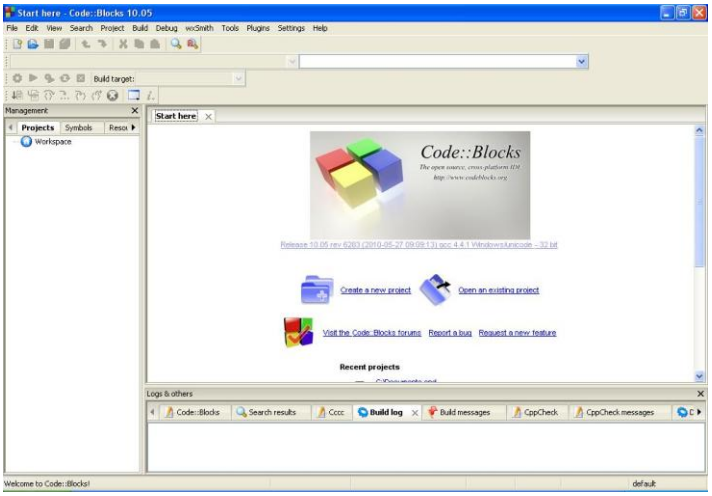


Figura 1.2: Entorno de desarrollo codeblocks

Para utilizar el ambiente para C++ puede descargar `codeblocks 10.05 con mingw` o algún otro, de acuerdo al sistema operativo y hardware que tenga.

2. Codeblock es una plataforma completa de desarrollo figura 1.2. En el curso solo utilizaremos algunas funcionalidades, sin embargo, es deseable comenzar aprendiendo un entorno de desarrollo profesional.

3. Codeblocks tiene muchos complementos que fácilmente ayudan a extender el ambiente de desarrollo a otros lenguajes.

4. Extensas ayudas para aprender C++ y Codeblocks.

5. Fácil de aprender:

6. Puede funcionar tanto en Linux o Windows.

7. Está basado en la plataforma de interfaces gráficas WxWidgets, lo cual quiere decir que puede usarse libremente en diversos sistemas operativos, y está licenciado bajo la Licencia pública general de GNU.

1.3.1. Instalación de Codeblocks

La instalación de Codeblocks es muy simple sea su ambiente de desarrollo Linux o Windows, descargue la herramienta y copie a un directorio de su disco. Luego ejecute Codeblocks seleccionando las opciones adecuadas.

Para la instalación en linux debe descargar el software apropiado. En la línea de comandos escribir:

```
sudo apt-get install codeblocks
```

1.3.2. Construir y hacer correr un programa

El proceso de construir un programa utilizando Codeblocks es sencillo. Hay que tener en cuenta una serie de conceptos que se explican a continuación:

1. Cuando se inicia por primera vez Codeblocks se presentará una pantalla de bienvenida que provee información y ayuda para su uso (figura 1.3). Para salir de esta pantalla se desmarca el ✓ que está al lado de la Show tips at startup. Esta pantalla aparecerá solo la primera vez.

2. Cerrando la pantalla de bienvenida se obtiene el ambiente de trabajo que se muestra en la figura 1.2. Esta pantalla tiene varias partes:

   - La ventana que dice `Projects`. Esta sección nos muestra los proyectos y programas que vamos creando.

   - La parte inferior `Logs & others` cuando ejecutemos el programa con la ayuda de Build nos mostrara los problemas y declaraciones.

   - Una terminal que nos muestra la salida del programa compilado.

   - Al centro se ve un espacio donde se escribirá el código del programa.

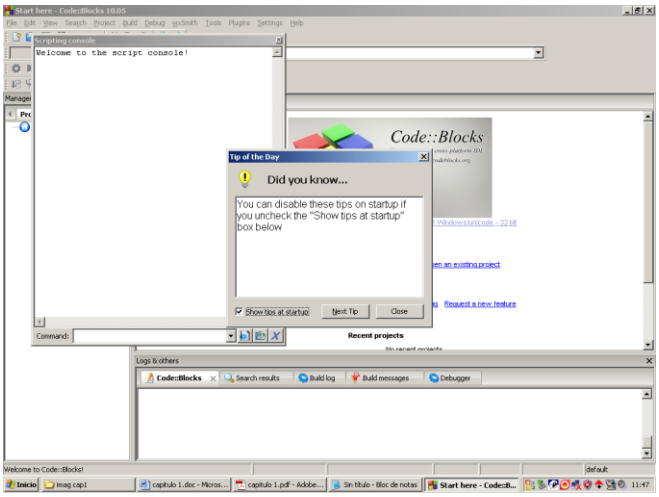


Figura 1.3: Pantalla de bienvenida de Codeblocks

3. Para empezar a crear un programa el primer paso es crear un proyecto. Dentro del proyecto estarán todos los programas del mismo. Desde la pestaña `File` escogemos `new` escojemos `Project` y obtenemos la pantalla de la figura 1.4. elegimos `Console aplication`, ingresamos el `nombre del proyecto` y escogemos `Finish`. En este momento podemos cambiar varios aspectos del proyecto, como ser, la versión de compilador que se utilizará. Un aspecto muy importante que hay que resaltar que si crea un proyecto que no es de C++ no podrá ejecutar un programa C++.

4. Una vez que ha creado un proyecto, `codeblocks` coloca por defecto un archivo main.cpp es en allí en donde tenemos que empezar a crear nuestro programa. Para esto colocamos el cursor de ratón en el nombre del proyecto y abrimos main.cpp. Obteniendo la pantalla de la figura 1.5. Aquí ingresamos programa y escogemos la opción `int main()` para indicar que es un

programa ejecutable.

5. En esta pantalla ya vemos un programa con una `biblioteca iostream`, y una función principal que es donde se introducirán las instrucciones, que se ven en la figura 1.1.

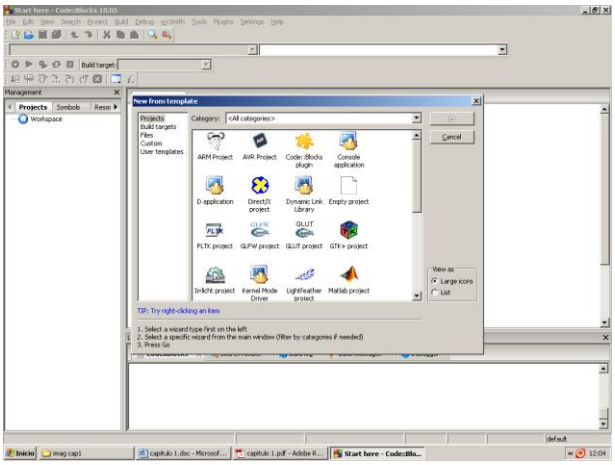


Figura 1.4: Opciones para crear un proyecto

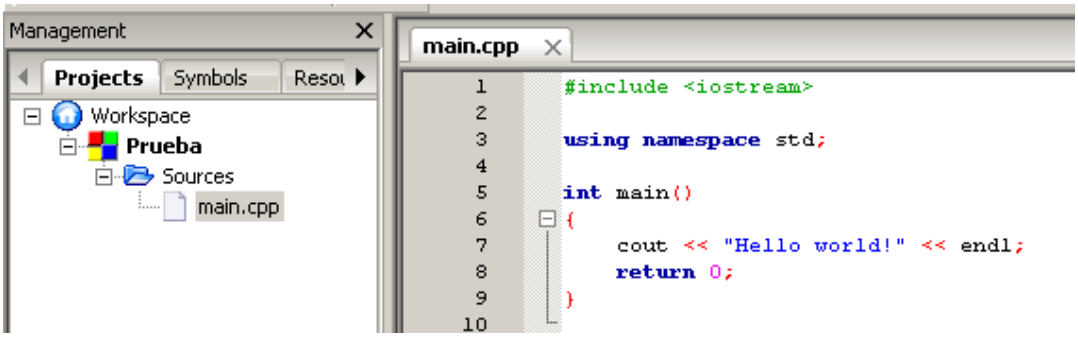


Figura 1.5: Pantalla del main.cpp

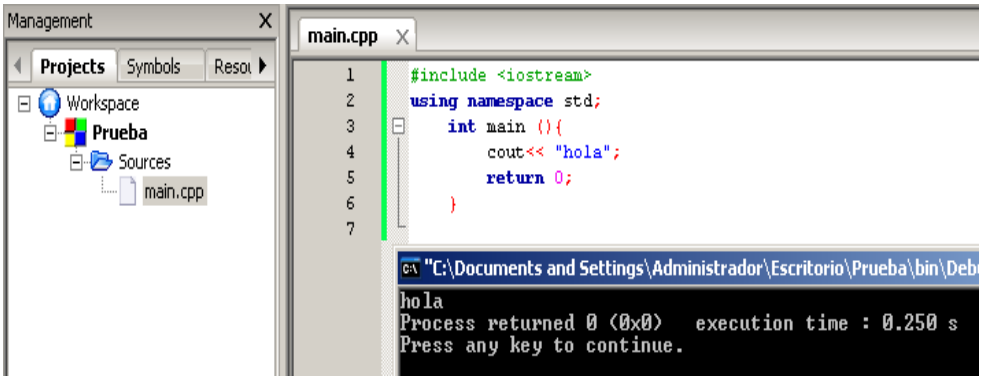


Figura 1.6: Salida del programa

Las líneas de la figura 1.1:

```
/**
* argumentos

*/
```

Es donde colocamos los comentarios generales del programa, tales como parámetros, fecha, autor. Todos los bloques de comentarios (múltiples líneas) comienzan con /* y terminan con */.

6. Cuando un comentario se pone en una sola línea, no es un conjunto de líneas, se utiliza //. El comentario. Aquí describimos que hace el programa.
7. A continuación podemos escribir las instrucciones que se ejecutarán. En el ejemplo pusimos

   ```
   cout << "hola";
   ```

   que especifica que queremos mostrar la palabra ”hola” por pantalla.
8. Antes de ejecutar el programa vea el mismo finalizado (figura 1.6). Para ejecutar el mismo, vamos al menú `Build` y elegimos “`Build and run`”. Ahora veremos la salida del programa en una ventana que se llama `Consola`.

   Si en algún caso no tenemos alguna ventana o la consola visible vamos a `View` escogemos `Perspectives` escogemos `Code::Blocks default`.

1.3.3. Estructura de directorios

Cuando creamos un proyecto y un programa C++ obtenemos la siguiente estructura de directorios:

- En el área de trabajo (workspace) una carpeta con el nombre del proyecto.
- En el interior de la carpeta con el nombre de proyecto una carpeta Sources.
- En la carpeta `Sources` el programa main.cpp.

1.4. Ejercicios

1. Instalar el Editor Codeblocks.
2. Escribir un programa que muestre su nombre utilizando Codeblocks.
3. Busque en qué lugar está el directorio workspace.
4. Describa los directorios creados en el proyecto.
5. Compile y haga correr el programa de la figura 1.1 usando los menús adecuados.

Capítulo 2

Tipos de Datos

2.1. Introducción

Seguramente usted ha utilizado una computadora para realizar una serie de tareas. Y son muy buenas para las tareas repetitivas. Puede realizar las mismas una y otra vez sin cansarse.

Un equipo debe ser programado para realizar tareas. Diferentes tareas requieren diferentes programas.

Por ejemplo para que se pueda escribir un programa primero ha tenido que construirse un editor que nos permite introducir el código y guardarlo en un archivo.

Las computadoras solo ejecutan un conjunto de operaciones básicas muy rápidamente. Con este conjunto básico es que se construyeron los programas que nos permiten posteriormente hacer tareas más complejas. Las instrucciones básicas pueden obtener datos de un lugar de la memoria, sumar dos números, guardar en la memoria, si el valor es negativo continuar en otra instrucción.

Para construir los programas que se usan hoy, tal es el caso de C++, en base de estas instrucciones básicas se crearon lenguajes con muchas más instrucciones para hacer la tarea de programación más sencilla.

Un programa de una computadora es una secuencia de instrucciones necesarias para realizar una tarea.

2.2. Entender la actividad de la programación.

La actividad de la programación consiste en escribir algoritmos para resolver problemas o tareas. Por esto es necesario definir con precisión que entendemos por algoritmo.

Un algoritmo es una secuencia de pasos que tiene un inicio y un final. Quiere decir que finaliza en algún momento. Los pasos deben ser precisos.

Por ejemplo si queremos guardar en B la suma de A + 2 el programa podría ser como sigue:

1. Condiciones iníciales A tiene un valor, el valor de B no nos interesa.
2. Obtener el valor de A.
3. Sumar 2.
4. Guardar el resultado en B.
5. Condiciones finales B contiene el valor de A + 2.

Analizando el ejemplo podemos ver que las instrucciones son precisas y el orden en el que se ejecutan es importante. Si cambiamos el orden seguro que el resultado será diferente.

En los algoritmos no se permiten valores no cuantificados claramente. Por ejemplo un algoritmo para

realizar una receta de cocina podría ser:

1. Poner 1 taza de harina.
2. Agregar una taza de leche.
3. Agregar un huevo.
4. Poner una cucharilla de sal.
5. Mezclar.
6. Hornear a 220 grados.

En este ejemplo la secuencia es precisa. Si por ejemplo si especificamos sal al gusto, deja de ser un algoritmo dado que el concepto al gusto no es un valor preciso.

Otro caso en que una secuencia deja de ser un algoritmo es cuando después de una cantidad de pasos no termina.

1. Inicio.
2. Asignar a N el valor de 100.
3. Repetir hasta que N mayor que 256.
4. Asignar a N el valor N/2.
5. Fin de repetir.
6. Fin del algoritmo.

Claramente se ve que el algoritmo no termina puesto que cada vez vamos reduciendo el valor de N en lugar de que vaya creciendo. Aún cuando se ha definido un inicio y un final, el algoritmo no termina, estará ejecutando continuamente.

Como dijimos los algoritmos son secuencia de pasos, no importando el lenguaje que se utiliza, o como se expresó el mismo. En el curso expresaremos los algoritmos en lenguaje C++. Muchas veces usamos una mezcla de lenguaje español con C++ para explicar una tarea, esto se denomina pseudo lenguaje.

2.3. Reconocer errores de sintaxis y de lógica

Cuando escribimos un programa existen dos tipos de errores, los de sintaxis y los de lógica. Los errores de sintaxis son aquellos en los que existen errores en la construcción de las instrucciones. Por ejemplo: la carencia de un punto y coma al final de una instrucción, una palabra que debe comenzar con mayúsculas y se escribió con minúsculas, un error ortográfico, etc.

Este tipo de errores es detectado por el compilador cuando compilamos el programa. El entorno de desarrollo C++ lo marca como un error ortográfico. Para ejecutar un programa no pueden existir errores de este tipo.

Los errores de lógica son más difíciles de descubrir dado que el compilador no puede detectarlos. Estos errores se producen cuando el programa no hace lo que deseamos. Por ejemplo supongamos que queremos contar el número de números que hay en una secuencia y hemos realizado la suma de los números, claramente es un error de lógica.

2.4. Tipos de datos

Cada valor que utilizamos en C++ tiene un tipo. Por ejemplo “Hola” es de tipo cadena, un número puede ser de tipo entero.

¿Cómo definimos datos en un programa? Para definir un dato la sintaxis que se utiliza en C++ es:

```
Nombretipo nombreVariable = valor;
```

Los nombres se construyen bajo las siguientes restricciones:

1. Comienzan con una letra mayúscula o minúscula.
2. Pueden contener letras y números.
3. También puede incluir el símbolo guión bajo (_) o el símbolo dólar ($).
4. No se permiten espacios.

Algunos nombres válidos son *nombre*, *caso_1*,*cedulaIdentidad*. Por convención de los nombres deben comenzar con una letra minúscula.

El signo = se denomina operador de asignación y se utiliza para cambiar el valor de una variable. En el ejemplo `nombreVariable` cambia su contenido con el contenido de `valor`.

Hay dos tipos de datos, los del núcleo del lenguaje denominados tipos primitivos o básicos y los implementados en clases. Los tipos básicos son tabla los observados en la tabla 1.1.

Tipo	Descripción	Tamaño
bool	Valor booleano: true o false	1 bits
int	Números enteros en el rango -2,147,483,648 a 2,147,483,647	32 bits
long	Numeros enteros en el rango -2,147,483,648 a 2,147,483,647	32 bits
short	Números enteros en el rango de -32,768 a 32,767	16 bits
double	Número de precisión doble de punto flotante en el rango de 2.23 E - 308 a 1.79 E + 308	64 bits
float	Número de precisión simple de punto flotante en el rango de 1.8 E - 38 a 3.40 E + 38	32 bits
char	Caracteres en formato Unicode	8 bits

Tabla 1.1 tipos de datos en C++

Los caracteres de la tabla Ascii normalmente se representan en un byte, sin embargo, para poder representar caracteres en diferentes lenguajes (español, inglés, árabe, etc.), se utiliza una norma denominada Unicode.

Un byte representa 8 bits o sea 8 dígitos binarios. Un número entero de 4 bytes utiliza un bit para el signo y 3 para almacenar un número. De esta consideración se deduce que el numero más grande que se puede almacenar es $2^{31} - 1 = 2, 147, 483, 647$.

Para definir los valores numéricos podemos utilizar una de las siguientes sintaxis.

- tipo nombre;
- tipo nombre = valor;

Vea que cada instrucción termina con un punto y coma. Por ejemplo:

```
int i;
```

Esto permite definir una variable de nombre i de tipo entero que no tiene un valor inicial. En este caso el valor inicial es NULL;

Si definimos:

```
int i=3;
```

Significa que estamos definiendo una variable de nombre i con valor inicial 3.

En el caso de las variables de punto flotante es necesario poner un punto decimal para indicar que los valores son del tipo con decimales. Por ejemplo:

```
double f=10.0;
```

No podemos colocar solo 10 porque este es un número entero y es necesario que los tipos igualen en toda asignación.

Errores de desborde

Cuando realizamos una operación con una variable, y excedemos el valor máximo que podemos almacenar, se produce un desborde. Supongamos que en una variable de tipo short tenemos almacenado el número 32767. Si agregamos 1 a ésta, no obtendremos 32768, el resultado es -32768 porque hemos excedido el número máximo que podemos almacenar en este tipo de variable.

Para entender esto, vemos que pasa, convirtiendo esto a números binarios. Si al valor 32767 = 01111111, sumamos uno la respuesta es 10000000 que en la representación de la computadora equivale a −32768

El desborde lo identificamos cuando al realizar una operación el resultado es incorrecto. Por ejemplo si esperamos un resultado positivo y obtenemos un resultado negativo.

Errores de redondeo

Cuando representamos un número en notación de punto flotante, el resultado se expresa de la forma *enteros.decimalesEexponente*. Por ejemplo 5,6666664E7 significa $5{,}6666664 \times 10^7$. Con esta representación las conversiones no son exactas. Por ejemplo 10/3 es 3,3333333333333333 sin embargo en la computadora nos da 3,3333333333333335. Esto podemos probar con el siguiente código:

```
#include<iostream>
#include<cstdio>
using namespace std;
int main(){
          double a = 10.0/3;
          printf("a= %.16f",a);
```

```
return 0;
}
```

Debido a estos errores que se producen hay que tener mucho cuidado como se manejan los números de punto flotantes para no obtener resultados erróneos. Esto es principalmente crítico en cálculos financieros donde no se permiten errores por la precisión de la representación de los números en la computadora.

2.4.1. Ubicación en la memoria de la computadora

Todas las variables se almacenan en la memoria de la computadora en forma secuencial. Esto quieres decir uno a continuación del otro.

Cuando tenemos una definición, por ejemplo, int a = 1234; el contenido de la variable a es 1234. El nombre a representa la dirección de memoria donde está ubicado. En este caso donde comienzan los 4 bytes de la variable a.

En la figura 1.7 podemos ver que el nombre de una variable básica es un apuntador a su contenido, que se almacena en la cantidad de bytes que corresponde al tipo de la variable.

Cuando el tipo no corresponde a uno básico, el nombre de la variable no apunta al contenido. Consideremos como ejemplo una cadena. Una cadena de caracteres puede tener cualquier longitud. Por esto no es posible, que el nombre apunte al contenido. Lo que se hace es que el nombre apunte a un lugar, donde se encuentra la dirección de memoria donde está la cadena.

Figura 1.7: Direccionamiento de las variables en la memoria

Lo que decimos es que la variable es un puntero a la dirección de memoria donde está el contenido. Si hacemos una operación con ésta variable, por ejemplo sumar 1, se estaría cambiando la dirección del contenido. Para evitar esto toda la operación sobre variables que no corresponden a los tipos básico se realizan con métodos.

2.4.2. Variables y constantes

Se pueden definir dos tipos de datos, los *variables* y los *constantes*. **Variables** son aquellos que pueden cambiar su valor durante la ejecución del programa. **Constantes** son aquellos que no pueden cambiar su valor durante la ejecución del programa.

Las variables se definen como vimos en la definición de variables. Simplemente se coloca el tipo y el nombre de la misma.

Para definir un valor constante por ejemplo,

```
double PI=3.1416;
```

puede realizarse como una variable normal. Esta definición no permitiría evitar que cambiemos el valor de `PI` . Si alteramos el valor por un error de lógica será difícil de hallar el mismo.

Para especificar al compilador que no se puede modificar el valor utilizaremos la palabra const en la definición quedando:

```
const double PI=3.1416;
```

Ahora, si tratamos de modificar esta variable obtendremos un error de compilación.

2.5. Caracteres

Para manejar caracteres en C++ tenemos el tipo de datos char que no tiene métodos. Los caracteres corresponden a los caracteres de la tabla ascii. Mostramos algunos de los caracteres en la tabla 1.2:

	0	1	2	3	4	5	6	7	8	9
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w

Tabla 1.2 El conjunto de caracteres ASCII

Los caracteres están acomodados correlativamente, si vemos el carácter 0 es número 48 el carácter 1 es el 49 y así sucesivamente. Si vemos las letras mayúsculas comienzan en el 65 hasta el 90

Para definir un carácter tenemos dos opciones:

- Asignamos a la variable el valor numérico del carácter. Por ejemplo para definir la letra A podemos escribir:

```
char letraA=65;
```

- Si no conocemos el valor ascii del carácter podemos escribir el mismo entre apostrofes.
  Para definir la misma letra A escribimos

```
char letraA='A';
```

2.5.1. Interpretación de los datos

Cuando escribimos un número éste puede representar tanto un carácter como un número.
¿Cómo sabe el compilador si es un número o un carácter?

El compilador no puede determinar esto. Uno debe decir que es lo que representa. Para esto se utiliza un concepto que en C++ se denomina *cast*. La sintaxis consiste en colocar el tipo entre paréntesis delante de la variable.

Por ejemplo si queremos asignar el carácter 80 de una variable entera a una variable carácter debemos hacer un *cast*. En una asignación ambos lados deben ser del mismo tipo. Un ejemplo seria:

```
int i=80;
char c=(char)i;
```

Lo mismo ocurre con todos los tipos de variables, por ejemplo para asignar una variable *int* a una *long*

```
int i=80;
long l=(long)i;
```

2.5.2. Salida por pantalla

Para mostrar los datos por pantalla se utiliza cout con la biblioteca iostream, cout solo puede mostrar secuencias de caracteres en la pantalla. Cuando imprimimos un número primero se convierte en una cadena de caracteres y luego se imprime. Por ejemplo si tenemos int i = 97 y deseamos imprimir el valor de i primero se debe imprimir el carácter 9 luego el 7 para que se muestre el número 97. Si se envía directamente el 97 se mostrará por pantalla la a.

El código es el siguiente:

```
int i=97;
cout << i;
```

Ahora, si queremos que en lugar de 97 se imprima la letra *a* hacemos un *cast*, el código es el siguiente:

```
int i=97;
cout << (char)i;
```

Para imprimir un texto, se coloca el mismo entre comillas.

```
cout << "hola";
```

Si se quiere imprimir textos, y números debe separar texto, variables y constantes. Esto se hace con el símbolo <<. Por ejemplo

```
int i=123;
cout << "El valor de i es " << i;
```

En el ejemplo tenemos una cadena y luego i es convertido a cadena con cout, dando un resultado que es una cadena. Si codifica

```
int i=123;
cout << i << " es el valor de i";
```

No producirá error de sintaxis. Por usar el separador <<.

Colocaremos una cadena al principio. Esta cadena de longitud cero, porque no encierra nada, se denomina cadena de valor null. La forma correcta de codificar es:

```
int i=123;
cout << "" << i<< "es el valor de i";
```

Caracteres especiales

Existen algunos caracteres especiales que son:

Carácter	Descripción
\t	El carácter de tabulación, que se denomina tab. En la tabla ascii es el carácter 9
\r	Retorno de carro, Significa volver al principio de la línea, es el carácter 13.
\n	Salto de línea. Coloca el cursor al inicio de la siguiente línea.

Tabla 1.3 caracteres especiales

Cuando usamos cout << ; se muestran los resultados, se muestran en una línea de la pantalla pero no se avanza a la siguiente línea. Si queremos que los caracteres siguientes contienen en la próxima línea debemos indicar esto con los caracteres especiales.

Dependiendo del sistema operativo hay diferencias. Para evitar este problema podemos utilizar la instrucción cout << "algún mensaje"<<endl;. La palabra endl indica que una vez completada la instrucción se imprima los caracteres necesarios para avanzar una línea e ir al comienzo.

Si deseamos incluir estos caracteres en una cadena solo lo incluimos en medio del texto. La instrucción cout << "que \n dice"; hará `que` se imprima que en una línea y `dice` al principio de la siguiente línea.

Como vimos el carácter \indica que a continuación hay un carácter de control. Si queremos imprimir el carácter \ entonces debemos colocar dos \\.

2.5.3. Despliegue de números con formato

Consideremos el siguiente código:

```
double total = 35.50;
cout << "Total= "<<total;
```

Al ver la respuesta vemos que el resultado es Total = 35,5. Bien aún cuando representa el mismo número la salida presenta un solo decimal. El formato de salida elimina los ceros de la izquierda y los de la derecha después del punto decimal.

Para imprimir valores numéricos con un formato específico se utiliza la instrucción printf() que tiene la siguiente sintaxis:

```
printf("formato",variable);
```

En formato se escriben los textos y las características de formato de la variable que queremos imprimir. Por ejemplo para imprimir Total = 35.50 en el campo de formato escribimos"Total %5.2f". Esto significa que imprimiremos la palabra Total seguida de un espacio luego viene una variable de punto flotante de tamaño fijo de 5 caracteres de los cuales 2 son decimales. Para el ejemplo la instrucción es:

```
printf("Total %5.2f",total);
```

Los formatos más comunes se muestran en la siguiente tabla:

Código	Descripción	Ejemplo
d	Decimal	123
x	Entero hexadecimal	4B
o	Entero octal	12
f	Punto flotante	35.50
e	Punto flotante con exponente	1.25e+2
s	Una cadena	Hola
n	Fin de línea independiente del sistema operativo	
0	Mostrar los ceros de la izquierda	0012
+	Mostrar el signo + en los números	+123
(	Los números negativos se muestran entre paréntesis	(123)
,	Mostrar separadores de miles	1,234

Tabla 1.4 formatos

Veamos unos ejemplos:

Formato	Salida
printf(" %x",1234)	4d2
printf(" %6.2f %6.2f",12.12,25.45)	12,12 25,45
printf(" %+d",-1234)	-1234
printf("(%d",-1234)	(1234
printf(" %d,",1234)	1234,
printf(" %06d",1234)	001234

Tabla 1.5 formatos de salidas

No se debe olvidar que para que se realice el despliegue en pantalla de números con formato se hace uso de la biblioteca `#include<cstdio>`, esta biblioteca contiene los prototipos de la biblioteca estándar de entrada/salida de información.

2.6. Ejercicios

1. Problemas de tipos de datos y caracteres

La lectura de datos es de teclado. Los resultados se muestran por pantalla.

a) Los siguientes ejercicios tienen la finalidad de que conozca los caracteres y los pueda encontrar en el teclado. Solo debe utilizar la instrucción cout con \n. Lo que el programa debe realizar es imprimir en pantalla las salidas que se muestran en cada uno de los incisos.

1)
```
+-----------+
| Su nombre |
+-----------+
```

2)
```
 -------
    -_-
 -------
```

3)
```
<------->
<_______>
```

```
<%%%%%%%>
</////////>
```

4)
```
 +--------+
+         +
 +--------+
+         +
 +--------+
+         +
 +--------+
+         +
+--------+
```

b) En los siguientes incisos utilice la instrucción cout con \n para cada enunciado.

1) Muestre el código ascii de los caracteres: 0,A,a,Ñ ,ñ.
2) Muestre el código ascii de todas las vocales con acento.
3) Muestre el carácter correspondiente a los enteros: 49, 66,64,97.
4) Muestre la representación binaria de las letras: A, a y describa que bits son diferentes.
5) Escriba una instrucción que muestre el resultado de sumar uno a los caracteres: 0, A, a, Ñ, ñ.

c) Escriba una instrucción printf() que produzca el resultado mostrado

1) El resultado de 1/3 con 2 dos enteros y 3 decimales
2) El valor máximo una variable entera con separadores de miles.
3) Tres números enteros de dos dígitos, cada uno ocupando un espacio de 5 caracteres.
4) Escriba un número flotante, y un número entero separados por un espacio.

d) Considere la siguiente descripción de datos

```
double a1=123.77,
a2 = 425.23,
a3 = 319.44,
a4 = 395.55;
```

Si deseamos imprimir el 10 % de cada uno de los valores el código es:

```
printf("\%f\%n",a1/10);
printf("\%f\%n",a2/10);
printf("\%f\%n",a3/10);
printf("\%f\%n",a4/10);
```

Para imprimir el 10 % de la suma de los valores escribimos:

```
printf("\%f\%n",(a1+a2+a3+a4)/10);
```

La suma de los porcentajes individuales no iguala con el porcentaje de la suma

Capítulo 3

Operadores aritméticos y lectura de teclado

3.1. Introducción

En este capítulo se explica cómo trabajar con los valores enteros utilizando operaciones tanto en bits individuales como en números. Como convertir de un tipo de datos a otro y finalmente como podemos ingresar los mismos por teclado.

3.2. Trabajando en binario

Cuando sea posible es preferible trabajar en binario, dado que, las computadoras trabajan en números binarios y esto mejorará el tiempo de proceso. Los operadores disponibles para trabajar manejo de bits son:

Operador	Descripción	Ejemplo
<<	Recorrer bits a la izquierda, insertando un bit 0 por la derecha. Los bits sobrantes a la izquierda se pierden.	Si tenemos 101 y recorremos un bit a la izquierda el resultado es 1010
>>	Recorrer bits a la derecha, insertando un bit 0 por la izquierda. Los bits sobrantes a la derecha se pierden.	Si tenemos 101 y recorremos un bit a la derecha el resultado es 10
&	Realiza una operación lógica and bit a bit	101&110 = 100
|	Realiza una operación lógica or bit a bit	101|110 = 111
ˆ	Realiza una operación lógica xor bit a bit	101 ˆ 110 = 011

Analicemos uno a uno esto operadores y veamos el uso de cada uno de ellos.

3.2.1. El operador de desplazamiento

El operador de desplazamiento es el que permite recorrer bits a la izquierda o derecha. Si definimos un número entero `int i = 10` la representación binaria del contenido de la variable `i es 1010`.

El operador de desplazamiento es un operador binario. Esto significa que tiene dos operadores, un número y la cantidad de bits a desplazar. Veamos el código siguiente:

```
int i = 10;
i=i<<1;
```

Esto significa que los bits de la variable `i` recorrerán a la izquierda un lugar. El resultado será 20 en decimal o 10100 en binario.

Si utilizamos el operador `i >> 1` se desplazaran los bits a la derecha dando como resultado 101 cuyo equivalente decimal es 5.

La base de los números binarios es 2, por lo tanto, agregar un cero a la derecha es equivalente a multiplicar por dos. Eliminar un bit de la derecha es equivalente a dividir por 2. Recorrer 2 bits a la izquierda es equivalente a multiplicar por 2 dos veces o sea multiplicar por 2^2, recorrer 3 bits, multiplicar por 2^3, así sucesivamente. Similarmente ocurre lo mismo al recorrer a la derecha, pero dividiendo en lugar de multiplicar.

3.2.2. El operador lógico and

El operador lógico `and` se representa con el símbolo & y permite realizar esta operación lógica bit a bit. La tabla siguiente muestra los resultados de realizar una operación and entre dos bits:

Operación	resultado
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

Si nos fijamos en el operador & vemos que solo cuando ambos bits son 1 el resultado es 1. Esto es equivalente a multiplicar ambos bits. ¿Cuál es el uso para este operador? Este operador se utiliza para averiguar el valor de uno o más bits. Por ejemplo si tenemos una secuencia de bits y realizamos una operación `and`

```
xxxxyxyyxxyx
000011110000
------------ and
0000yxyy000
```

No conocemos que son los bits `x` y `y` pero sabemos que el resultado será el mismo bit si hacemos & con 1 y 0 si hacemos la operación & con 0. La secuencia de bits `000011110000` del ejemplo se denomina máscara.

Si queremos averiguar si un número es par, podemos simplemente preguntar si el último bit es cero y esto se puede hacer con una operación and donde todos los bits se colocan en cero y el bit de más a la derecha en 1. Si el resultado es cero el número será par, si es uno impar.

3.2.3. El operador lógico or

El operador lógico `or` se representa por **|** y realiza la operación que se muestra en la tabla bit a bit:

Operación	resultado
0|0	0
0|1	1
1|0	1
1|1	1

Esta operación permite colocar un bit en 1 vea que cada vez que se hace | con 1 el resultado es siempre 1. Cuando queremos colocar bits en uno usamos esta instrucción.

3.2.4. El operador lógico xor

El operador lógico xor se representa por ^ y realiza la operación xor que se muestra en la tabla bit a bit:

Operación	resultado
0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

Esta operación es equivalente a sumar los bits. Si realizamos una operación `xor` con de una variable consigo misma es equivalente a poner la variable en cero. Una de las aplicaciones del `xor` es el cambiar todos los bits de una variable, lo unos por ceros y los ceros por unos.

Si realizamos una operación `xor` con todos los bits en 1 se obtendrá esto. Por ejemplo 1010 ^ 1111 dará el resultado 0101.

Otra característica es que si realizamos `xor` con un valor dos veces se obtiene otra vez el valor original. Veamos, por ejemplo, 43 = 101011, 57 = 111001 si hacemos 43 ^ 57 el resultado es 18 = 010010, ahora si hacemos 43 ^ 18 obtenemos otra vez 57. Del mismo modo 57 ^ 18 = 43.

3.2.5. Aplicación de manejo de bits

Una de las aplicaciones del manejo de bits es el de mostrar un número entero por pantalla en binario. Tomemos por ejemplo el número 43 = 101011. Vemos el código

```
int i = 43;
cout<<i;
```

Por pantalla se verá el número 43. Ahora si queremos mostrar el equivalente en binario, tenemos que trabajar bit a bit, y realizamos lo siguiente:

1. Creamos una máscara con un 1 en el primer bit que debemos mostrar
2. Realizamos una operación and.
3. Recorremos este bit al extremo derecho.
4. Mostramos el bit.

5. Recorremos la máscara un lugar a la derecha
6. Repetimos el proceso con todos los bits

Para el ejemplo el código seria:

```
#include<iostream>
using namespace std;
int main(){
    int i = 43;
    cout<< i << "=";
    // mostramos el bit 5
    int mascara = 32;
    int bit = i & mascara;
    bit = bit >> 5;
    cout<< bit;
    // mostramos el bit 4
    mascara= mascara >>1;
    bit = i & mascara;
    bit = bit >> 4;
    cout<<bit;
    // mostramos el bit 3
    mascara= mascara >>1;
    bit = i & mascara;
    bit = bit >> 3;
    cout<<bit;
    // mostramos el bit 2
    mascara= mascara >>1;
    bit = i & mascara;
    bit = bit >> 2;
    cout<<bit;
    // mostramos el bit 1
    mascara= mascara >>1;
    bit = i & mascara;
    bit = bit >> 1;
    cout<<bit;
    // mostramos el bit 0
    mascara= mascara >>1;
    bit = i & mascara;
    cout<<bit;
    return 0;
 }
```

3.3. Trabajando con variables

Para trabajar directamente en números sin preocuparnos de que internamente están definidos en binario, tenemos las siguientes operaciones binarias:

Operación	Descripción
+	Suma de dos variable
-	Resta de dos variables
/	División de dos variables
%	Hallar del resto de una división

y las siguientes operaciones incrementales

Operación	Descripción
++	Incrementa en uno
- -	Decremento en uno

Para realizar operaciones aritméticas entre dos variables del mismo tipo simplemente utilizamos el operador deseado y asignamos el resultado a otra variable. Por ejemplo para sumas A con B y guardar el resultado en C escribimos c = A + B.

La expresión 2 + 3* 5 puede dar dos resultados de acuerdo al orden en que se realicen las operaciones. Si primero hacemos la suma 2 + 3 = 5 y multiplicamos por 5 el resultado es 25 si primero hacemos la multiplicación 3* 5 = 15 y luego la sumamos 2 el resultado es 17. El resultado debe evaluarse en función de la prioridad de los operadores. Primero se evalúa la multiplicación y después la suma.

Los operadores que realizan operaciones incrementales, solo trabajan sobre una variable. Por ejemplo A + + incrementa la variable A en uno.

Los operadores se evalúan de acuerdo al orden de precedencia que se muestra en la siguiente tabla. Si tienen el mismo orden de precedencia se evalúan de izquierda a derecha.

Prioridad	Operador
1	( )
2	++ –
3	*/ %
4	+ -
5	>>, <<
8	&
6	ˆ
7	|

3.3.1. Ejemplos de expresiones

Las siguientes expresiones se evalúan de acuerdo a la prioridad de los operadores y dan los resultados mostrados:

Expresión	Resultado
3 + 2 * 3	Primero se realiza la multiplicación luego la suma el resultado es 9
(3 + 2) * 3	Primero se realizan las operaciones entre paréntesis luego la multiplicación el resultado es 15
(3 + 2* 3)* 2	Primero se realiza lo que está entre paréntesis de acuerdo a la prioridad de operaciones, dando 9 luego se multiplica por 2 el resultado es 18
3/2* 3	Como las operaciones son de la misma prioridad primero se hace 3/2 como son números enteros el resultado es 1 luego se multiplica por 3 dando 3
3* 3/2	Primero se hace 3* 3 = 9 dividido por 2 da 4
3,0/2,0* 3,0	Como los números son de punto flotante 3,0/2,0 el resultado es 1,5 por 3 da 4.5
+ + a + b	Primero se incrementa el valor de a y luego se suma b
a + b + +	Primero se suma a + b luego se incrementa el valor de b

3.3.2. El archivo de encabezado cmath

Como se ve no hay operadores para hacer ráiz cuadrada, exponencial, funciones trigonométricas, etc. Para estas funciones disponemos del encabezado `#include<cmath>`. La tabla muestra algunos métodos de cmath.

Método	Descripción
fabs(a)	Devuelve el valor absoluto de a
sqrt(a)	Devuelve la raíz cuadrada de a
pow (a,b)	Devuelve el valor de a^b
cos(a)	Devuelve el coseno de a
sin(a)	Devuelve el seno de a

Por ejemplo, dados dos valores `A, B` dados por `A=2, B=3` para hallar A^B escribimos la expresión `pow (a,b)`.

3.4. Operadores de asignación

Para facilitar la escritura de expresiones se han creado los operadores de asignación. Por ejemplo si necesitamos realizar una operación con una variable y deseamos guardar el resultado en la misma variable utilizamos los operadores de asignación. Por ejemplo `a+ = b` es equivalente a `a = a + b`.

Ejemplo	Equivalencia
a+ = expresion	a = a + expresion
a− = expresion	a = a − expresion
a* = expresion	a = a * expresion
a/ = expresion	a = a/expresion
a % = expresion	a = a %expresion

3.5. Convertir el tipo de datos

Para realizar conversiones de datos es necesario hacer una cast. Veamos algunos ejemplos.

- Para eliminar los decimales de una variable de punto flotante la llevamos a una de tipo entero como sigue

```
float f = 34.56;
int i = (int)f;
```

- Si deseamos que una variable int se convierta en long

```
int i = 34;
long l = (long)i;
```

- Si hacemos el proceso inverso vale decir convertir de long a int, solo se obtiene el resultado correcto si el valor puede ser almacenado en una variable entera. El código siguiente claramente da un resultado equivocado.

```
long l = LONG_MAX;
int i = (int)l;
cout << l <<endl;
cout<< i <<endl;
```

El valor de l es `2147483647` y el resultado en la variable i es `32767`. Claramente incorrecto.

En los números de punto flotante se trabaja de la misma forma

```
double d = 55.34;
float f = (float)d;
```

Si el valor no puede almacenarse en la variable `f` el resultado es Infinity.

3.6. Lectura del teclado

Para leer datos del teclado utilizamos el archivo de encabezado <iostream> que contiene las funciones de entrada y salida estándar.

Es necesario crear una instrucción `cin >>` variable para poder utilizar la función y leer del teclado.

Para leer una variable entera el código queda como sigue:

```
#include<iostream>
using namespace std;
int main(){
  int lee;
  cin>>lee;
  return 0;
}
```

Cada lectura de una variable entera inicia después de los espacios y termina cuando se encuentra el primer espacio, que se utiliza como delimitador entre número. Por ejemplo para leer dos números `i, j` que están en una línea

```
123 987
```

Codificamos:

```
#include<iostream>
using namespace std;
 int main(){
    int i, j;
    cin>>i>>j;
    return 0;
 }
```

En los números de punto flotante se trabaja de la misma forma

```
double d = 55.34;
float f = (float)d;
```

Si el valor no puede almacenarse en la variable f el resultado es Infinity

Si los datos de la entrada están definidos en múltiples líneas el resultado es el mismo.

3.7. Errores de redondeo

En muchos casos se producen errores de redondeo cuando no se pueden convertir exactamente los datos.

```
double f = 4.35;
cout << 100 *f<<endl;
// Imprime 435
```

Utilice el método ceil() para redondear al entero más pequeño no menor que el valor.

```
#include<iostream>
#include <cmath>
using namespace std;
 int main(){
   double f = 4.396;
```

```
        long redondeo = ceil(f);
        cout << redondeo<<endl;
    }
```

Para convertir de tipo utilice cast colocando el tipo entre paréntesis.

```
(int) (balance  * 100)
```

3.8. Ejercicios

Para todos los ejercicios siguientes debe ingresar los datos por teclado y mostrar los resultados por pantalla en una sola fila.

1. Escriba un programa que dado un carácter en letras mayúsculas y luego mostrar por pantalla el carácter en letras minúsculas. Para esto utilice una instrucción `Xor`, que le permitirá cambiar de mayúsculas a minúsculas y de minúsculas a mayúsculas con la misma intención.

   ```
   Ejemplo de entrada
   A
   Ejemplo de salida a
   ```

2. Escriba un programa que lea un carácter del teclado e imprima 0 cuando la letra es mayúscula y 1 cuando es minúscula. para esto utilice la instrucción and y desplaza- miento de bits.

   ```
   Ejemplo de entrada
   A
   Ejemplo de salida
   0
   ```

3. Escriba un programa que lea un número del teclado y con manejo de bits imprima uno o cero según sea par o impar imprima 0, 1 respectivamente.

   ```
   Ejemplo de entrada
   123
   Ejemplo de salida
   1
   ```

4. Ingrese un número por teclado luego desplace 3 bits a la derecha. Muestre el número. Luego escriba una instrucción de división que logre el mismo resultado.

5. Escriba un programa C++ que dados dos números enteros imprima

   La diferencia

   El producto

   El promedio

   La distancia, el valor absoluto de la diferencia

   El máximo

   El mínimo

```
Ejemplo de entrada
5 7
Ejemplo de salida
-2 35 6.0 2 7 5
```

6. Escriba un programa que dada la altura y el diámetro en centímetros de una lata cilíndrica de refresco calcule el volumen.

```
Ejemplo de entrada
10 7
Ejemplo de salida
384.84510006474966
```

7. Escriba un programa que dados dos puntos $A(x_1, y_1)$, $B(x_2, y_2)$ escriba la ecuación de la recta que pasa por ellos.

```
Ejemplo de entrada
0 1 5 6
Ejemplo de salida y = x +1
```

8. Dadas dos circunferencias representadas por un punto y su radio que no se intersectan. Escriba un programa que despliegue el centro y radio de la circunferencia más pequeña que intersecta a ambas.

```
Ejemplo de entrada
0 1 2
0 6 1
Ejemplo de salida
0 4 1
```

9. La figura 1.8 muestra un triángulo rectángulo que se utilizará en este y los siguientes ejercicios. Ingrese los valores de a y β halle el valor de c

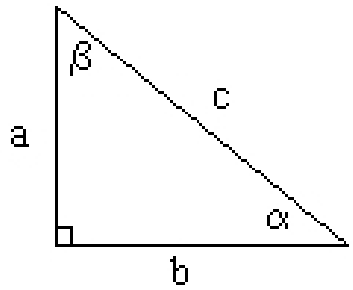


Fig. 1.8 triangulo

10. Ingrese los valores de a, b, c y muestre el perímetro del triángulo.

11. Ingrese los valores de b y α y calcule el valor de c.

Capítulo 4

Estructuras de control

4.1. Introducción

Los lenguajes de programación implementan tres tipos de instrucciones

- Instrucciones de asignación
- Instrucciones de iteración
- Instrucciones de comparación

Los operadores de asignación ya fueron explicados, permiten asignar un valor a una variable. Los operadores de iteración permiten repetir bloques de instrucciones. Y los operadores de condición se utilizan para comparar valores. En el capítulo explicaremos éstas instrucciones y el uso en la resolución de problemas.

4.2. Agrupamiento de instrucciones

Las instrucciones se pueden agrupar en bloques con el uso de corchetes `{}`. Esto permite que las definiciones de variables tengan un ámbito de vigencia. Vemos el código siguiente:

```
{
int i=1;
cout<< i << endl;
}
cout<< i << endl;
```

La variable i está definida dentro de un bloque y por esto solo es accesible en su interior. Esto se puede apreciar en las instrucciones `cout<< i << endl;` La primera instrucción permite mostrar el valor de `i` en pantalla. La segunda que esta fuera del bloque muestra el error *cannot be revolved* que indica que es una variable no accesible en este lugar.

Si queremos hacer esta variable disponible dentro y fuera del bloque hay que definirla fuera del bloque.

```
int i=1;
{ cout<< i << endl;
}
     cout<< i << endl;
```

Las variables que se definen dentro del bloque se denominan variables locales. En otros casos se denominan variables globales.

4.3. Estructuras de control condicionales

Las estructuras de control condicionales son las que nos permiten comparar variables para controlar el orden de ejecución de un programa. Estas estructuras de control implementan con la instrucción `if, if else, if else if, y ?`. Estas instrucciones nos permiten decidir que ciertas instrucciones que cumplan una determinada condición se ejecuten o no.

4.3.1. Estructura de control if

El diagrama 1.1 muestra el flujo de ejecución de una instrucción if.

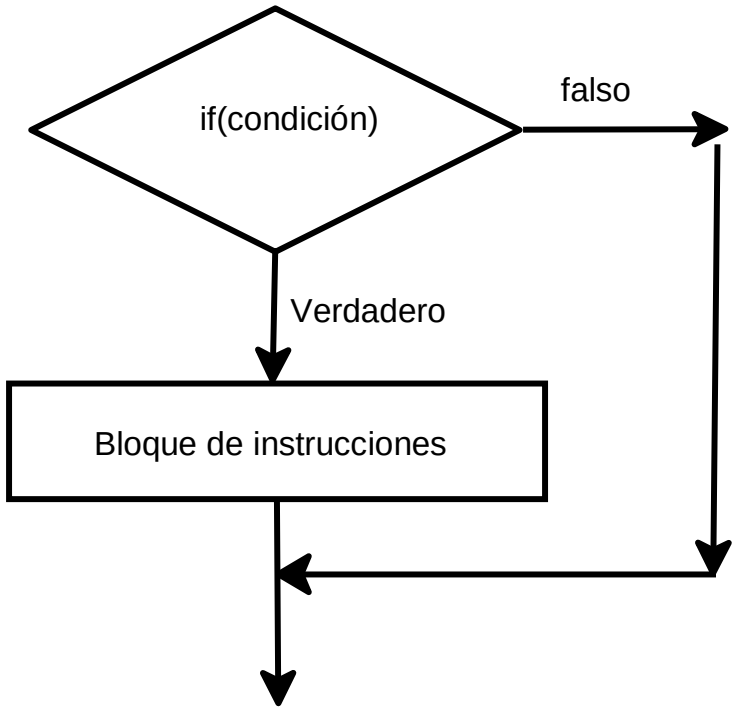


Figura 1.1: Flujo de ejecución de un if

La sintaxis de C++ es:

```
if (condición) {
bloque de instrucciones
}
```

o

```
if (condición)
instrucción;
```

Note que después de los paréntesis del if no se coloca un punto y coma.

4.3.2. Operadores condicionales

Los operadores condicionales para comparar variables se muestran en la tabla siguiente:

Operador	Descripción
==	igual
<	menor que
<=	menor o igual
>	mayor que
>=	mayor o igual
! =	no igual
!	negar

Los operadores condicionales trabajan con dos variables y el resultado puede ser verdadero (true) o falso (false). Por ejemplo `a > b, a == b`, etc. Hay que tomar en cuenta que los operadores `==, <=, > 0, ! =` se escriben sin espacios.

Ejemplo.

Supongamos que queremos resolver la ecuación `ax + b = 0`. Se ingresan los valores de a, b por teclado, y se imprime el valor calculado de x. La solución de esta ecuación es `x = −b/a.` Como sabemos solo existe si a es mayor que cero. Para resolver esto realizamos el siguiente programa:

```
#include<iostream>
using namespace std;
 int main(){
   int a,b;
   cin>> a >>b ;
   if(a!=0)
     cout << (float) -b/a << endl;
  return 0;
 }
```

La instrucción `if (a!=0)` verifica que se realizará una división por cero. Cuando `a` es diferente a cero se calcula el resultado de `x.` Cuando `a = 0` no obtiene ningún resultado.

4.3.3. Estructura de control if else

El diagrama 1.2 muestra el flujo de ejecución de una instrucción if else.

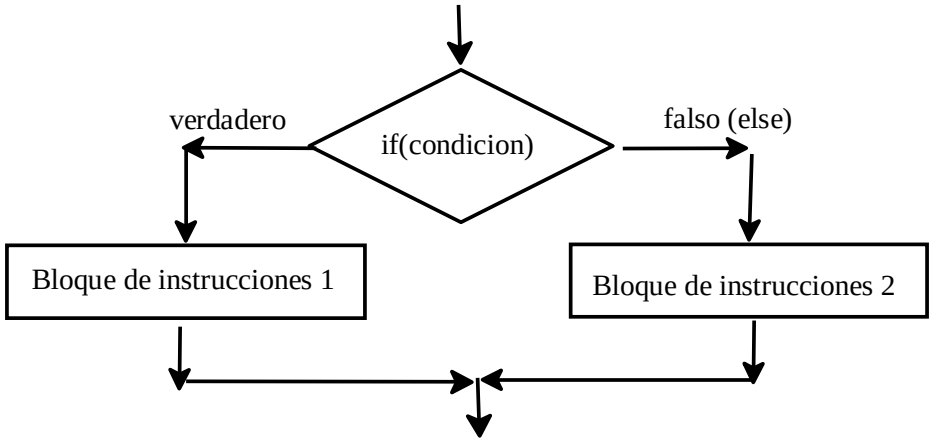


Figura 1.2: Flujo de ejecución de un *if else*

La sintaxis de C++ es:

```
if (condición) {
bloque de instrucciones 1
}
else {
bloque de instrucciones 2
}
```

o

```
if (condición)
instrucción1;
else
instrucción2;
```

Esta instrucción nos dice que si cumple la condición se ejecuta el bloque de instrucciones 1 y si no se cumple se ejecuta la secuencia de instrucciones 2.

Consideremos el ejemplo anterior para hallar la solución de `ax + b = 0`. Utilizando la estructura if else podemos mostrar un mensaje indicando que no existe solución. El código es:

```
#include<iostream>
using namespace std;
int main(){
  int a,b;
  cin>> a >>b ;
  if (a !=0)
  cout<<(float)-b/a <<endl;
  else
  cout<<"No existe una solucion"<<endl;
  return 0;
}
```

4.3.4. Estructura de control if else if

Muchas veces debemos anidar las instrucciones `if`. Esto se puede hacer como sigue:

```
if (condicion1){instrucciones1 if (condicio2){
instrucciones2
}
else {
instrucciones2
}
else {
instrucciones3
}
```

Este forma de anidar se puede repetir las veces que sea necesario sin un límite. Ahora podemos mejorar el programa para resolver `ax + b = 0` considerando el caso en que `a = b = 0` en el cual pondremos un mensaje apropiado. El programa queda como sigue:

```
#include<iostream>
using namespace std;
int main(){
  int a,b;
  cin>> a >>b ;
  if (a !=0)
  cout << (float)-b/a << endl;
  else {
    if (b==0)
    cout <<"No puede ingresar a y b en 0" <<endl;
    else
    cout << "No existe una solución"<<endl;
   }
 return 0;
}
```

4.3.5. Conectores lógicos and, or

Las expresiones lógicas pueden asociarse a través de conectores lógicos `and` y `or`. En C++ éstos conectores representan por && y || respectivamente. Vea que son diferentes a las operaciones con bits que tienen un solo símbolo.

Con estas instrucciones podemos realizar comparaciones más complejas. Por ejemplo
`(a > b)&&(c > d)`. Que significa que debe cumplirse la primera condición y la segunda condición simultáneamente.

La tabla siguiente muestra cómo trabaja el operador and.

Expre.A	Operador	Expre.B	Resultado
true	&&	true	true
true	&&	false	false
false	&&	true	false
false	&&	false	false

La tabla siguiente muestra cómo trabaja el operador or.

Expre.A	Operador	Expre.B	Resultado
true	||	true	true
true	||	false	true
false	||	true	true
false	||	false	false

La tabla siguiente muestra cómo trabaja el operador not.

Operador	Expre.	Resultado
!	true	false
!	false	true

4.3.6. Prioridad de los operadores

Los conectores lógicos se procesan de acuerdo al siguiente orden de prioridades: Primero los paréntesis las instrucciones and y después las or. Veamos un ejemplo: Sea `a = 1, b = 2, c = 1` y deseamos evaluar la expresión `(a > b)&&(b < c) || (a == c)`

a > b	**Oper**.	b < c	**Oper**.	a == c
false	&&	false	||	true

Primero realizamos las operaciones and y tenemos `false&&false` cuyo resultado es `false`.

Continuamos con la operación or y tenemos que `false || true` que da `true`. Para los valores datos esta expresión dará como resultado verdadero.

Si queremos cambiar el orden de evaluación debemos usar los paréntesis quedando `(a > b)&&((b < c) || (a == c))`. En este caso operación or da false || true es true y luego haciendo and con false el resultado es false, que como ve, es diferente al anterior.

4.3.7. Propiedades y equivalencias

En la utilización de los operadores lógicos es necesario conocer las siguientes equivalencias.

Expresión	Equivalencia	Descripción
!(!a)	a	Doble negación
!(a&&b)	!a ||!b	negación de un and
!(a || b)	!a&&!b)	negación de un or

Las equivalencias dos y tres, se denominan leyes de De Morgan. Cuando tenemos una serie de condiciones podemos aplicar estas equivalencias para simplificar las expresiones lógicas.

4.3.8. Estructura de control ?

La estructura de control ? es equivalente a una estructura if else. La sintaxis es: `varibale=condicion ? expresión por verdad : expresión por falso;`

Por ejemplo si deseamos hallar el máximo entre `a y b` y guardar el resultado en la variable m, utilizando if else la codificación es como sigue:

```
if (a>b)
m=a;
else
m=b;
```

Utilizando la estructura ? la sintaxis para hallar el máximo es:

```
m=a>b?a:b;
```

Lo que expresa esta instrucción es que si `a > b` entonces m toma el valor de a en otros casos toma el valor de b.

4.3.9. Estructura de control switch

Para introducir la instrucción switch supongamos que tenemos las notas de 0 – 5, y queremos imprimir el texto reprobado para las notas 1, 2, 3, el texto raspando cuando la nota es 4, y aprobado cuando es 5. Utilizando una secuencia de instrucciones if else se codifica como sigue:

```
#include<iostream>
using namespace std;
int main(){
    int nota;
    cin>> nota;
    if(nota==1)
    cout << "Reprobado"<<endl;
    else
    if (nota==1)
          cout<<"Reprobado"<<endl;
    else
          if(nota==2)
          cout<<"Reprobado"<<endl;
    else
          if (nota==3)
          cout<< "Reprobado"<<endl;
    else
          if (nota==4)
          cout<< "Raspando"<<endl;
    else
          if (nota==5)
            cout<< "Aprobado" <<endl;
    return 0;
  }
```

Este código como ve es muy poco entendible, poco modificable, y propenso a error.

La instrucción switch permite resolver estos problemas. La sintaxis es

```
switch (variable) {
case valor entero : instrucciones; break;
case valor entero : instrucciones; break;
case valor entero : instrucciones; break;
}
```

Esta instrucción trabaja exclusivamente sobre un valor entero. En cada instrucción `case` se compara el valor indicado con el valor de la variable. Si es menor o igual se ingresa a las instrucciones que están después de los dos puntos. La instrucción break hace que la secuencia continué al final del bloque. El ejemplo anterior utilizando la instrucción switch se codifica como sigue:

```
#include<iostream>
using namespace std;
int main(){
    int nota;
    cin>> nota;
    switch (nota){
          case 1:
          case 2:
          case 3:cout<<"Reprobado"<<endl; break;
          case 4:cout<<"Aprobado"<<endl; break;
     }
 return 0;
}
```

Como se ve es una codificación más simple de entender.

4.4. Estructuras de control iterativas

Las instrucciones de iteración, son instrucciones que permiten repetir varias veces un bloque de instrucciones. Se conocen como estructuras de control repetitivas o iterativas. También se conocen como bucles o ciclos. Nos permiten resolver diversos tipos de problemas de forma simple.

Supongamos que deseamos sumar una cantidad de números leídos del teclado. Es posible escribir un programa que lea un número y sume, luego lea otro número sume, etc. Esta estrategia genera los siguientes problemas:

- El código sería muy largo
- Si aumentamos más datos a sumar debemos aumentar las instrucciones
- Es propenso a tener errores
- Si hay que hacer cambios puede ser muy moroso
- Muchas partes del programa van a estar duplicadas.

4.4.1. Ciclo for

El ciclo for tiene la siguiente sintaxis:

```
for (expresión 1; expresión 2; expresión 3) {
bloque de instrucciones
}
```

El proceso de esta instrucción es como sigue:

1. Se ejecutan las instrucciones definidas en expresión 1
2. Se evaluá la expresión 2. Si es verdadera se ejecuta el bloque de instrucciones
3. Se ejecutan las instrucciones de expresión 3 y se vuelve a 2

El diagrama 1.3 muestra un diagrama de como se ejecuta el ciclo for. Veamos algunos ejemplos de utilización. Queremos calcular el factorial de un número `n`, que se define como `n! = (n)(n - 1)(n - 2)...,1`. Para esto hay que construir un bucle con un valor que varía desde 1 hasta n. Dentro del ciclo se realiza la multiplicación.

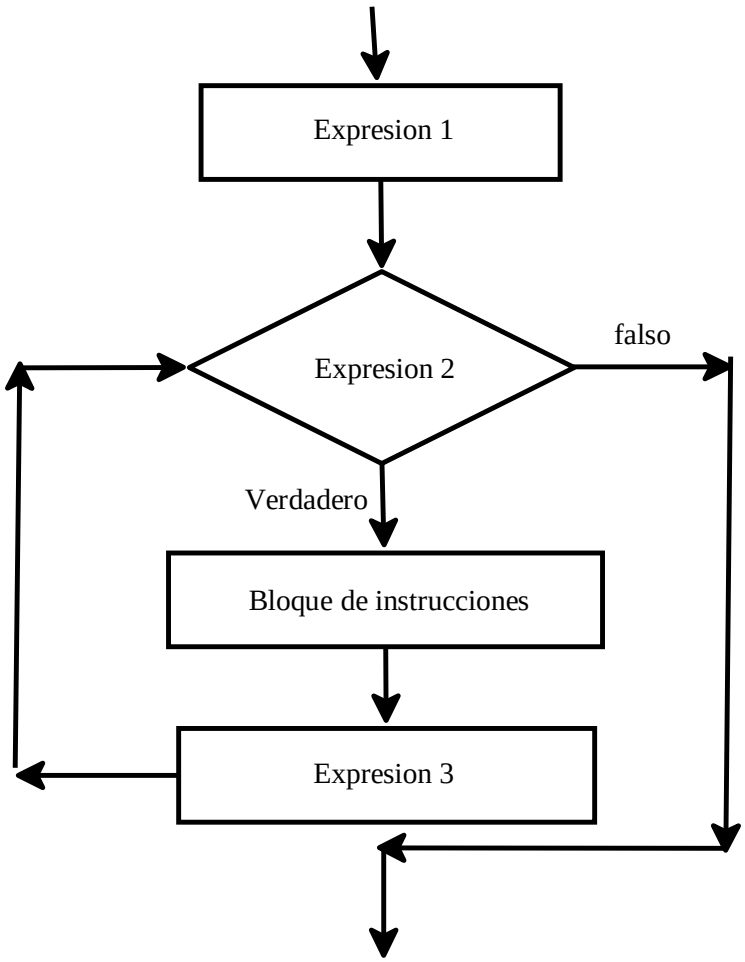


Figura 1.3: Diagrama de la instrucción for

En la expresión 1 crearemos una variable que nos servirá para contar el número de iteraciones, que en este caso es n. Esto hacemos con la instrucción int = 1. Al poner int como tipo para la variable i hacemos que i sea una variable local dentro del bloque de instrucciones.

Luego definimos la expresión 2 que regula la finalización del ciclo. En este caso debemos hacer variar el valor de i hasta que tome el valor de n. La condición para controlar la finalización es i <= n. Mientras que se cumpla esta condición se ejecuta el bloque de instrucciones.

Finalmente viene la expresión 3. Aquí incrementamos el valor de i en uno con la instrucción i + +.

Ahora el código que resuelve el problema. Iniciamos una variable que utilizaremos para hallar el factorial. Esta variable la denominaremos f y la inicializamos en 1. Luego podemos multiplicar por i y así al finalizar el ciclo for f tendrá el factorial de n. El código para calcular el factorial de 5 es:

```
#include<iostream>
using namespace std;
int main(){
 int n=5, f=1;
 for (int i=1;i<=n; i++){
   f=f*i;
 }
 cout << f << endl;
 return 0;
}
```

Hay que ver que la variable f y la variable n fueron definidas fuera del ciclo. Si nos fijamos luego de la instrucción for no se coloca un punto y coma para evitar que la misma termine antes de iniciado el bloque de instrucciones a ejecutar.

Una forma abreviada de escribir puede ser:

```
#include<iostream>
using namespace std;
int main()
{
int f=1;
  for (int i=1,n=5;i<=n;f=f*i,i++);
 cout << f << endl;
 return 0;
}
```

En este código en el primer bloque de instrucciones hemos agregado la definición de la variable n con su valor inicial 5. En el tercer bloque hemos agregado la instrucción que calcula el factorial. Con esto no se requiere definir un bloque, todo queda contenido en la instrucción for. Esta es la razón por la que colocamos un punto y coma al final la instrucción.

Veamos otro ejemplo. Se quiere hallar el resultado de la expresión:

$$s = \sum_{i=0}^{i=10} i^2$$

Para este problema inicializamos una variable s en 0. Luego con un for podemos variar i en todo el rango de la suma. El código resultante es:

```
 #include<iostream>
using namespace std;
int main(){
int s=0;
for(int i = 0;i<=10;i++){
s+=(i*i);
}
cout<< s << endl;
return 0;
}
```

Ahora si conocemos una expresión que nos de directamente el resultado de la suma, el programa es más eficiente. Dado que demora mucho menos tiempo de proceso. En muchas expresiones esto es cierto, en cambio para otras no es posible. En el ejemplo que mostramos es posible:

$$s = \sum_{i=0}^{i=n} i^2 = n(n + 1)(2n + 1)/6$$

4.4.2. Ciclo while

El ciclo while permite ejecutar las instrucciones de un bloque hasta que se cumpla cierta condición. la codificación de esta instrucción es como sigue:

```
while (expresión) {
bloque de instrucciones
}
```

El diagrama 1.4 muestra como se ejecuta el ciclo while. Para asegurar que se llegue a

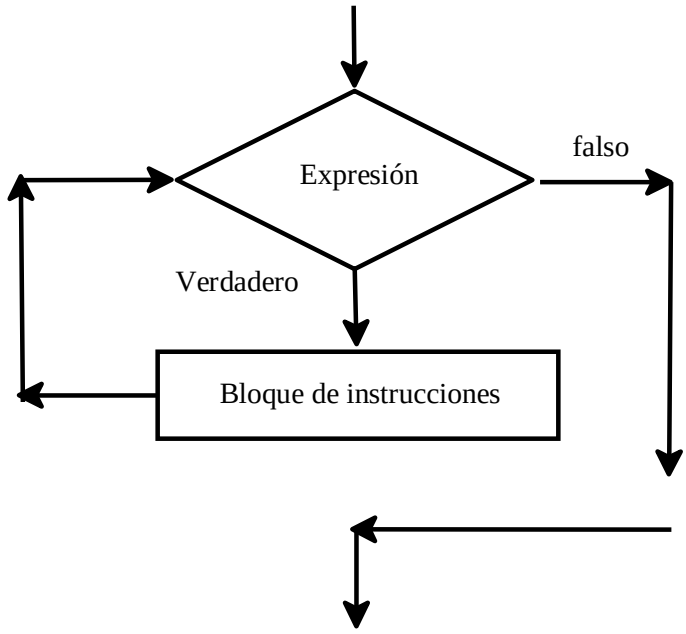


Figura 1.4: Diagrama de la instrucción while

la condición de finalización debe existir algo que haga variar la condición dentro del ciclo.

El ejemplo que realizamos para hallar el factorial también puede realizarse utilizando una instrucción while. el programa siguiente muestra esta implementación.

```
#include<iostream>
using namespace std;
int main(){
int n=5,f=1, i=1;

while(i<=n){
          f=f*i; i++;
     }
 cout << f <<endl;
 return 0;
}
```

La relación que existe entre un ciclo for y while es como sigue:

```
for (expresión 1; expresión 2; expresión 3) {
bloque de instrucciones;
}
```

Equivale a

```
expresión 1;
while(expresión 2) {
bloque de instrucciones;
expresión 3;
}
```

4.4.3. Ciclo do while

El ciclo do while se utiliza para garantizar que el flujo del programa pase una vez por el bloque de instrucciones. La sintaxis es como sigue:

```
do {
bloque de instrucciones
} while (expresión);
```

El diagrama 1.5 muestra como se ejecuta el ciclo do while.

El programa de factorial mostrado puede también resolverse utilizando un ciclo do while.
El programa siguiente muestra la forma de resolver el mismo.

```
#include<iostream>
using namespace std;
int main(){
int n=5,f=1, i=1;
```

```
do {
f=f*i;
i++;
}
while (i<=n);
cout << f << endl;
return 0;
}
```

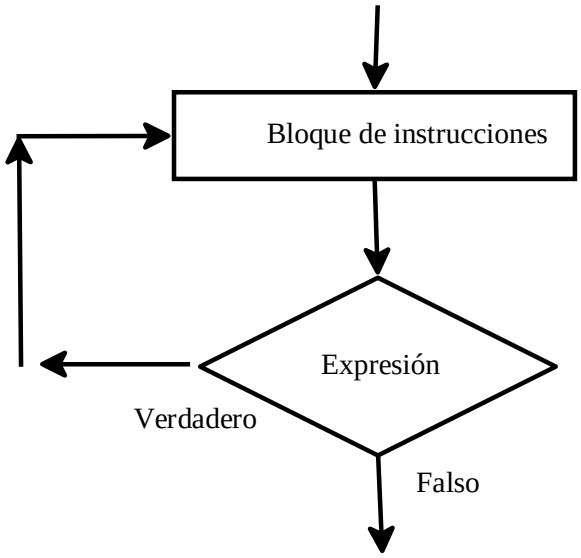


Figura 1.5: Diagrama de la instrucción do while

4.4.4. Ciclos anidados

Los ciclos pueden anidarse, esto podemos escribir un ciclo dentro de otros ciclos. La forma de anidar bucles se muestra en la figura 1.6.

Veamos un ejemplo de anidar dos ciclos for:

```
#include<iostream>
using namespace std;
int main(){
    for(int i=0;i<5;i++){
          for(int j=0;j<5;j++){
                cout << "(" << i <<"," << j << ")";
          }
          cout << " " <<endl;
    }
return 0;
}
```

El resultado que produce es:

```
(0,0) (0,1) (0,2) (0,3) (0,4)
(1,0) (1,1) (1,2) (1,3) (1,4)
(2,0) (2,1) (2,2) (2,3) (2,4)
(3,0) (3,1) (3,2) (3,3) (3,4)
```

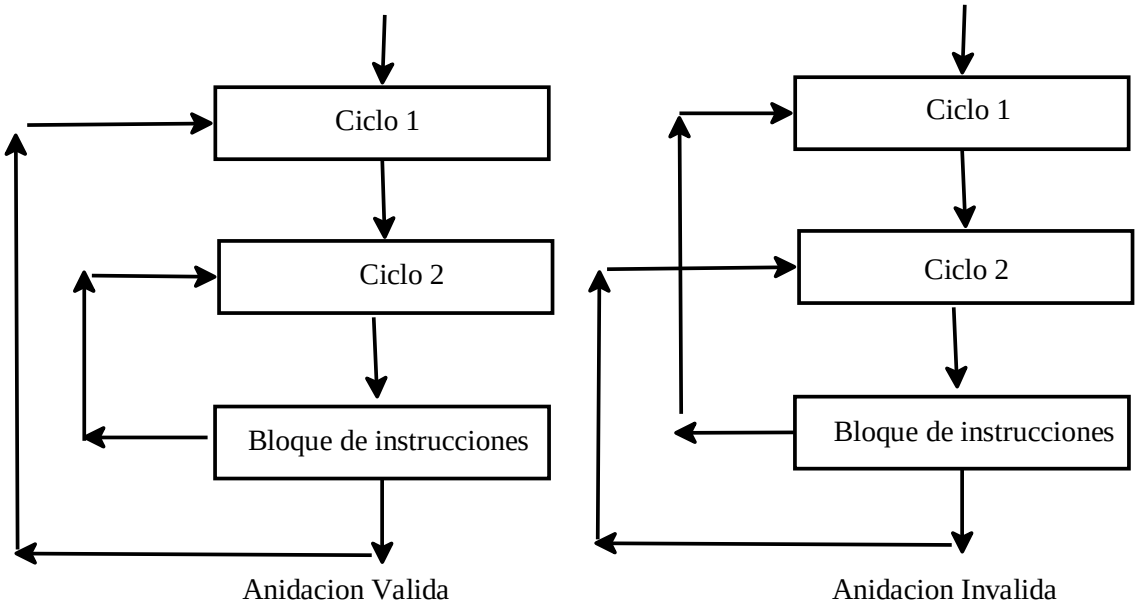


Figura 1.6: Como anidar Ciclos

```
(4,0) (4,1) (4,2) (4,3) (4,4)
```

El ciclo exterior hace variar la variable i de 0 a 4. El ciclo interior varía `0 ≤ j ≤ 4`. La instrucción `cout << "(" << i <<"," << j << ")"` imprime los valores de i, j sin avanzar a la siguiente línea. Cuando termina el bucle interior avanzamos una línea.

Podemos anidar todo tipo de bucles, respetando siempre que un bucle exterior encierra completamente un ciclo interior. No deben existir bucles cruzados como se muestra en la figura 1.6.

4.5. Lectura de secuencias de datos

Generalmente no se lee un solo valor y debemos leer una secuencia de datos. A esto llamamos proceso de lotes. Supongamos que queremos hallar el promedio de una secuencia de números. Para hacer posible ésto ingresaremos primero la cantidad de datos y luego los datos.

```
5
10 5 80 60 40
```

Recordamos que el promedio se calcula con la fórmula $p = \frac{\sum_{i=1}^{i=n} d_i}{n}$

Para leer estos datos y calcular el promedio construimos el siguiente programa:

```
#include<iostream>
using namespace std;
int main(){
    int n;
    cin >> n;
    int s=0, dato;
    for(int i=0;i<n;i++){
          cin>> dato;
          s=s+dato;
    }
    cout << (float)s/n << endl;
    return 0;
}
```

Esto está de acuerdo a la entrada de datos y procesa un solo conjunto de datos. Una posibilidad es que se tengan que ingresar muchas secuencias de las que calcularemos el promedio. Por ejemplo podríamos optar por varias opciones para definir la cantidad de secuencias, Veamos dos posibilidades:

1. Primero especificar la cantidad de casos por ejemplo:

```
2
5
10 5 80 60 40
4
90 20 15 80
```

Hemos anotado en la primera línea la cantidad de secuencias, que son 2, inmediatamente hemos colocado las dos secuencias como en el ejemplo anterior. La solución viene dada por un ciclo `for` como se muestra en el programa:

```
#include<iostream>
using namespace std;
int main(){
    int n;
    cin>> n;
    int s=0, dato;
    for(int i=0;i<n;i++){
          cin >> dato;
          s=s+dato;
    }
    cout <<(float)s/n<<endl;
    return 0;
}
```

2. Una segunda opción es colocar un señuelo al final que indique que hemos llegado al final. Por ejemplo podemos colocar un cero al final para indicar que no hay más secuencias sobre las que queremos

trabajar. Para esto colocamos los datos de prueba como sigue:

```
5
10 5 80 60 40
4
90 20 15 80
0
```

Podemos hacer esto de varias formas: Utilizando la instrucción for, con la instrucción while, y también con do while. Expliquemos primero como aplicar un for para este caso. Nuestra expresión inicial será leer el número de casos n, luego si éste es mayor a cero, no estamos al final y procesamos el bloque de instrucciones. Finalizado el proceso del bloque de instrucciones, leemos otra vez el número de casos y continuamos. El programa siguiente muestra esta lógica.

```
#include<iostream>
using namespace std;
int main(){
    int n;
    for(cin>>n; n>0;cin>>n){
          int s=0, dato;
          for(int j=0;j<n;j++){
          cin >> dato;
          s=s+dato;
    }
    cout <<(float)s/n<<endl;
    }
    return 0;
}
```

Para resolver esto con la instrucción while, primero leemos un valor de n luego, iniciamos un while para verificar que llegamos al final. Cuando termina el bloque de instrucciones leemos el próximo valor de n. Esto nos da el código siguiente:

```
#include<iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    while(n>0){
          int s=0, dato;
          for(int j=0;j<n;j++){
                cin>> dato;
                s=s+dato;
          }
          cout <<(float)s/n<<endl;
          cin >> n;
    }
    return 0;
}
```

Finalmente utilizando la instrucción do while. Primero leemos la cantidad de datos del primer caso de prueba. Procesamos y cuando llegamos al while leemos otra vez para saber si hay más casos de prueba. El código es el siguiente:

```
#include<iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    do {
          int s=0, dato;
          for(int j=0;j<n;j++){
          cin>>dato;
          s=s+dato;
          }
          cout <<(float)s/n<<endl;
          cin>>n;
     }while(n>0);
 return 0;
}
```

A usted le dan un número entero N. El factorial de N se define como N(N-1)(N-2)...1. Calcule el factorial de N, quite todos los ceros de la derecha. Si el resultado tiene más de K dígitos, devuelva los últimos K dígitos, en los otros casos devuelva el resultado completo.

Por ejemplo el número 10 tiene el factorial 10*9*8*7*6*5*4*3*2*1=3628800. Quitamos los ceros de la derecha para obtener 36288 finalmente nos piden 3 dígitos imprimimos 288.

Datos de entrada

Los datos de entrada son los números N y K separados por un espacio donde
1 <= N<= 20 y 1 <= K <= 9. La entrada termina cuando no hay más datos.

Datos de salida

Por cada dato de entrada escriba los dígitos especificados por K en una línea.

Ejemplo de datos de entrada
10 3
6 1
6 3
7 2
20 9
1 1

Salida para los datos de ejemplo
288
2
72
04
200817664
1

```
#include<iostream>
#include<string>
using namespace std;
int main() {
    long long f;
    int i, m, N, K;
    string line;
    getline(cin,line);
    while (cin){
        cin>>N;
        cin>>K;
        f=1;
        for (i = 1; i<= N; i++)
                f=f*i;string s;
        //convierte de long long a string
        long long temp;
        while(f / 10!=0){
            temp=f%10;
            f= f/10;
            temp =temp + 48;
            s = (char)temp + s;
            }
         f=f+48;
         s = (char)f + s ;
         m = s.length ();
         while ( s.at(m - 1) == '0'){
                m = m - 1;
                s = s.substr(0, m);
            }
          if (s.length()<= K)
                cout<<s<<endl;
          else
                cout<<s.substr (s.length()-K, s.length ())<<endl;
    }return 0;
}
```

Notese que se declaro una variable `long long` la cual puede almacenar números más grandes que `int` y `long`, `long` cuenta con 4 bytes y `long long` cuenta con 8 bytes (64 bits) que va del rango de - 9,223,372,036,854,775,808 a 9,223,372,036,854,775,807.

Para sacar los 0`s primero se debe hallar el factorial del numero pedido, convertir esta salida a un tipo de dato string, con el ciclo while se comienzan a buscar los 0`s haciendo una comparación carácter por carácter, se evalúa si la longitud de la cadena resultante es menor igual a K, y continua iterando hasta que no hayan mas datos.

Capítulo 5

Cadenas

5.1. Definición

Se dice cadenas a las secuencias de texto (caracteres) con las que se quiere trabajar. Todos estos textos generalmente están en ascii. Uno puede crear su propia representación, pero, para mostrar por pantalla debe estar en conjunto de caracteres definido en el sistema.

En el español, es ascii, con extensiones para los caracteres latinos y se denomina latin-1 en Windows y iso-8859-15 en Linux y otros sistemas operativos.

En C++ se denominan tipos de datos string y pertenecen a los archivos de encabezado de la clase <string>.

La sintaxis de C++ es:

```
string nombre="cadena";
```

La definición se la comienza con el nombre del tipo de dato que es string, luego el nombre de la variable y el contenido es el conjunto de caracteres encerrado entra comillas. Por ejemplo:

```
string saludo="Hola";
```

5.2. Recorrido de la cadena

Las cadenas se pueden ver como un conjunto de caracteres uno a continuación del otro. El nombre de la cadena representa la dirección de memoria donde se encuentran los caracteres que conforma la cadena. Por esta razón no es posible acceder directamente a sus valores y se requiere utilizar los métodos de la clase <string>.

Consideremos la siguiente definición:

```
string saludo=”Hola”;
string saludo2=saludo;
```

Lo que se hizo primero es definir la cadena saludo que apunta al texto Hola, segundo se asigno a la dirección de la cadena”Hola” al nombre saludo2. Esto significa que solo existe un”Hola”, cualquier cambio en saludo es un cambio en saludo2.

Si comparamos `saludo == saludo2` el resultado es verdadero porque ambos apuntan a la misma cadena.

Si deseamos tener dos cadenas “Hola” se puede definir como sigue:

```
string saludo="Hola";
string saludo2("Hola");
```

Viendolo en el programa:

```
#include<iostream>
#include<string>
using namespace std;
int main(){
    string saludo="Hola";
    string saludo2("Hola");
    return 0;
}
```

Para recorrer una cadena se tiene el método `cadena.at(posición)`. Vemos la cadena"HOLA".

	H	O	L	A
Posición	0	1	2	3

Para mostrar toda una cadena en pantalla tenemos la instrucción que vimos:

```
cout<< cadena;
```

Si queremos mostrar la misma, carácter por carácter, utilizamos at. Por ejemplo para mostrar la cadena saludo recorremos la cadena con un for como sigue:

```
#include<iostream>
#include<string>
using namespace std;
int main(){
string saludo="Hola";
for (int i = 0; i< 4; i++)
   cout<< saludo.at(i)<<endl;
return 0;
}
```

La cual nos muestra en pantalla:

```
H
o
l
a
```

5.3. Métodos de la clase <string>

La clase <string> tiene varios métodos de los que describimos los más utilizados:

Para utilizalos se debe seguir la siguiente sintaxis:

```
cadena1.metodo(cadena2);
```

Método	Descripción	Resultado
at(int)	Devuelve el carácter ubicado en la posición	un carácter
compare(str)	Compara la cadena con str	0 cuando son iguales, negativo si es menor y positivo si es mayor
swap (str)	Intercambia el contenido de dos objetos string	Objetos intercambiados.
find(str)	Busca si existe la cadena str	verdadero o falso
length()	Retorna el tamaño de la cadena	un entero
substr(int)	Obtiene la cadena desde la posición hasta el final de la cadena	cadena
substr (int, int)	Obtiene la cadena desde la posición int hasta la posición int	cadena
clear()	Vacia la cadena.	

Para ejemplificar el uso de los métodos descritos consideremos las siguientes definiciones:

```
string cad1 = "AbCdEf";
string cad2 = "aBcDeF";
string cad3 = "abcabcabc";
```

cout << cad1.at(0) << endl; da como resultado A

cout << cad1.compare(cad2)<<endl; da como resultado −1

cout << cad3.substring(2,5)<<endl; da como resultado cab

cout << cad3.length()<< endl; da como resultado 9

5.4. Lectura del teclado

Utilizando la clase <iostream> se tienen la instrucción `cin` para la lectura por teclado

Veamos algunos ejemplos considerando la entrada como sigue:

```
la casa
de la escuela es
```

El siguiente código

```
string cad1, cad2, cad3;
cin >> cad1 >> cad2 >> cad3;
```

hará que:

```
cad1 = "la";
cad2 ="casa";
cad3 = "de";
```

Si leemos por líneas:

```
string cad1, cad2, cad3;
cin >> cad1;
cin >> cad2;
cin >> cad3;
```

hará que:

```
cad1 = "la";
cad2 ="casa";
cad3 = "de";
```

Una palabra se dice palíndromo si la misma palabra es exactamente idéntica al derecho y al revés como por ejemplo `ABA, AA, A y CASAC`. Por otra parte cualquier secuencia de caracteres puede ser vista como `$1$` o más secuencias de palíndromos concatenadas. Por ejemplo `$guapa$` es la concatenación de los palíndromos `g, u y apa; o casacolorada` es la concatenación `de casac, olo, r, ada.` El problema consiste dado una palabra de a lo máximo 2000 letras minúsculas, imprima el número mínimo de palabras palíndromas en que se puede dividir.

Entrada

La entrada consiste de una línea por caso, cada línea contiene una cadena s de entre 1 y 2000 caracteres. La entrada termina con EOF. Puede estar seguro que la entrada no contiene más de 1000 casos.

Salida

Por cada caso imprime el número mínimo de palabras palíndromas en que se puede dividir *s*.

Ejemplo de datos de entrada
casacolorada

casac

hola

Salida para los datos de ejemplo
4

1

4

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s,line;
         getline(cin,line);
          while(cin){
               cin >>s;
               int l=0, i=0,c=0, j=0, k=s.length();
               while (j<s.length()){
                    string cad = s.substr(j,s.length());
                    for (l=cad.length();l>0;l--){
                         string subcad=cad.substr(0,l);
                         for (i=0;i<subcad.length()/2;i++)
                          if(subcad.at(i)!=subcad.at(subcad.length()-1-i))
                                   break;
                          if (i<subcad.length()/2){
                               continue;
                         }
                         else {
                               c++;
                               j+=l;
                               break;
                         }
                    }
               }
               cout<<c<<endl;
          }
          return 0;
}
```

5.5. Convertir de cadena a entero

Supongamos que hemos leído una cadena con la instrucción cin. Para convertir esta cadena a un número entero utilizamos la instrucción atoi(cadena). El resultado sera un número entero. La tabla siguiente muestra como convertir de cadena a diferentes tipos de datos.

Método	Descripción
atoi (cadena)	Convertir de cadena a tipo int
atof(cadena)	Convertir de cadena a tipo double
atol(cadena)	Convertir de cadena a tipo long int
rand()	Genera numero randomico

Como ejemplo construyamos un programa para convertir una cadena a un numero entero: Por ejemplo 2539, para convertir escribimos el código siguiente:

```
#include<iostream>
#include<cstdlib>
using namespace std;
int main(){
    int i=atoi("2539");
    cout << "La cadena de caracteres\"2539\" convertida a int es " << i<< endl;
    return 0;
}
```

Nótese que se hace uso de la biblioteca de encabezado cstdlib, esta biblioteca contiene los métodos y prototipos de función para la conversión de números a texto, texto a números, asignación de memoria, números aleatorios y varias otras funciones de utilería

5.6. Manejo de excepciones

Las excepciones son en realidad errores durante la ejecución. Si uno de esos errores se produce y no implementamos el manejo de excepciones, el programa sencillamente terminará abruptamente. Es muy probable que si hay ficheros abiertos no se guarde el contenido de los buffers, ni se cierren, además ciertos objetos no serán destruidos, y se producirán fugas de memoria.

En programas pequeños podemos prever las situaciones en que se pueden producir excepciones y evitarlos. Las excepciones más habituales son las de peticiones de memoria fallidas.

Los tipos de la expresión del throw y el especificado en el catch deben coincidir, o bien, el tipo del catch debe ser una clase base de la expresión del throw. La concordancia de tipos es muy estricta, por ejemplo, no se considera como el mismo tipo int que unsigned int.

Si no se encontrase ningún catch adecuado, se abandona el programa, del mismo modo que si se produce una excepción y no hemos hecho ningún tipo de manipulación de excepciones. Los objetos locales no se destruyen, etc.

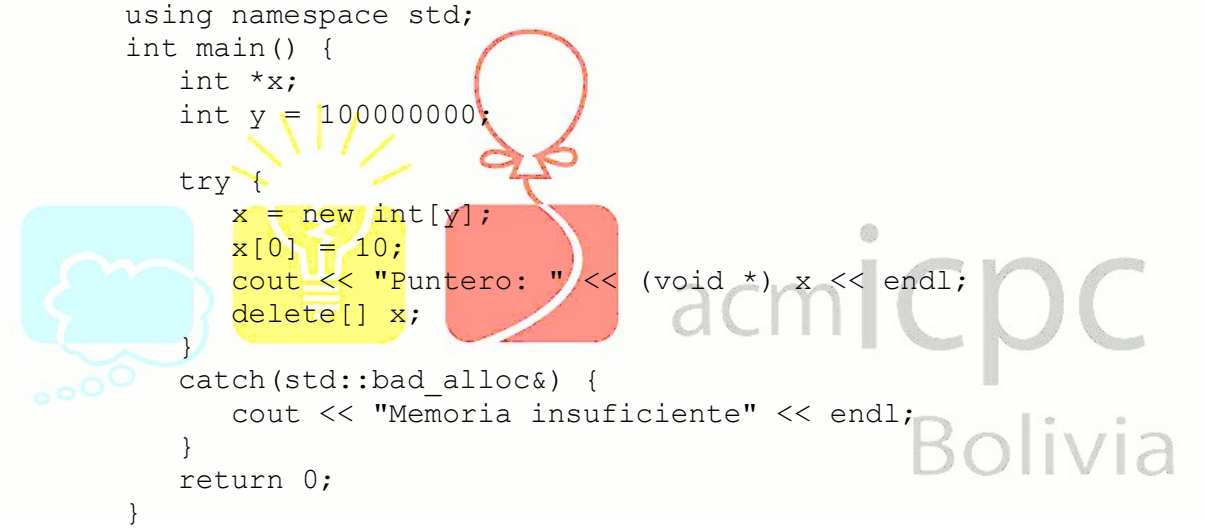
Para evitar eso existe un catch general, que captura cualquier throw para el que no exista un catch concreto:

```
#include <iostream>
using namespace std;
int main() {
        try {
                throw 'x'; //
        }
        catch(int c) {
                cout << "El valor de c es: " << c << endl;
        }
        catch(...) {
                cout << "Excepción imprevista" << endl;
        }
        return 0;
}
```

Por ejemplo tenemos: Veamos este ejemplo, en el que intentamos crear un *array* de cien millones de enteros:

```
#include <iostream>
using namespace std;
int main() {
   int *x;
   int y = 100000000;

   try {
      x = new int[y];
      x[0] = 10;
      cout << "Puntero: " << (void *) x << endl;
      delete[] x;
   }
   catch(std::bad_alloc&) {
      cout << "Memoria insuficiente" << endl;
   }
   return 0;
}
```



En el programa anterior se devuelve el valor de puntero, este programa depende de y si y fuese 1000000000 se devuelve una excepción mediante el mensaje *"memoria insuficiene"*.

Capítulo 6

Arreglos unidimensionales - vectores

6.1. Definición

Un vector o arreglo es una estructura de datos que permite almacenar valores secuencialmente en la memoria de la computadora. Se utiliza como contenedor para almacenar datos, en lugar de definir una variable por cada dato.

Todos los datos deben ser del mismo tipo. No se pueden mezclar tipos de datos. Los valores se colocan en secuencia a partir de la posición cero.

La sintaxis para definir un vector es:

1. Se define el tipo de dato seguido del nombre de la variable.
2. se coloca seguidamente [ ] que indica que se trata de un vector.
3. Finalmente en [ ] se coloca con el tamaño definido.

Veamos algunos ejemplos:

```
int vector [8];
```

Esta definición crea una estructura que define 8 elementos enteros contiguos. El nombre vector es la dirección donde se encuentran los valores de la estructura de datos.

La figura 1.9 muestra como ha sido definido el vector.

Si definimos un vector de enteros, la sintaxis es:

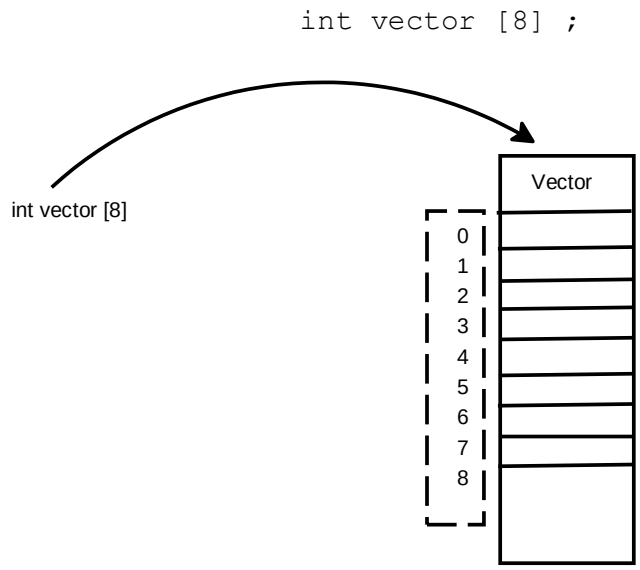


Figura 1.9: Definición de un vector

En este caso, cada elemento del vector es la dirección de la memoria donde se encuentra la cadena.

La figura 1.10 muestra como ha sido definido el vector de cadenas. Para definir un vector

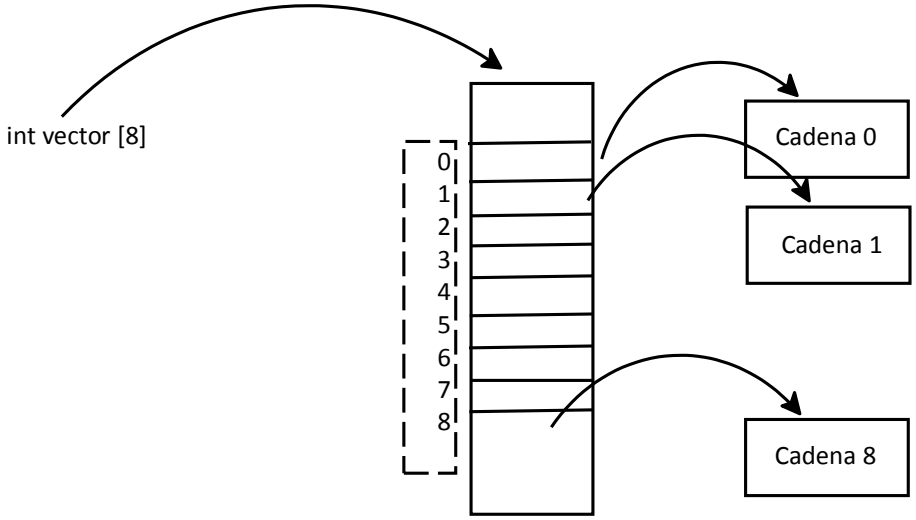


Figura 1.10: Definición de un vector de cadenas

con valores iníciales el procedimiento es como sigue:

1. Se define el tipo de dato seguido del nombre de la variable
2. Se define el nombre de la variable seguido de [dimension] que indica que se trata de un vector.
3. Se coloca el símbolo igual seguido de un {
4. Se colocan todos los valores separados por una coma
5. Se termina con }

Como ejemplo definamos un vector de enteros:

```
int vector [4] = {1,10,5,15};
```

6.2. Recorrido

Cada uno de los elementos de un vector se acceden por la posición donde está almacenado el valor que queremos acceder. Consideremos la definición:

```
int v [4] = {1,10,5,15};
```

Para acceder a cualquier valor se escribe el nombre de la variable seguido de `[ ]` con el número de elemento que queremos acceder. Veamos `v[0] = 1, v[3] = 15.`

El tamaño del vector se puede almacenar en una variable constante o colocar directamente.

Para listar los elementos del vector definido, podemos utilizar un `for`

```
#include <iostream>
using namespace std;
int main(){
    const int len=4;
    int v[len]={1,10,5,15};
    for (int i =0; i<len ;i++)
    cout<<v[i] << endl;
    return 0;
}
```

Los índices del vector van desde 0 hasta dimensión − 1. No es posible acceder a valores fuera de los límites de un vector.

También mediante métodos podemos obtener la dimensión del vector con la instrucción `sizeof(array)`:

```
#include <iostream>
using namespace std;
int main(){
   int array[231];
   int nElementos;
   nElementos = sizeof(array)/sizeof(int);
   cout<<nElementos<<endl;
   return 0;
}
```

Tómese en cuenta que `sizeof(array)` es el numero de bytes que tiene array, y por eso es que se divide entre la cantidad de bytes de el tipo de dato del vector en este caso `sizeof(int)` para asi obtener la dimensión del vector.

6.3. Valores iníciales.

Cuando definimos un vector los valores iníciales que tiene son:

- Si son valores numéricos toman cualquier valor que este en la memoria, por lo que es necesario poner estos en cero antes de utilizarlos.
- Si son referencias como el caso de las cadenas toma el valor null
- Si son valores `bool` toma el valor `false`

6.4. Ejemplos de aplicaciones

Para ejemplificar el uso de vectores vamos a hallar algunos indicadores estadísticos típicos: media, varianza, moda, máximo.

1. Hallar la media. La fórmula de la media es:

$$m = \frac{\sum_{i=1}^{n} x_i}{n}$$

Definiremos un vector para almacenar n valores, el recorrido del mismo será de 0 hasta n − a inclusive. Para este los ejemplos siguientes colocaremos un vector con valores constantes. El código es como sigue:

```
#include <iostream>
using namespace std;
int main() {
    const int le=20;
    int x[le]= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8};
    double m=0.0;
    int suma=0;
    for (int i=0; i<le;i++){
          suma+=x[i];
    }
    m=(double)suma/le;
    cout<< m <<endl;
    return 0;
}
```

2. Hallar la varianza. La varianza se calcula con la fórmula:

$$v^2 = \frac{\sum_{i=1}^{n}(m - x_i)^2}{n}$$

Para resolver éste problema debemos hacer recorrer dos veces el vector una vez para hallar la media m y la segunda para hallar la varianza. El programa es como sigue:

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    const int le=20;
    int x[le]= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8};
    double m=0.0;
    double v=0.0;
    int suma=0;
    for (int i=0; i<le;i++){
          suma+=x[i];
    }
    m=(double)suma/le;
    suma=0;
    cout<<"Media = "<<m<<endl;
    for (int i=0; i<le;i++){
          suma+=(m-x[i])*(m-x[i]);
    }
    v=sqrt(suma)/le;
    cout<< "Varianza = "<< v <<endl;
    return 0;
}
```

3. Hallar el máximo. Para hallar el valor máximo, primero definimos el máximo con el valor más pequeño que se puede almacenar. Luego debemos recorrer el vector y cada vez comparar con el máximo y almacenar el valor mayor. El siguiente programa halla el máximo:

```
#include <iostream>
#include <cmath>
#include <climits>
using namespace std;
int main() {
    const int le=20;
    int x[le]= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8};
    int maxi=INT_MIN;
    for (int i=0; i<le;i++){
          maxi=max(maxi,x[i]);
    }
    cout << "Maximo = "<< maxi<<endl;
    return 0;
}
```

Tomese en cuenta que se hace uso de las librerias: `cmath, climits.` La primera `cmath` contiene prototipos de las funciones de la biblioteca matemática y nos permite usar el método max para hallar el máximo de dos entradas. La segunda `climits` contiene los limites de tamaño de los números enteros del sistema y nos permite usar el método `INT_MIN` que devuelve el numero entero más pequeño del tipo de dato `int.`

4. Hallar la moda. La moda se define como el valor que más se repite, para esto haremos dos procesos. El primero para hallar las frecuencias de repetición de cada uno de los elementos de x. La segunda para hallar el máximo de estas repeticiones.

   Para hallar la frecuencia, es decir, cuantas veces ocurre cada elemento de x definimos un vector `f` donde en `f[0]` servirá para contar cuantas veces aparece el `0, f[1]` para el 1, así sucesivamente. Hemos definido f de tamaño 10 porque sabemos que ningún valor excede a 10 valores. La instrucción `f [x[i]]++` incrementa el vector de frecuencias en cada iteración.

   La segunda parte es la que halla el máximo. Con el mismo algoritmo anterior, almacenando junto al máximo la posición donde se encuentra. La posición representa la moda. En el segundo ciclo el recorrido es sobre el tamaño de f. El programa resultante es:

```
#include <iostream>
#include <cmath>
#include <climits>
using namespace std;
int main() {
    const int le=20;
    const int le2=10;
    int x[le]= {9,4,8,3,7,3,5,2,4,1,2,5,6,1,2,2,4,4,4,8};
    int f[le2];
    for (int i=0; i<le;i++){
          f[x[i]]++;
    }
    int max=INT_MIN;
```

```
    int moda=0;
    for (int i=0; i<le2;i++){
          if(f[i]>max){
                max=f[i];
                moda=i;
          }
    }
    cout<< "Moda = "<<moda<<endl;
    return 0;
}
```

5. Veamos un ejemplo más complejo. Se trata de sumar los puntos de una partida de bolos (bowling). En un juego de bolos un jugador tiene 10 oportunidades de anotar en el transcurso de un juego. Cuando lanza la bola, puede voltear 10 palos si voltea todos. En este caso se contabiliza 10 más lo que haya obtenido en sus siguientes dos jugadas.

   Cuando no voltea todos los palos, se anota la cantidad de palos, luego se hace un segundo lanzamiento y se agrega la cantidad de palos que volteo. Si en ésta segunda oportunidad completa los 10 palos volteados, se agrega lo que obtenga en su siguiente entrada.

Figura 1.11: Juego de bolos

Por ejemplo 5, 2, 3, 4 significa que primero volteo 5 palos, luego 2. En ésta entrada tiene 7 puntos en su segunda entrada tiene 3 + 4 = 7 puntos. Veamos otro caso 10, 5, 2 en este caso en la primera entrada volteo los 10 palos, por lo que el puntaje para esta entrada es 10 + 5 + 2 = 17. En la siguiente entrada tiene 5 + 2 = 7 acumulando hasta este momento 24 puntos. Si los datos son 7, 3, 5, 2, significa que en su primera entrada volteo los 10 palos en dos intentos, por lo que el puntaje que se asigna es 7 + 3 + 5 = 15, y no acumula los últimos 2 puntos. Dada la lista de todos los palos volteados en un juego, calcular el puntaje final.

Expliquemos el programa siguiente:

```
#include <iostream>
#include <cmath>
#include <climits>
using namespace std;
int main() {
    int x [19]= {5,2,10,9,0,8,1,5,5,10,6,4,4,2,9,1,10,10,10};
    //int x[]= {10,10,10,10,10,10,10,10,10,10,10,10,10};
    int  puntosJugada[10];
    //inicializamos con 0 el vector puntosJugada
    for(int p=0;p<10;p++)
          puntosJugada[p]=0;
    int jugadas=0, i=0;
    while (jugadas<10){
                if (x[i]==10){
                      puntosJugada[jugadas]+=x[i+2]+x[i+1]+10;
                      i++;
                     jugadas++;
                }
                else
                  if ((x[i]+x[i+1])<10){
                           puntosJugada[jugadas]+=x[i]+x[i+1];
                            i=i+2;
                            jugadas++;
                  }
                  else {
                      puntosJugada[jugadas]=+x[i+2]+10;
                      i=i+2;
                      jugadas++;
                  }
     }
    int suma=0;
    for (int j=0 ;j<10;j++)
          suma+=puntosJugada[j];
    cout << "Datos iniciales "<<endl;
    for (int j=0 ;j<19;j++)
          cout<< x[j]<< ",";
    cout << "\nPuntos por jugada "<<endl;
    for (int j=0 ;j<10;j++)
          cout<< puntosJugada[j]<< ",";
    cout << "\nPuntaje final "<<suma<<endl;
    return 0;
    }
```

Primero hemos definido un vector donde se establece el puntaje correspondiente a cada jugada, lo denominamos `puntosJugada`, de tamaño 10 dado que solo se puede jugar 10 veces. También establecemos un índice `jugada` que se incrementa en uno por cada jugada.

Se compara primero si derribó 10 palos, en cuyo caso se suma los dos valores siguientes y se incrementa el índice jugada en uno, y el índice que recorre el vector en uno. Luego vemos si hizo 10 en dos lanzamientos, en cuyo caso se incrementa el valor del próximo lanzamiento, y se incrementa el

contador que recorre el vector en dos. En otro caso se incrementa el número de palos derribados. El resultado que se obtiene es:

```
Datos iníciales   [5, 2, 10, 9, 0, 8, 1, 5, 5, 10, 6, 4,
                   4, 2, 9, 1, 10, 10, 10]
Puntos por jugada [7, 19, 9, 9, 20, 20, 14, 6, 20, 30]
Puntaje final 154
```

Capítulo 7

Arreglos multidimensionales

7.1. Definición

Para introducir los arreglos multidimensionales, definamos un vector como sigue:

```
int v[];
```

Al colocar dos corchetes hemos indicado que se trata del arreglo de una dimensión. Si escribimos

```
int v[][];
```

dos veces los corchetes hemos definido un arreglo de dos dimensiones. Los arreglos bidimensionales también se conocen como matrices. Definamos un arreglo bidimensional de 2x3,

```
int v[2][3];
```

Este arreglo tiene 2 filas por 3 columnas. Para definir valores constantes la sintaxis es como sigue:

```
int x[2][3]= {{5,2,10},{9,0,8}};
```

Para acceder a un elemento específico se requieren 2 índices. El de la fila y el de la columna. Así `x[1][0] = 9`, esto es la fila `1` columna `0`.

Para mostrar la matriz tenemos la opción de recorrer con la instrucción `for`

```
for (int i=0; i<dimension;i++)
cout<< x[i]<<endl;
```

Los arreglos de dos dimensiones se denominan matrices. Supongamos que queremos mostrar matriz del ejemplo anterior como sigue:

```
(0,0)=5 (0,1)=2 (0,2)=10
(1,0)=9 (1,1)=0 (1,2)=8
```

El código nos muestra el recorrido del vector y cómo podemos armar ésta salida.

```
#include <iostream>
using namespace std;
int main() {
    int x[2][3]= {{5,2,10},{9,0,8}};
    for (int i=0; i<2;i++){
          for (int j=0; j<3;j++)
                cout << "("<<i<<","<<j<<")="<<x[i][j]<<" ";
          cout << "\n";
    }
}
```

Cada vez que terminamos de mostrar una fila bajamos.

7.2. Ejercicios Resuelto

Escriba un programa que lea un número impar del teclado y genere una matriz cuadrada donde, sin repetir ningún numero, la suma de las filas, columnas y diagonales da el mismo número. Por ejemplo en la matriz siguiente la suma es 15

6	1	8
7	5	3
2	9	4

Para ejemplificar resolvamos el ejercicio denominado cuadrado mágico. El truco para construir un cuadrado de dimensión impar, esto es, el número de elementos de la fila es impar, es comenzar en el medio de la primera fila.

0	1	0
0	0	0
0	0	0

Ahora avanzamos un lugar hacia arriba, en diagonal, para no salirnos del cuadrado hacemos que el movimiento sea cíclico.

0	1	0
0	0	0
0	0	2

Repetimos el movimiento

0	1	0
3	0	0
0	0	2

Ahora no podemos continuar en diagonal, porque la casilla de destino está ocupada, o sea que bajamos un fila.

0	1	0
3	0	0
4	0	2

Continuamos

0	1	6
3	5	0
4	0	2

Luego se baja un lugar

0	1	6
3	5	7
4	0	2

Y termina en:

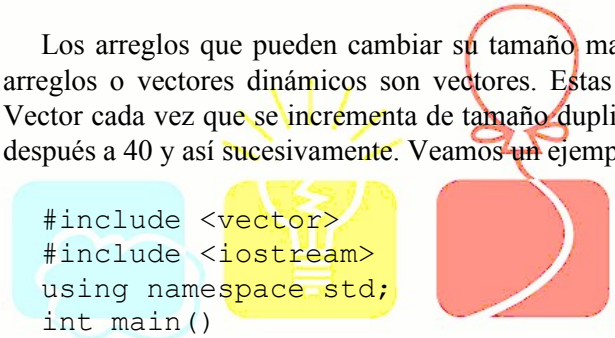
8	1	6
3	5	7
4	9	2

El programa siguiente construye el cuadrado mágico.

```
#include<iostream>
using namespace std;
int main(){
    int n,a=0,i=1,b,m[100][100];
    cout<<"Ingrese dimension para la matriz = ";
    cin>>n;
    b=n/2;//esto es para situar el mumero 1 en el centro
    while(i<=(n*n)){
            if(a<0)a=n-1;//a viene las columnas
            if(b==n)b=0;//b viene a ser las filas
            if(m[a][b]>=1 && m[a][b]<=(n*n)){
                    a=a+2;
                    b--;
                    if (!(a<n))a=a-n;
                          if(b<0)b=n-1;
                                m[a][b]=i;
             }
            else {m[a][b]=i;}
            --a;
            ++b;
            i++;
            }
    // el for ya solo es para mostrar
    //cuando ya está toda la operacion hecha
    for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                    cout<<m[i][j]<<"   ";
            }
            cout<<endl;
    }
    return 0;
}
```

7.3. Arreglos dinámicos

Los arreglos que pueden cambiar su tamaño manteniendo sus valores se denominan dinámicos. Los arreglos o vectores dinámicos son vectores. Estas dos estructuras son similares con la diferencia que Vector cada vez que se incrementa de tamaño duplica su capacidad. Comienza en 10 luego duplica a 20, después a 40 y así sucesivamente. Veamos un ejemplo




```
#include <vector>
#include <iostream>
using namespace std;
int main()
{    //Declaracion simple vector.
     vector <int>enteros;
     int num;
     cin >> num;
     //se agrega un elemento al final del vector
     enteros.push_back(num);
     //se accede a una posicion estatica
    //del vector y se muestra su amplitud
     cout << "En la posicion 0 esta : " << (enteros[0]) << endl
          << "Longitud del vector : "   << (enteros.size()) << endl
          << "Capacidad: " << (enteros.capacity()) << endl
          << "Longitud maxima del vector: " <<
              (enteros.max_size()) << "\n\n";
     //Se limpia el vector, es decir, se vacia
     enteros.clear();
     for (int i=1; i <=10; i++)
          enteros.push_back(i);
     cout <<"Hay en el vector: "<< (enteros.size()) << " elementos\n";
     //Elimino el último elemento del vector
     enteros.pop_back();
     cout <<"Hay en el vector: "<< (enteros.size()) <<
            " elementos\n";
     return 0;
}
```

Entrada : 666

Salida:

El resultado de la ejecución es como sigue:

```
En la posicion 0 esta: 666
Longitud del vector: 1
Capacidad: 1
Longitud maxima del vector: 1073741823
Hay en el vector: 10 elementos
Hay en el vector: 9 elementos
```

Nótese que se está haciendo uso de `#include<vector>`, `vector` es un contenedor de secuencia y es una clase que proporciona una estructura de datos con localidades de memoria contiguas. Esto permite un acceso directo y eficiente a cualquier elemento de un `vector` por medio del operador de subíndice [].

Cuando se acaba la memoria de un `vector`, este asigna automáticamente un area de memoria contigua mas grande, copia los elementos originales a la nueva memoria y libera la memoria anterior.

Los métodos de C++ para la clase `Vector` se describen en el cuadro

Método	Descripción
assing	Asignar elementos al vector
at	Regresa el componente de una posición especifica
back	regresa una referencia a el último componente del vector
begin	regresa un iterator al principio del vector
capacity	regresa el número de elementos que pueden ser contenidos por el vector
clear	remueve todos los componentes del vector
erase	remueve componentes del vector
insert	insertar componentes en el vector
max_size	regresa el número máximo de elementos soportados por el vector
pop_back	remueve el último componente del vector
push_back	agrega un componente al final del vector
size	regresa el número de componentes en el vector
swap	intercambia el contenido de un vector con el de otro

7.4. Arreglos de más de dos dimensiones

Para definir arreglos de cualquier dimensión se ponen más corchetes en c la definición. Por ejemplo para definir un arreglo de tres dimensiones:

```
int v [][][];
```

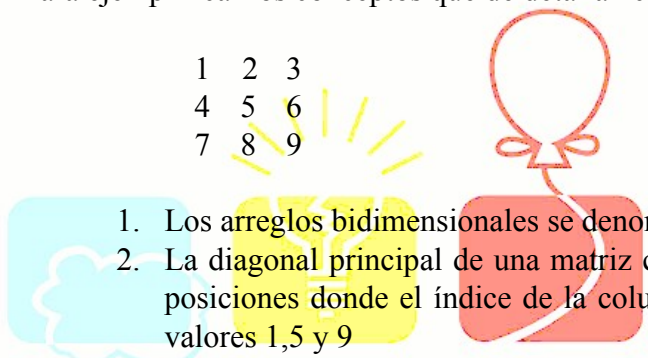
Para acceder a los valores del arreglo requeriremos tres índices. Por ejemplo para definir un cubo con caras de tres por tres, hay que especificar 6 caras y el tamaño de cada cara:

```
int v[6][3][3];
```

Ejercicios

Para ejemplificar los conceptos que de detallan considere la siguiente matriz

1 2 3
4 5 6
7 8 9



1. Los arreglos bidimensionales se denomina matrices
2. La diagonal principal de una matriz comprende los valores que se encuentran en las posiciones donde el índice de la columna iguala al de la fila. En el ejemplo son los valores 1,5 y 9
3. La matriz triangular superior son los elementos que estas por encima de la de la diagonal principal. Si ponemos estos valores en cero en el ejemplo, el resultado seria:
   1 0 0
   4 5 0
   7 8 9
4. Si ponemos ceros a la matriz triangular inferior es:
   1 2 3
   0 5 6
   0 0 9
5. Si mostramos la matriz por filas en la pantalla se obtendrá el siguiente resultado:
   1 2 3
   4 5 6
   7 8 9
6. Si mostramos la matriz por columnas el resultado es el siguiente:
   1 4 7
   2 5 8
   3 6 9
7. La suma de dos matrices C=A + B se hace $c_{ij} = a_{ij} + b_{ij}$
8. La multiplicación de dos matrices C=A*B
   $$c_{ij} = \sum_{i=1}^{k} a_{ik}\, b_{kj}$$
9. La matriz transpuesta es $c_{ij} = a_{ji}$

Para los ejercicios siguientes escriba una programa. Para estos ejercicios defina las matrices en forma constante en la memoria

1. Escriba un programa para llenar de ceros la matriz N xN triangular superior
2. Escriba un programa para llenar de ceros la matriz N xN triangular inferior
3. Escriba un programa para imprimir una matriz N xM por filas
4. Escriba un programa para imprimir una matriz N xM por columnas
5. Escriba un programa para sumar dos matrices de N xM
6. Escriba un programa para multiplicar dos matrices M xN
7. Escriba un programa para hallar la matriz transpuesta de una matriz de M xN .
8. Escriba un programa para generar una matriz caracol de unos y ceros

ÍNDICE