C++

MANUAL TEÓRICO-PRÁCTICO

Alan D. Osorio Rojas

http://slent.iespana.es/programacion/index.html

C++

Manual teórico-práctico

Alan D. Osorio Rojas

Noviembre del 2006

http://slent.iespana.es/programacion/index.html

DEDICATORIAS

A mis amigos,
porque siempre
es necesario superarse.

M. gracias por todo.

ÍNDICE

DEDICATORIAS..3
INTRODUCCIÓN..6
JUSTIFICACIÓN..8
CAPÍTULO 1 INTRODUCCIÓN AL LENGUAJE C++..10
HISTORIA...10
LENGUAJES ASOCIADOS...13
ESTRUCTURA BÁSICA DE UN PROGRAMA...15
CAPITULO 2 TIPOS DE DATOS EN C++..18
TIPOS DE DATOS...22
OPERACIONES BÁSICAS..25
CAPITULO 3 ENTRADA Y SALIDA BÁSICA...31
ENTRADA Y SALIDA..31
MODIFICADORES DE FORMATO...33
OTRAS FUNCIONES MUY ÚTILES..37
CAPITULO 4 ESTRUCTURAS DE CONTROL...41
if...42
? :..45
switch …..46
while..48
do while...51
for..52
CAPÍTULO 5 FUNCIONES...56
PROTOTIPOS..56
PASO DE ARGUMENTOS...58
VALORES DE RETORNO..59
MACROS..59
RECURSIVIDAD...61
CAPÍTULO 6 ARRAYS Y CADENAS...63
DECLARACIÓN...63
ASIGNACIÓN DE VALORES...64
PASO DE ARGUMENTOS...68
ARREGLOS MULTIDIMENSIONALES...71
CAPÍTULO 7 APUNTADORES...75
OPERADORES REFERENCIA-DESREFERENCIA...75
ARITMÉTICA DE OPERADORES..77
ARREGLOS DE APUNTADORES (ARREGLOS DE CADENAS)...................................79
PASO DE ARGUMENTOS POR REFERENCIA...82
CAPÍTULO 8 ESTRUCTURAS..86
ENUMERACIONES..86
UNIONES...88
ESTRUCTURAS..91

CAPÍTULO 9 ENTRADA Y SALIDA POR ARCHIVOS..96
MEMORIA DINÁMICA..96
ARCHIVOS..100
APÉNDICE...108
BIBLIOTECA ESTÁNDAR DE C++...108
RECOMENDACIONES Y/O SUGERENCIAS..113
BIBLIOGRAFÍA...115

INTRODUCCIÓN

El presente trabajo está dividido en 9 capítulos. En cada uno de ellos habrá ejemplos en donde se apliquen los conceptos descritos, es recomendable que se pongan en práctica conforme valla avanzando en la lectura de este documento. Cada ejemplo está explicado casi a detalle, para entender el funcionamiento de cada sección de código.

El capítulo 1 da una introducción al lenguaje C++, contando la historia de este lenguaje y haciendo mención de otros lenguajes de programación con los que está relacionado. Posteriormente presenta una descripción de la forma básica de un programa.

A partir del capítulo 2 entra directamente al aprendizaje del lenguaje, empezamos por definir los tipos de datos que se manejan y después los operadores con los que cuenta el lenguaje. Aunque en los ejemplos escritos en el trabajo no usemos cada uno de estos operadores, si debemos tener en cuenta que podemos utilizarlos en cualquier momento para realizar operaciones complejas de una forma rápida o en cualquier otra circunstancia.

En el capítulo 3 tratamos el tema de la entrada y salida básica. Es sumamente necesario conocer la forma en la que nos comunicamos con el usuario final de nuestro programa. También aprenderemos la forma de presentar mensajes en la pantalla, la forma más eficaz de mantener la comunicación con el usuario.

Las estructuras de control son estudiadas en el capítulo 4, nos servirán para mantener un flujo en nuestro programa, conocemos el uso de los bucles y las tomas de decisiones.

El capítulo 5 trata del uso de las funciones, una manera de hacer que nuestro código sea reutilizable, y una excelente manera de reducir el código a escribir, damos

ejemplos de su uso y describimos los datos y comportamiento de una función. Dentro de este apartado tocamos el tema de la recursividad, y aunque no utilizamos mucho este tipo de funciones, me parece importante el saber de su existencia y forma de operar., en un futuro estoy seguro que les serán muy útiles.

Los arrays y las cadenas son vistos en el capítulo 6, la forma de abordar este tema procuro sea de la forma mas entendible, por medio de esquemas explico el funcionamiento de este tipo de datos que es muy importante para cualquier proyecto. Se proporcionan ejemplos para la mejor comprensión, uno de ellos es un programa para cifrar una frase, proporcionando una contraseña, el programa es muy simple, pero da la pauta para que se tenga una idea de lo que se puede llegar a hacer con un simple código. De las cosas pequeñas se pueden hacer cosas grandes.

Los apuntadores, quizá el tema que más trabajo cuesta entender, es tratado en el capítulo 7. Para facilitar la comprensión de este tema también presento esquemas de comportamiento, un vistazo rápido a la memoria. Explicamos los arreglos de apuntadores y el código de un juego simple, pero que puede ser mejorado bastante por el lector.

El capítulo 8 explica las características y uso de los tipos de dato compuesto como los son las uniones, estructuras y enumeraciones. Los ejemplos presentados en este capítulo son muy interesantes, y son explicados detalladamente, animando al lector a que los complete y/o los mejore para que le sean de mucha más utilidad.

El último capítulo, el 9 trata de la entrada y salida por archivos. Sin embargo, aprovechando el nivel de conocimientos que se han adquirido explica el concepto de memoria dinámica junto con algunas aplicaciones. Y entrando de verdad al tema de archivos, explica los procedimientos a seguir para abrir/crear un archivo, y leer o escribir en él. Presenta un programa muy interesante para mantener una “conversación” con la computadora, el cual invita a ser expandido y mejorado.

JUSTIFICACIÓN

La computación llegó para facilitar el trabajo humano. No es difícil imaginar la gran utilidad que tiene la computación en todas las actividades de la vida moderna del hombre, ya sea como apoyo escolar, en el trabajo, el entretenimiento y la comunicación.

Todo esto se debe a las nuevas prestaciones de la tecnología informática, a las cada vez mejores herramientas de desarrollo, y por supuesto, a los diseñadores y desarrolladores de soluciones software.

Es por eso que el interés de los informáticos hacia el campo de la programación debe crecer, para así desarrollar eficaz y eficientemente programas computacionales que respondan a las necesidades específicas de usuarios finales.

Con esta idea en mi mente, fue como decidí hacer este manual de tal manera que, las personas que lo consulten o lean, se interesen en esta parte de la computación de la misma forma en la que yo. Los ejemplos que contiene este manual, se han hecho para que al lector no le parezca tedioso sino algo que le puede llegar a agradar, los hice pensando en lo que me gusta desarrollar y en lo que muchos podrían ver como divertido.

Con la llegada del software libre, el trabajo de los programadores es más interesante. Las aportaciones de otros desarrolladores hacia un producto propician que éste sea cada vez más eficiente y adaptable a las necesidades del consumidor. Pero para llegar a este punto, es indispensable tener las capacidades suficientes, y esto sólo se logrará incitando cada vez más, a las personas relacionadas con el mundo de la computación, a entrar al área de la programación.

¿Por qué un manual de lenguaje C++ y no otro?, porque es el lenguaje en el cual se han basado para crear otros lenguajes como Java o C#, porque se pueden encontrar muchas similitudes con otros lenguajes y además porque sigue siendo vigente, podríamos señalar que la base del sistema operativo Linux esta creado casi al 100% en éste lenguaje. Aunque una programación avanzada no sea el objetivo de este manual, se tiene presente el incitar a que, después de concluir la lectura de éste, se tengan los conocimientos suficientes y los deseos de investigar y aprender por su cuenta más sobre este u otro lenguaje de programación.

CAPÍTULO 1 INTRODUCCIÓN AL LENGUAJE C++

HISTORIA

La historia del lenguaje de programación C++ comienza a principios de los años 70, con un programador de nombre Dennis Ritchie que trabajaba en los laboratorios de AT&T Bell.

Trabajando con un lenguaje llamado BCPL inventado por Martin Richards (que luego influyó para crear el B de Ken Thompson), Dennis deseaba un lenguaje que le permitiese manejar el hardware de la misma manera que el ensamblador pero con algo de programación estructurada como los lenguajes de alto nivel. Fue entonces que creó el C que primeramente corría en computadoras PDP-7 y PDP-11 con el sistema operativo UNIX. Pero los verdaderos alcances de lo que sería éste, se verían poco tiempo después cuando Dennis volvió a escribir el compilador C de UNIX en el mismo C, y luego Ken Thompson (diseñador del sistema) escribió UNIX completamente en C y ya no en ensamblador.

Al momento de que AT&T cedió (a un precio bastante bajo) el sistema operativo a varias universidades, el auge de C comenzaba. Cuando fueron comerciales las computadoras personales, empezaron a diseñarse varias versiones de compiladores C, éste se convirtió en el lenguaje favorito para crear aplicaciones.

En 1983, el Instituto Americano de Normalización (ANSI) se dio a la tarea de estandarizar el lenguaje C, aunque esta tarea tardó 6 años en completarse, y además con la ayuda de la Organización Internacional de Normalización (ISO), en el año de 1989 definió el C Estándar.

A partir de éste, se dio pie para evolucionar el lenguaje de programación C. Fue en los mismos laboratorios de AT&T Bell, que Bjarnes Stroutstrup diseñó y desarrolló C++ buscando un lenguaje con las opciones de programación orientada a objetos. Ahora el desarrollo del estándar de C++ acaparaba la atención de los diseñadores. En el año 1995, se incluyeron algunas bibliotecas de funciones al lenguaje C. Y con base en ellas, se pudo en 1998 definir el estándar de C++.

Algunas personas podrían pensar que entonces C++ desplazó a C, y en algunos aspectos podría ser cierto, pero también es cierto que algunas soluciones a problemas requieren de la estructura simple de C más que la de C++, C generalmente es usado por comodidad para escribir controladores de dispositivos y para programas de computadoras con recursos limitados.

La base del lenguaje fue creada por programadores y para programadores, a diferencia de otros lenguajes como Basic o Cobol que fueron creados para que los usuarios resolvieran pequeños problemas de sus ordenadores y el segundo para que los no programadores pudiesen entender partes del programa.

C++ es un lenguaje de nivel medio pero no porque sea menos potente que otro, sino porque combina la programación estructurada de los lenguajes de alto nivel con la flexibilidad del ensamblador. La siguiente tabla muestra el lugar del lenguaje respecto a otros.

Alto nivel	Ada Modula-2 Pascal Cobol FORTRAN Basic
Nivel medio	Java C++ C FORTH Macroensamblad or
Nivel bajo	Ensamblador

Tabla 1 Lugar de C++[1]

Ejemplos de lenguajes estructurados:

No estructurados	**Estructurado s**
FORTRAN Basic Cobol	Pascal Ada C++ C Java

En la actualidad los lenguajes que originalmente eran no estructurados han sido modificados para que cumplan con esta característica, tal es el caso de Basic, que actualmente soporta la programación orientada a objetos. Podemos notar cuando un lenguaje de programación es viejo si vemos que no cumple con la programación estructurada. C++ es, también, un lenguaje orientado a objetos, y es el mismo caso de Java. Al referirnos a lenguaje estructurado debemos pensar en funciones, y también a sentencias de control (if, while, etc.)

[1] Herbert Schildt. C Manual de referencia.

Muchos compiladores de C++ están orientados hacia el desarrollo bajo entornos gráficos como Windows 98. Este sistema operativo está escrito casi completamente en C, incluso cuando la compañía Microsoft creó el compilador Visual C++. "Pero deberíamos preguntarnos si esta aparente anomalía refleja una mentalidad inmadura de C++ entre los diseñadores del este sistema y el deseo de la compañía de influir en el código de sistema operativo existente o una relativa idoneidad de los dos lenguajes para escribir sistemas operativos"[2]. En lo personal, pensaría en otra razón popular entre usuarios de Linux.

C++ es un superconjunto de C, cualquier compilador de C++ debe ser capaz de compilar un programa en C. De hecho la mayoría admite tanto código en C como en C++ en un archivo. Por esto, la mayoría de desarrolladores compilan con C++ su código escrito en C, incluso hay quienes, siendo código en C ponen la extensión CPP (extensión de los archivos de código C++) y lo compilan con C++ (hasta hace unos días yo hacía esto), lo cual no es recomendable por norma al programar.

Cuando se compila código C en un compilador C++ este debe cumplir con los estándares definidos en 1989, cualquier palabra definida en el estándar de 1999 no funcionará.

La evolución de C++ continúa, y la diversidad de compiladores contribuye a ello.

LENGUAJES ASOCIADOS

C++ Builder

Creado por una empresa de nombre Inprise, sobradamente conocida por la calidad de sus herramientas de desarrollo, que llevan la firma Borland, entre las que están Borland C++, Intrabuilder, JBuilder, Delphi, C++ Builder.

[2] Al Stevens, Programación con C++.

C++ Builder surgió de la fusión de dos tecnologías: Borland C++ que soporta el lenguaje C++ estándar con todas sus novedades, y el entorno RAD de Delphi.

Visual C++

En el año de 1992, la compañía Microsoft introduce C/C++ 7.0 y la biblioteca de clases MFC, los cuales tenían la finalidad de que el desarrollo de aplicaciones para Windows, escritas en C, fuera más fácil. Sin embargo, no resultó como esperaban, así que un año más tarde fue creado Visual C++ 1.0, que parecía más amigable a los desarrolladores, porque tenía una versión mejorada de las clases MFC.

Con Visual C++ se introdujo una tecnología de desarrollo a base de asistentes. En general es un entorno de desarrollo diseñado para crear aplicaciones gráficas orientadas a objetos. Pero si cree que se trata sólo de arrastrar componentes hacia un “formulario”, como lo puede hacer en Visual Basic, está muy equivocado. La programación en él es más compleja que eso.

C#

Este lenguaje es una evolución de C y C++. Creado por Microsoft y presentado como Visual C# en el conjunto de Visual Studio .NET. Está diseñado para crear una amplia gama de aplicaciones empresariales. La biblioteca para programar en este lenguaje es .NET Framework. El lenguaje es simple, moderno, y está orientado a objetos.

El código de C# se compila como código administrado, lo que significa que se beneficia de los servicios de Common Language Runtime. Estos servicios incluyen la interoperabilidad entre lenguajes, la recolección de elementos no utilizados, mayor seguridad y mejor compatibilidad entre las versiones.

Java

El lenguaje de programación Java se había creado como una plataforma para desarrollo de sistemas de información para Internet y para aparatos electrodomésticos. Fue creado con base en C++. Poco tiempo después de su liberación (en 1995), se empiezan a ver las capacidades de este lenguaje. Pronto deja de ser un lenguaje que sólo se usaba en Internet, y empiezan a crearse programas completos con él.

El lenguaje es orientado a objetos y multiplataforma, esto quiere decir que el código se puede transportar a diferentes sistemas operativos y ejecutarse en ellos por medio de la maquina virtual de Java.

Al igual que con C++ existen varios compiladores para este lenguaje como lo son JBuilder y Visual J++ (ahora evolucionado a J#).

ESTRUCTURA BÁSICA DE UN PROGRAMA

Código	Descripción
`#include <iostream.h>`	}declaración de librerías
`int main(void){`	}función main
`    cout<<"hola mundo"<<endl;`	}secuencia de instrucciones
`    return 0;`	}valor de retorno de la función
`}`	}llaves de cierre de la función

Analicemos cada parte de nuestro primer programa.

```
#include <iostream.h>
```

La parte del #include se refiere a la biblioteca de funciones que vamos a utilizar. Es decir para llamar a una biblioteca en particular debemos hacer lo siguiente:

```
#include <librería_solicitada>
```

El estándar de C++ incluye varias bibliotecas de funciones, y dependiendo del compilador que se esté usando, puede aumentar el número.

```
int main(void){
```

Todo programa en C++ comienza con una función `main()`, y sólo puede haber una. En C++ el `main()` siempre regresa un entero, es por eso se antepone "int" a la palabra "main". Los paréntesis que le siguen contienen lo que se le va a mandar a la función. En este caso se puso la palabra "void" que significa vacío, es decir que a la función main no se le está mandando nada, podría omitirse el void dentro de los paréntesis, el compilador asume que no se enviará nada. La llave que se abre significa que se iniciará un bloque de instrucciones.

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

Esta es una instrucción. La instrucción cout está definida dentro de la biblioteca `iostream.h`, que previamente declaramos que íbamos a utilizar. Una función, en este caso `main()` siempre comienza su ejecución con una instrucción (la que se encuentra en la parte superior), y continúa así hasta que se llegue a la última instrucción (de la parte inferior). Para terminar una instrucción siempre se coloca ";". Pero además de instrucciones se pueden invocar funciones definidas por el usuario (por supuesto diferentes de main) como se verá mas adelante.

```
return 0;
```

Esta es otra instrucción, en este caso la instrucción `return` determina que es lo que se devolverá de la función `main()`. Habíamos declarado que main devolvería un

entero, así que la instrucción return devuelve 0. Lo cual a su vez significa que no han ocurrido errores durante su ejecución.

```
}
```

La llave de cierre de la función `main()` indica el termino del bloque de instrucciones.

En algunos programas de ejemplo, notará el uso de dobles diagonales ("//"). Estas diagonales se usan para escribir comentarios de una línea dentro del código del programa. Además podrá encontrar el uso de "/*" "*/" estos caracteres encierran un comentario de varias líneas y cualquier cosa que se escriba dentro de ella no influenciará en el desempeño del programa.

También verá que muchas veces utilizaremos una diagonal invertida ("\"). Este signo se utiliza cuando una instrucción ocupará varias líneas y por razones de espacio en la hoja es mejor dividirla en partes.

CAPITULO 2 TIPOS DE DATOS EN C++

En la sección anterior vimos la forma general de un programa, un programa sumamente sencillo. Ahora veamos un programa muy parecido al anterior:

```
#include <iostream.h>
int main( ){
     int variable;
     variable=5;
     cout<<variable;
     return 0;
}
```

Notemos en esta ocasión sólo la parte: `int variable;` . A esta sección se le denomina declaración. Se trata de la declaración de una variable de nombre "variable".

Una variable es una posición de memoria con nombre que se usa para mantener un valor que puede ser modificado por el programa[3]. Las variables son declaradas, usadas y liberadas. Una declaración se encuentra ligada a un tipo, a un nombre y a un valor.

Básicamente , la declaración de una variable presenta el siguiente aspecto:

```
tipo nombre [=valor];
```

[3] Herbert Schildt. C Manual de referencia.

Los corchetes significan que esa parte es opcional. Por ejemplo, la declaración:

```
int mi_variable=5;
```

declara una variable tipo entero de nombre “mi_variable” y le asigna el valor “5”.

C++ es sensible a mayúsculas y minúsculas, así que si el nombre de nuestra variable empieza con una letra en mayúsculas, debemos de asegurarnos que durante el resto del código nos refiramos a ella exactamente como la escribimos. Los nombres de las variables no pueden usar signos de puntuación, sólo caracteres “A-Z”, “a-z”, “_“, “0-9”, aunque ningún nombre debe comenzar con un número (0-9). Además no se deben de repetir nombres de variables en el mismo contexto.

Además de las restricciones anteriores, existe otra, y esta tiene que ver con las palabras reservadas del lenguaje, que no son muchas a comparación de otros lenguajes como Basic. Las palabras que se presentan en la siguiente lista, no pueden ocuparse como nombres de variables, funciones, u otras instrucciones definidas por el programador.

and	and_eq	asm	auto
bitand	bitor	bool	break
case	match	char	class
compl.	const	const_cast	continue
default	delete	do	Double
dynamic_cast	else	enum	explicit
export	extern	false	float
for	friend	goto	if
inline	int	long	mutable
namespace	new	not	not_eq
operator	or	or_eq	private
protected	public	register	reinterpret_cast
return	short	signed	sizeof
static	static_cast	struct	switch
template	this	throw	true
try	typedef	typeid	typename

union	unsigned	using	virtual
void	volatile	wchar_t	while
xor	xor_eq		

Tabla 2 Palabras reservadas de C++

Las variables se pueden declarar en tres sitios básicos: dentro de las funciones (ya sea la función main u otras creadas por el programador), estas variables son llamadas locales; en la definición de parámetros de una función, como se verá más adelante; y fuera de todas las funciones, variables globales.

Ejemplo:

```
#include <iostream.h>

int variable_global=10;

int main(){
     int variable_local=20;

    cout<<"\nprograma que muestra los usos de variables "\
          "globales y locales\n"<<endl;
    cout<<"la variable global tiene asignado un: "\
        <<variable_global<<endl;
    cout<<"\nla variable local tiene asignado un: "\
     <<variable_local<<endl;
    return 0;
}
```

Una variable global puede ser modificada en cualquier parte del programa, mientras que una variable local sólo puede ser modificada y utilizada dentro de la función en la que se ha declarado. Por supuesto, antes de utilizar una variable y hacer operaciones con ella, hay que declararla.

Por lo general, siempre se trata de utilizar lo menos posible la declaración de variables globales. El siguiente ejemplo muestra que se pueden declarar variables en cualquier parte del programa, siempre y cuando se declaren antes de usarlas.

```
#include <iostream.h>

int main( ){
     int variable1=10;

     cout<<"la variable 1 local tiene almacenado un: "\
          <<variable1<<endl;
     variable1=50;

     int variable2=variable1+30;

     cout<<"\nla variable 2 almacena un: "\
          <<variable2<<endl;
     return 0;
}
```

En un programa puede que necesitemos declarar un dato y asignarle un nombre, pero que éste no pueda ser modificado. En este caso debemos declarar una constante.

Por ejemplo, el siguiente programa calcula el área de un círculo.

```
#include <iostream.h>
int main( ){
     const float pi=3.141592;
     int radio=5;
     float area;
     area=pi*radio*radio;
     cout<<"el area del circulo es: "<<area<<endl;
     return 0;
}
```

Declaramos una constante del tipo de datos float , le damos el nombre “pi” y le asignamos el valor 3.141592. Este valor jamás podrá ser modificado en ninguna parte del programa

La declaración de una constante presenta el siguiente aspecto:

```
const tipo nombre = valor;
```

TIPOS DE DATOS

Los prototipos de variables utilizados en los programas de ejemplo, hasta el momento, han sido en su mayoría de tipo entero (int), pero es ilógico pensar que éste sea el único que se llegue a utilizar. Además del tipo entero existen otros.

Los tipos de datos atómicos son los tipos de datos más sencillos a partir de los cuales se pueden construir otros más complejos. La siguiente tabla ilustra estos tipos con sus intervalos de valores posibles y el número de bytes que ocupan.

Tipo	Tamaño en bytes	Intervalo
short	2	-32768 a 32767
unsigned short	2	0 a 65535
long	4	-2147483648 a 2147483647
unsigned long	4	0 a 4294967295
int	dependiendo del compilador utilizado podría ser 2 o 4	-32768 a 32767
unsigned int	2 o 4	0 a 65535
float	4	1.17549535e-38 a 3.402823466e+38 con 8 cifras decimales
double	8	2.2250738585072014e-308 a 1.7976931348623158e+308 con 16 cifras decimales

long double	10	
char	1	-128 a 127
unsigned char	1	0 a 255
bool	1	Valor lógico o booleano que puede ser trae (cierto) o false (falso).

Tabla 3 Tipos de datos en C++

En la biblioteca `<limits.h>` encontrará constantes definidas para los intervalos que pueden almacenar algunos tipos de variables. Compile y ejecute el siguiente programa, después consulte la biblioteca `<limits.h>` para ver más constantes definidas.

```
#include <limits.h>
#include <iostream.h>
int main( ){
     cout<<"PROGRAMA QUE MUESTRA LOS VALORES MAXIMOS Y "\
          "MINIMOS \n DE ALGUNOS DE LOS TIPOS DE DATOS "\
          "ATOMICOS"<<endl;
     cout<<"\n int maximo: "<<INT_MAX<<" int minimo: "\
          <<INT_MIN<<endl;
     cout<<"\n char maximo: "<<CHAR_MAX<<" char minimo: "\
          <<CHAR_MIN<<" tamaño en bits: "<<CHAR_BIT<<endl;
     cout<<"\n long maximo: "<<LONG_MAX<<" long minimo: "\
          <<LONG_MIN<<endl;
     cout<<"\n short maximo: "<<SHRT_MAX<<" short minimo: "\
          <<SHRT_MIN<<endl;
     return 0;
}
```

Por medio de los ejemplos dados hasta el momento nos podemos dar una idea de la forma en que se asignan valores a los tipos de datos numéricos. Tan sólo falta mostrar como se pueden asignar valores en formato octal y hexadecimal. Las siguientes instrucciones lo aclararán.

```
int  variable1=022;
int variable2=0x12;
```

Ambas variables están almacenando el número 18, la primera en octal y la otra en hexadecimal. También se pueden asignar números exponenciales.

```
float variable3=2e-6;
```

Pero no sabemos nada sobre los de tipo char.

Para declarar y asignar un carácter a una variable se hace de la siguiente forma.

```
char nombre = '[carácter]';
```

Y si es una constante sólo se agrega la palabra const al principio de la declaración. Note que el carácter asignado está encerrado entre comillas simples. Así, la siguiente declaración asigna la letra "A" mayúscula a una constante de nombre "inicial".

```
const char inicial='A';
```

El valor (carácter) que se almacena en una variable o constante de tipo char, es el valor entero del carácter en el juego de caracteres ASCII. Entonces a la constante "inicial" se le asignó un valor de 65.

Para representar caracteres de control es necesario usar secuencias de escape.

Caracteres de control	**Significado**
\n	salto de línea
\t	tabulador horizontal
\v	tabulador vertical
\a	alerta sonora
\0	carácter nulo
\b	mueve el cursor hacia atrás

Tabla 4 Secuencias de escape

Cualquier carácter se puede representar con la “\” seguida del propio carácter o del código octal o hexadecimal. Por ejemplo las comillas dobles será ‘\”’ o ‘\042’ o ‘\x22’.

OPERACIONES BÁSICAS

En ejemplos anteriores ya hicimos uso de algunos operadores, “+” y “*”, suma y multiplicación respectivamente.

Operador	Significado
+	adición
-	sustracción
*	multiplicación
/	división
%	resto de la división entera

Tabla 5 Operaciones básicas

El siguiente programa ilustrará mejor.

```
#include <iostream.h>
int main( ){
     int a=5, b=10, c=20, r;
     r=a+b;    a=c%r;    //aquí “a” valdrá 5(resto de 20/15)
     c=b-a;    a=a*2;
     cout<<"a="<<a<<" b="<<b<<" c="<<c<<" r="<<r<<endl;
     cout<<"la suma de a y b es: "<<a+b<<endl;
     return 0;
}
```

Estos operadores no son exclusivos de los tipos de datos numéricos. Un dato tipo char también es modificable mediante operadores. Recuerde que lo que se guarda es el código ASCII.

```
#include <iostream.h>
int main( ){
     char car1='a', car2;
     car2=car1+15;
     cout<<"car2="<<car2<<endl;
     car1=car2-10;
     cout<<"car1="<<car1<<endl;
     cout<<"la suma de los dos es:"<<car1+car2<<endl;
     car1=car1-car2;
     cout<<"car1="<<car1<<endl;
     return 0;
}
```

En estos dos últimos ejemplos, note las siguientes instrucciones:

```
a=a*2;
car1=car1-car2;
```

Ambas pueden escribirse de una mejor forma:

```
a*=2;
car1-=car2;
```

Podría decirse que es una abreviatura de código. El operador seguido de la asignación.

Ahora conozcamos a otros operadores muy útiles. “++” y “--“. Éstos tienen la función de aumentar y disminuir en uno al dato que opera. Pero su comportamiento depende de si sea en prefijo (antes del operando) o sufijo (después de).

```
#include <iostream.h>
int main( ){
     int a,b,c;
     a=17*5+2;        //primero resuelve la multiplicación
```

```
    cout<<"a="<<a<<endl;
    b=a++ -7;        //a-7=80, luego se incrementa a
    cout<<"b="<<b<<endl;
    c=++b * --a;     //se incremente b a 81,
                //luego disminuye a a 87, luego evalúa b*a
    cout<<"c="<<c<<endl;
    return 0;
}
```

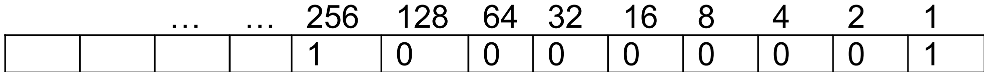
También puede utilizar los paréntesis (como propiedad asociativa) para guiar al programa hacia qué operaciones debe de hacer primero. Hasta aquí, podría considerarse que se han visto las operaciones básicas (suma, resta, multiplicación y división). Pero no son las únicas. Aunque los siguientes operadores no le parezcan importantes o no los comprenda por completo, más adelante se dará cuenta de sus usos.

Operadores de desplazamiento

<<	Desplazamiento de bits a la izquierda
>>	Desplazamiento de bits a la derecha

Tabla 6 Operadores de desplazamiento

Desplazar los bits de un número entero hacia la derecha equivale a dividirlo por 2. Veamos por qué.



		…	…	256	128	64	32	16	8	4	2	1
				1	0	0	0	0	0	0	0	1

Esta es una simple representación de la manera en que está almacenado el número 257 en 2 bytes. Cuando aplicamos el desplazamiento de un bit a la derecha ocasionará lo siguiente.

		…	…	256	128	64	32	16	8	4	2	1
				0	1	0	0	0	0	0	0	0

Desplaza un bit a la derecha y el campo vacío lo llena con un cero. Vea el siguiente ejemplo de aplicación.

```
int var=16;
var >>= 1;          //ahora var tiene asignado un 8
```

Estos operadores también admiten la asignación de la forma “<<=” y “>>=”.No debe confundir el uso de estos operadores con los de inserción de flujo, de las instrucciones cin y cout (que se explicarán más adelante).

Operadores relacionales y de igualdad

Son operadores binarios que permiten comparar los valores de dos variables o expresiones.

Operador	Significado
<	menor que
<=	menor o igual
>	mayor que
>=	mayor o igual
= =	igual
!=	desigual

Tabla 7 operadores relacionales y de igualdad

Estos operadores devuelven 0 si la comparación es falsa o 1 si es verdadera.

Operadores de bits y lógicos

Operador	Significado
&	conjunción (Y) de bits
^	O exclusivo de bits
|	O inclusivo de bits

Tabla 8 Operadores de bits

Estos operadores también pueden operar con asignación (& =, | =, ^ =).

Operador	Significado
&&	conjunción (Y) lógica
| |	disyunción (O) lógica

Tabla 9 operadores lógicos

Precedencia de operadores

Símbolo	Significado
::	resolución de alcance
++ -- ( ) [ ] -> .	incremento sufijo decremento sufijo llamada a función elemento de tabla acceso a miembro desde un puntero acceso a miembro
++ -- ! ~ - + & * sizeof new delete (tipo)	incremento prefijo decremento prefijo negación lógica complemento a uno cambio de signo (operador unario) identidad (operador unario) dirección seguir un puntero tamaño en bytes reserva de memoria liberación de memoria conversión explícita de tipos

.* ->*	acceso a miembros de clase acceso a miembros de clase desde puntero
* / %	multiplicación división resto de la división entera
+ -	suma resta
<< >>	desplazamiento de bits a la izquierda desplazamiento de bits a la derecha
< <= > >=	menor que menor o igual mayor que mayor o igual
= = ! =	igual desigual
&	conjunción (Y) de bits
^	O exclusivo de bits
|	O inclusivo de bits
&&	conjunción (Y) lógica
| |	disyunción (O) lógica
= *=, /=, %=, +=, - =, <<=, >>=,&=, ^=, | =	asignación simple asignaciones compuestas
?:	expresión condicional
trhow	lanzamiento de excepción
,	separador de expresiones

Tabla 10 Precedencia de operadores en C++

CAPITULO 3 ENTRADA Y SALIDA BÁSICA

ENTRADA Y SALIDA

En los programas hechos hasta el momento, hemos utilizado la instrucción `cout<<` para mandar mensajes a la pantalla.

La mayoría de los programas en C++ incluyen el archivo de encabezado `<iostream.h>`, el cual contiene la información básica requerida para todas las operaciones de entrada y salida (E/S) de flujo.

Cuando usamos la instrucción:

```
cout<<"Mensaje a la pantalla"<<endl;
```

Estamos enviando una cadena de caracteres ("Mensaje a la pantalla") al dispositivo de salida estándar (la pantalla). Luego, el manipulador de flujo `endl` da el efecto de la secuencia de escape '\n'.

Pruebe el siguiente programa:

```
#include <iostream.h>

int main(){
     cout<<"cadena de caracteres"<<endl;
     cout<<2+2<<endl;         //imprime un entero
     cout<<9/2<<endl;         //imprime un flotante
     cout<<(int)(3.141592+2)<<endl;    //imprime un entero
```

```
     return 0;
}
```

La instrucción `cout<<` puede imprimir tanto números enteros como flotantes sin necesidad de decirle específicamente el tipo de datos del que se trata, pero, por supuesto notemos que al enviarle una cadena de caracteres esta debe de estar entre comillas. Ya en ejemplos anteriores vimos como se mandaba a la pantalla el valor de una variable, así que no hace falta más ilustración al respecto.

La interacción con el usuario es algo muy importante en la programación, imaginemos que en este preciso momento y con los conocimientos que tenemos hasta ahora, necesitamos hacer un programa que calcule la distancia a la que caerá un proyectil lanzado a determinada velocidad y ángulo, o simplemente un programa que calcule las raíces de una ecuación cuadrática. Sería muy molesto estar cambiando los valores de las variables directamente en el código para cada caso que queramos calcular. Por eso debemos ver cuanto antes la forma de leer datos desde el teclado.

La principal función para leer desde el teclado es `cin>>`, pero es mejor ver un ejemplo para tener la idea clara.

```
#include <iostream.h>
int main(){
     int numero;
     char car;
     float otroNum;
     cout<<"escribe un numero:"<<endl;
     cin>>numero;
     cout<<"\nel numero que tecleaste es: "<<numero<<endl;
     cout<<"dame una letra"<<endl;
     cin>>car;
     cout<<"\ntecleaste: "<<car<<endl;
     cout<<"escribe un numero flotante"<<endl;
     cin>>otroNum;
     cout<<"\neste es: "<<otroNum;
```

```
	return 0;
}
```

En resumen, `cin` es el flujo de entrada asociado al teclado, `cout` es el flujo de salida estándar asociado a la pantalla, y existe otro, que aunque a lo largo de este trabajo casi no lo utilicemos, es necesario nombrarlo, `cerr`, que es el flujo de error estándar asociado a la pantalla.

Los operadores << y >> son operadores de inserción y extracción de flujo respectivamente, y no deben confundirse con los de desplazamiento de bits. Estos operadores son muy eficaces porque no es necesario especificar formatos para presentación o lectura, ellos los presentan en función al tipo de datos de la variable. Aunque en ocasiones podría necesitar de nuestra ayuda para obtener los resultados específicos que queremos, y para eso están los modificadores de formato.

MODIFICADORES DE FORMATO

En el primer ejemplo del presente capítulo quizá no entendió del todo lo qué hacía la instrucción:

```
cout<<(int)(3.141592+2)<<endl;
```

Si probó el código y lo ejecutó seguramente esperaba que apareciese en la pantalla 5.141592, esto habría pasado si no hubiésemos hecho una conversión de tipo. La inclusión de “(int)” dentro de la instrucción provocó que el resultado de la operación, que debía ser 5.141592, se transformara a un entero y por resultado perdiera sus valores decimales.

A esto se le llama hacer un cast, y es muy común hacerlo cuando no queremos que se pierdan determinadas propiedades en los resultados de las operaciones. Por ejemplo, las siguientes líneas de código, que son equivalentes tienen por efecto convertir el tipo de la variable de origen al de la variable destino:

```
int B;
double A= (double) B;
```

```
int B;
double A = double(B);
```

Otra de las herramientas para la especificación de formatos son los manipuladores de flujos, así como el manipulador de flujo "`endl`" da una secuencia de escape, los manipuladores `dec`, `oct` y `hex`, hacen que la variable a la que le anteceden sea presentada en formato decimal, octal o hexadecimal respectivamente.

```
#include <iostream.h>
int main(){
     int numero=25;
     int leido;
     cout<<"numero es en octal: "<<oct<<numero<<endl;
     cout<<"en hexadecimal: "<<hex<<numero<<endl;
     cout<<"en decimal: "<<dec<<numero<<endl;
     cout<<"ahora teclea un numero"<<endl;
     cin>>hex>>leido;
     cout<<"el leido vale: "<<hex<<leido<<endl;
     cout<<"y en decimal: "<<dec<<leido<<endl;
     cout<<"y en octal: "<<oct<<leido<<endl;
     return 0;
}
```

Estos manipuladores de flujo no son exclusivos de la salida, también funcionan con la entrada de datos, por defecto, la entrada desde el teclado se lee en decimal y también la presentación de datos se hace en decimal, así que si se sabe que el usuario dará a la máquina un dato en hexadecimal, sería buena idea anticiparlo y poner el formato en que será leído.

En cuanto a la precisión, al imprimir en pantalla un número de tipo flotante, no importa cuantas cifras significativas tenga (mientras esté dentro de los límites de almacenamiento de flotantes), sólo aparecerán 6 números después del punto, y la ultima cifra será redondeada en caso de pasar los 6 dígitos. Es decir, que si queremos mandar a la pantalla el número 1.23456789, en su lugar aparecerá 1.234568.

Cuando se quieren resultados más exactos, nos es muy útil la librería `iomanip.h` que contiene el manipulador `setprecision()` con el que podemos determinar la precisión con la que queremos que aparezcan esos datos en la pantalla. Existe también la función miembro `pecision()`, que puede resultar un poco más cómoda en algunos casos porque el efecto de precisión funciona para todas las instrucciones `cout` que le sigan. A ambas se le pasa como parámetro el número de cifras significativas deseadas. Veamos un ejemplo.

```
#include <iostream.h>
#include <iomanip.h>
int main(){
     cout<<1.23456789<<endl;
     cout.precision(4);
     cout<<1.23456789<<endl;
     cout<<setprecision(5)<<1.23456789<<endl;
     cout.precision(7);
     cout<<1.23456789<<endl;
     cout<<setprecision(9)<<1.23456789<<endl;
     return 0;
}
```

En ocasiones es útil dar una presentación a nuestros datos en la pantalla, tan sólo el dejar unos cuantos espacios de margen en la pantalla puede ser un poco tedioso para quienes no están enterados de las siguientes funciones:

`setw` que se encarga de hacer aparecer el texto alineado determinados espacios a la derecha, si el numero de espacios solicitados para su alineación es menor que el numero de caracteres que se imprimirán, no tendrá un efecto.

`setfill` se ocupa del carácter de relleno, el carácter por defecto es el espacio en blanco, pero nosotros podemos especificar el que queramos.

Vea el siguiente ejemplo.

```
#include <iostream.h>
#include <iomanip.h>
int main(){
     cout<<setw(5)<<setfill('%')<<"hola"<<endl;
     cout<<setw(5)<<89;
     cout<<setw(8)<<"centavo"<<endl;
     return 0;
}
```

En la primera instrucción `cout`, hacemos aparecer el texto alineado 5 espacios y en esos espacios aparecerá el carácter '%'. Es necesario el manipulador `setw` para cada operación de salida en pantalla, mientras que `setfill` funciona para todas las que se incluyan. Estas funciones, junto con las de alineación son una gran herramienta para la presentación. Las funciones de alineación se incluyen como sigue, siendo la predeterminada la alineación a la derecha.

```
cout.setf(ios::right, ios::adjustfield);
cout.setf(ios::right, ios::adjustield);
```

OTRAS FUNCIONES MUY ÚTILES

La función de lectura desde el teclado `cin` no es la única que existe. Hay otros métodos de lectura desde el teclado para la lectura de caracteres, la siguiente tabla resumirá sus principales características.

Función	Sin argumentos	Un argumento	2 argumentos	3 argumentos
get	Lee un carácter y devuelve su valor numérico	Lee un carácter y lo guarda en la variable de argumento	Lee una serie de caracteres y los guarda en el primer argumento, se detiene si llega a la cantidad máxima de almacenamiento enviada como segundo argumento	Lee una serie de caracteres y los almacena en el primer argumento hasta llegar a la cantidad máxima de almacenamiento (al igual que la de dos argumentos), pero también se detiene si encuentra el carácter especificado en el tercer argumento.
getline	* * * * * * * *	* * * * * * * *	Funciona igual que el get de dos argumentos, pero no guarda el delimitador de flujo.	* * * * * * * *
read	* * * * * * * *	* * * * * * * *	Almacena en el	* * * * * * * *

			primer argumento un número determinado de caracteres	
write	* * * * * * * *	* * * * * * * *	Muestra, de la cadena enviada como primer argumento, determinado numero de caracteres (segundo argumento)	
ignore	Ignora un carácter del flujo (opción predeterminada)	Ignora el numero de caracteres enviados como argumento	* * * * * * * *	* * * * * * * *
gcount	Devuelve el número de caracteres leídos en la ultima lectura sin formato	* * * * * * * *	* * * * * * * *	* * * * * * * *

Tabla 11 Funciones de lectura sin formato

Ahora veamos un ejemplo.

```
#include <iostream.h>
int main(){
     char caracter,nombre[20], apellido[20];
     char seg_apellido[20];

     cout<<"Teclea tu nombre por favor"<<endl;
     cin.get(caracter);
```

```
    cin.get(nombre,20,' ');
    cin.get();
    cin.get(apellido,20,' ');
    cin.get();
    cin.get(seg_apellido,20);
    cout<<caracter<<"Tu letra inicial es:"<<caracter<<endl;
    cout<<"tu nombre es: "<<caracter<<nombre<<endl;
    cout<<"tu apellido paterno es: "<<apellido<<endl;
    cout<<"tu apellido materno es: "<<seg_apellido<<endl;
    cout<<"Escribe el nombre de tu empresa: "<<endl;
    cin.ignore();
    cin.getline(nombre,20);
    cout<<"el nombre de tu empresa: "<<nombre<<" tiene "
    <<cin.gcount()<<" caracteres"<<endl;
    cin.get();     //espera que oprimas enter para terminar
    return 0;
}
```

Puede que esta no sea la mejor manera para hacer un programa de este tipo, pero para nuestro objetivo creo que esta bien. Analicemos.

Primero declaramos una variable char, y 3 variables de cadenas de caracteres, por el momento no profundizaremos en que es exactamente, pero podemos decir que es una “tira” o serie de caracteres (`char`) con 20 elementos.

Pide al usuario que teclee su nombre completo, para luego leer inmediatamente el primer carácter que teclea y lo almacena en la variable “caracter”. Sigue leyendo y almacena lo que escribe en la variable “nombre” hasta que encuentre un espacio o llegue a 20 caracteres. Luego, otra instrucción get se encarga de leer y desechar el carácter espacio, para nuevamente leer hasta encontrar otro espacio.

Deja de leer hasta que se presione enter, hayas escrito o no tus dos apellidos. Así que cuando oprimes enter, “carecer” tiene tu letra inicial y la presenta en pantalla, para aparecer tu nombre imprime “caracter” seguido de “nombre”, porque a ésta

última no almacenó el primer carácter. Las siguientes instrucciones son parecidas hasta que pide el nombre de tu empresa.

La función ignore descarta del flujo el carácter de terminación, para luego permitir a `getline()` leer el nombre de tu empresa con un máximo de 20 caracteres o hasta que oprimas enter (pero no lo guarda). Se escribe el nombre y con `gcount()` presenta los caracteres leídos. Finalmente espera que oprimas enter para terminar el programa.

CAPITULO 4 ESTRUCTURAS DE CONTROL

En nuestra vida cotidiana, todos tenemos una lógica a seguir, continuamente tomamos decisiones, y estas decisiones repercuten en nuestra acción siguiente. Por ejemplo, supongamos el caso de un estudiante de nivel preparatoria que cursa el sexto semestre, él está pensando en presentar el examen para la universidad, sin embargo, sus calificaciones no le han dado mucho aliento últimamente, y está en riesgo de tener que repetir ese semestre, si ocurre eso, el resultado que tenga en el examen no importará. Lo que valla a pasar marcará el camino a seguir en su vida.

Analicemos de una forma general este caso:

curso el sexto semestre
presento el examen de admisión
si paso el examen y además paso el semestre
    estaré en la universidad
si paso el semestre pero no paso el examen
    estaré trabajando
si no paso el semestre
    curso el sexto semestre (es decir, regresa al principio)

Estas son las opciones que él se plantea, aunque algunos pensarán en otras más. Puede estudiar, trabajar o repetir todos los pasos anteriores.

En la programación también se tienen que tomar decisiones que influirán en el comportamiento del programa, y también se pueden repetir series de pasos hasta obtener un resultado. En el presente capítulo aprenderemos eso.

if

Se trata de una estructura de selección. Es decir que si se cumple el condicional se ejecutarán varias instrucciones más. En el ejemplo anterior, observamos que :

si paso el examen y además paso el semestre

    estaré en la universidad

“si” podemos interpretarlo como el if de código c++
“paso el examen y además paso el semestre” es la condición
“estaré en la universidad” es la acción a ejecutar.

Escribamos el código:

```
#include <iostream.h>

int main(){
     char examen[2],semestre[2];
     cout<<"Contesta si o no a las preguntas"<<endl;
     cout<<"Pasaste el examen?"<<endl;
      cin.get(examen,2);
      cin.ignore(20,'\n');
      cout<<"Pasaste el semestre?"<<endl;
     cin.get(semestre,2);
      cin.ignore(20,'\n');           //ignora el enter
     if((examen[0]=='s')&&(semestre[0]=='s')){
          cout<<"estas en la universidad"<<endl;
     }
     cout<<"fin del programa"<<endl;
     cin.get();     //espera a que oprimas enter
     return 0;
}
```

Analicemos lo que nos importa, la sentencia if.

La condición a cumplir es: (examen[0]= =’s’)&&(semestre[0]= =’s’), en realidad se trata de dos condiciones que se deben de cumplir: que la variable examen tenga el carácter ‘s’ y que la variable semestre tenga el carácter ‘s’ (vease la tabla 7 operadores relacionales y de igualdad). Si la primera condición se cumple, entonces comprueba la segunda condición (gracias a la conjunción lógica &&).

Un condicional regresa 1 en caso de ser verdadero y un 0 en caso de ser falso, esto dos valores, en c++ están definidos como tipos de datos booleanos, `true` y `false` respectivamente, así que si el condicional resulta true (verdadero), se evalúa la siguiente instrucción dentro del bloque if, en caso contrario no hace nada.

En el ejemplo anterior el bloque de instrucciones a ejecutar en caso de ser verdadera la condición se encuentra dentro de llaves, pero no es necesario cuando sólo se trata de una sola línea de código.

Cuando queremos, además controlar las opciones en caso de que la condición resulte falsa, tenemos que incluir aquellas instrucciones en un bloque `else`.

Por ejemplo, en el código anterior podemos añadir un `else` a la sentencia `if`, de manera que nos quede de la siguiente forma:

```
if((examen[0]=='s')&&(semestre[0]=='s')){
          cout<<"estas en la universidad"<<endl;
}
else{
          cout<<"no estas en la universidad"<<endl;
}
```

Igualmente el bloque de instrucciones a ejecutar se estructura dentro de llaves, aunque tampoco es necesario cuando se trata de una sola instrucción.

En ocasiones necesitaremos construir bloques de instrucciones if y else más complejos, es decir anidamientos. El ejemplo del estudiante también nos ayudará a visualizar este concepto.

```
si paso el examen y además paso el semestre
        estaré en la universidad
si paso el semestre pero no paso el examen
        estaré trabajando
si no paso el semestre
        curso el sexto semestre
```

Construyendo mejor esta serie de decisiones, podemos acomodarla de la siguiente forma, al estilo seudocódigo.

```
if paso el examen y además paso el semestre
        estaré en la universidad
else
        if paso el semestre pero no paso el examen
                estaré trabajando
        else
                curso el sexto semestre
```

De esta manera controlamos las posibles respuestas que pudiese dar. En caso de que pase el examen y el semestre estará en la universidad, si pasa el semestre pero no pasa el examen estará trabajando, y en cualquier otro caso, cursará el semestre.

Nuestro código ya terminado quedará de la siguiente manera:

```
#include <iostream.h>

int main(){
     char examen[2],semestre[2];
     cout<<"Contesta si o no a las preguntas"<<endl;
     cout<<"Pasaste el examen?"<<endl;
      cin.get(examen,2);
      cin.ignore(20,'\n');
      cout<<"Pasaste el semestre?"<<endl;
     cin.get(semestre,2);
      cin.ignore(20,'\n');           //ignora el enter
     if((examen[0]=='s')&&(semestre[0]=='s'))
          cout<<"estas en la universidad"<<endl;
     else
          if((examen[0]=='n')&&(semestre[0]=='s'))
               cout<<"estaras trabajando"<<endl;
     else
          cout<<"cursa el sexto semestre"<<endl;
     cout<<"fin del programa"<<endl;
     cin.get();     //espera a que oprimas enter
     return 0;
}
```

En esta ocasión se eliminaron las llaves que no eran necesarias.

? :

Este se trata de un operador del tipo condicional al estilo if/else. Se utiliza para condicionales cortos (de una sola línea).

Por ejemplo:

```
#include <iostream.h>
int main(){
    int calificacion;
    cout<<"Cual fue tu calificacion del semestre?"<<endl;
    cin>>calificacion;
    cout<<(calificacion>=6 ? "pasaste" : "reprobaste");
    cin.ignore();
    cin.get();
    return 0;
}
```

La forma de este operador es la siguiente:

condición `?` acción a ejecutar en caso verdadero `:` acción a ejecutar en caso falso `;`

La condición es: calificación>=6. En caso de ser verdadera se enviará “pasaste” como argumento a la instrucción `cout`, en caso contrario se enviará reprobaste.

switch …

Muchas veces nos metemos en aprietos cuando necesitamos tener el control sobre muchas opciones que pudiese tomar el usuario, porque resulta muy complicado pensar en varios if/else anidados, para esos casos tenemos otra herramienta muy cómoda, la estructura de selección múltiple `switch`.

La forma general es:

```
switch (parámetro a evaluar o comparar){
        case a :                    //cuando el parámetro tiene un valor a
                Acciones a ejecutar;
        case b:                     //cuando el parámetro tiene un valor b
                Acciones a ejecutar
        .
```

.

.

caso por default;

}

```
#include <iostream.h>
int main(){
    int opcion;
    cout<<"Menu de opciones"<<endl;
    cout<<"1.- Opcion 1"<<endl;
    cout<<"2.- Opcion 2"<<endl;
    cout<<"3.- Opcion 3"<<endl;
    cout<<"elige una opcion"<<endl;
    cin>>opcion;
    switch(opcion){
         case 1:
              cout<<"ejecucion 1"<<endl;
              break;
         case 2:
              cout<<"ejecucion 2"<<endl;
              break;
         case 3:
              cout<<"ejecucion 3"<<endl;
              break;
         default:
                 cout<<"es una opcion no valida"<<endl;
                 break;
    }
    cout<<"presiona enter para salir"<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Notemos que “opcion” es una variable de tipo entero, asi que `case 1` , se refiere a que la variable “opcion” tiene un valor de 1. Si se tratase de una variable de tipo carácter, entonces debería de escribirse `case ‘1’`.

Encontramos una nueva instrucción, `break`. Esta se encarga de alterar el flujo de control. Cuando se encuentra un `break`, inmediatamente se salta el resto de la estructura. No es exclusiva de la estructura `switch`, también se puede colocar en bloques `if`, `for`, `while` o `do while` (que se verán a continuación). Sin embargo es mejor no depender demasiado del `break`.

while

La sentencia de control `while` se encarga de repetir un bloque de código *mientras* se cumpla una condición. El bloque de código se debe de encontrar entre llaves, excepto si es una sola línea.

La forma general de esta instrucción es la siguiente:

```
while (condición a cumplir) {
        acciones a ejecutar;
}
```

Un ejemplo de éste lo podemos ver en el cálculo del promedio de 10 números, pensemos que son las calificaciones del semestre. Mirando rápidamente la forma de estructurar nuestro programa (algo que se debe de hacer siempre, y antes de sentarse frente a nuestro ordenador), podemos pensar en la siguiente solución:

```
Declarar variables e inicializarlas
Mientras (no se hagan 10 iteraciones){
        Leer una calificación
        Sumarla al total
        Determinar que se ha hecho una iteración
}
```

calcular el promedio
mostrar el promedio

El programa queda como sigue:

```
#include <iostream.h>
int main(){
    int calificacion, suma=0, iteracion=1;
    float promedio;
    while(iteracion<=10){
          cout<<"teclea tu calificacion "<<iteracion<<endl;
          cin>>calificacion;
          suma+=calificacion;
          ++iteracion;
    }
    promedio=(float)suma/(iteracion-1);
    cout<<"el promedio de calificaciones es: "
        <<promedio<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

El programa es sencillo, quizá una parte que pudiese causar dudas es el cálculo del promedio. Lo que sucede es que se realiza un cast para convertir temporalmente la variable suma de `int` a `float`, así se hará la división y se tendrá un resultado decimal cuando sea necesario.

Hasta aquí nuestro programa va bien, pero puede mejorar, podemos tener un mejor control del programa y hacer que el usuario decida cuantas calificaciones va a introducir. Podemos hacer que el usuario simplemente teclee un “-1” cuando acabe de escribir sus calificaciones, así, en lugar de comparar el número de iteraciones, comprobará si la calificación es igual a “-1” para salirse.

Aquí el programa:

```
#include <iostream.h>
int main(){
    int calificacion=0, suma=0, iteracion=0;
    float promedio=0;
    while(calificacion!=-1){
          ++iteracion;
          suma+=calificacion;
          cout<<"teclea tu califiaccion "<<iteracion<<endl;
          cin>>calificacion;
    }
    promedio=(float)suma/(iteracion-1);
    cout<<"el promedio de calificaciones es: "
        <<promedio<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Podemos ver como cambiando unas instrucciones de lugar y un valor obtenemos un mejor resultado.

La variable “iteracion” la inicializamos con un valor de 0, al inicio del bucle le aumentamos una unidad, luego a “suma” le añadimos la calificación que en este caso aún es 0, luego pide y lee la calificación tecleada.

Al siguiente ciclo comprueba si se escribió “-1”, si no entonces aumenta el contador “iteracion”, suma la calificación introducida, por ejemplo un 8, y pide la siguiente, el usuario teclea, de nuevo, 8.

Hace de nuevo una comprobación y entra de nuevo al ciclo, aumenta el valor de “iteracion” a hora tiene un 3, “suma” ahora vale 8 que era lo que tenía antes más el 8 recientemente leído, es decir 16, pide la siguiente calificación, y teclea la persona “-1”.

Otra comprobación da como resultado un falso (por que se escribió -1) y por lo tanto no entra al ciclo, entonces calcula el promedio, donde notemos que a "iteracion" se le resta uno para tener el verdadero número de calificaciones introducidas. Por último presenta el resultado.

Hay otras formas de hacer este programa, por ejemplo cuando no esta inicializada la variable de calificaciones o iteraciones, y necesitamos que ese ciclo se ejecute por lo menos una vez. En este caso podemos usar algo muy parecido, se trata de la sentencia `do while`.

do while

Cuando necesitamos que un ciclo se ejecute por lo menos una vez, es necesaria esta sentencia.

La forma general es:

```
do{
        Acciones a ejecutar;
}
while(condicion a cumplir);
```

Pongamos como ejemplo el mismo de la estructura `switch`, el menú de opciones. Necesitamos que por lo menos una vez lea la opción que elige, y en caso de que elija una no disponible, en lugar de aparecer el mensaje "es una opción no válida" y termine el programa, espere a que el usuario escoja una de la lista.

El programa de ejemplo:

```
#include <iostream.h>
int main(){
    int opcion;
    do{
        cout<<"elije una opcion de la lista"<<endl;
        cout<<"1.-opcion 1"<<endl;
        cout<<"2.-opcion 2"<<endl;
        cout<<"3.-opcion 3"<<endl;
        cin>>opcion;
    }while((opcion<1)||(opcion>3));
    cout<<"esta opcion si fue valida"<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Nunca debemos olvidar el punto y coma después del `while`. Usada en combinación con el bloque `switch`, podemos tener muy buenos resultados y encontrar varias aplicaciones de ellos.

for

El ciclo for es al que más se recurre cuando se requiere realizar operaciones secuenciales, en donde se conoce el número de iteraciones o la condición a comprobar.

La forma general de esta sentencia es:

```
for(asignación inicial ; condición ; instrucción ) {
        bloque de instrucciones;
}
```

Si se trata de una sola instrucción a ejecutar no es necesario ponerla entre llaves, incluso podría ponerse dentro del paréntesis el for, separado por una coma.

```
for(asignación inicial ; condicion ; instrucción1)
        unica instrucción;
```

ó, si se trata de una instrucción muy corta podría ponerse:

```
for(asignación inicial ; condicion ; instrucción1 , instrucción 2)
```

Pensemos en una aplicación. Hagamos un programa que calcule las coordenadas de un proyectil que se mueve en movimiento parabólico. Almenos determinaremos 10 puntos por los que pasa. Si se tienen dudas acerca de las fórmulas recomiendo que consulten un libro sobre física.

Las formulas son las siguientes:

$$x=(Vi)(\sin\Theta)(t)$$

$$y=(Vi)(\sin\Theta)(t)+\frac{1}{2}(g)(t^2)$$

Donde Vi es la velocidad inicial.

Θ es el ángulo con el que es lanzado.

t es el tiempo.

g es la gravedad.

La forma en que pensaría hacer este programa es la siguiente:

```
Declarar e inicializar variables
Pedir datos
for (contador=1; hasta que el contador llegue a 10; aumentar contador){
        Calcular coordenada x
```

Calcular coordenada y

Imprimir las coordenadas

}

Termina el programa.

No parece tan complicado ¿verdad? Pues en realidad no lo es tanto. El programa terminado aquí está.

```
#include <iostream.h>
#include <math.h>
int main (){
    int t;
    float x=0,y=0,ang,vi, grav=9.81;
    cout<<"cual es el angulo de lanzamiento?"<<endl;
    cin>>ang;
    cout<<"cual es la velocidad inicial?"<<endl;
    cin>>vi;
    ang=((ang*3.141592)/180);
    for(t=0;t<=10;t++){
          x=(vi*cos(ang)*t);
          y=(vi*sin(ang)*t)-(0.5*grav*t*t);
          cout<<"t= "<<t<<" x= "<<x<<" y= "<<y<<endl;
    }
    cout<<"fin del programa"<<endl;
    cin.ignore();
    cin.get();
    return (0);
}
```

El ciclo `for` comienza inicializando la variable del tiempo en 0. La condición que se tiene que cumplir para entrar en el ciclo es que t sea menor o igual a 10.

Cuando entra en el ciclo calcula las coordenadas *x* y *y*. Hacemos uso de las funciones cos y sin que se encuentran dentro de la librería `math.h`. Dichas funciones calculan el coseno y el seno de un ángulo determinado en radianes, es por eso que la variable “ang” es transformada a radianes. Luego de haber calculado las

coordenadas, las imprime en pantalla. Y finalmente ejecuta la última instrucción del `for` aumentar una unidad a *t* (t++).

En el momento en el que la variable *t* sea mayor que 10, no entrará en el ciclo y llegará el fin del programa.

CAPÍTULO 5 FUNCIONES

Cuando tratamos de resolver un problema, resulta muy útil utilizar la filosofía de "divide y vencerás". Esta estrategia consiste en dividir nuestro problema en otros más sencillos

Cuando realizamos un programa, por ejemplo, en el que se repetirán varias instrucciones pero con distintos valores que definan los resultados, podemos construir el programa a base de funciones. Una función es un bloque de instrucciones a las que se les asigna un nombre. Entonces, cada que necesitemos que se ejecuten esa serie de instrucciones, haremos una invocación a la función.

Ya hemos hecho uso de algunas funciones, en el capítulo anterior usamos `cos` y `sin`, que están presentes en la librería `math.h`. Así no nos preocupamos por todo lo que haga esa función para devolver un resultado, lo único que nos preocupó fue que datos teníamos que mandar y los que íbamos a recibir.

PROTOTIPOS

El prototipo de una función se refiere a la información contenida en la declaración de una función. Una función debe de estar definida o al menos declarada antes de hacer uso de ella.

Cuando se declara una función debe de especificarse el tipo de dato que va a devolver, el nombre de la función, y los parámetros. La siguiente declaración:

```
int suma(int a, int b);
```

Especifica una función que devuelve un tipo de dato entero, tiene por nombre suma, y al invocarla, recibe 2 parámetros de tipo entero.

Esta declaración debe de escribirse antes de la función main, y su definición puede escribirse después de ésta.

Por ejemplo:

```
#include <iostream.h>
int suma(int x, int y);
int main(){
    int dato1,dato2;
    cout<<"calculare la suma de 2 numeros"<<endl;
    cout<<"escribe el dato 1 y luego el dos"<<endl;
    cin>>dato1;
    cin>>dato2;
    cout<<"la suma es: "<<suma(dato1,dato2)<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
int suma(int a, int b){
    return a+b;
}
```

Observamos que primeramente declaramos la función suma para usarla en el bloque main, y al final del código la definimos. Podíamos haber escrito la función completa antes del main.

La función main también tiene un prototipo, y por lo tanto también puede recibir datos.

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

El parámetro `argc` es un entero que contiene el número de argumentos pasados a la función desde la línea de comandos. Es decir, almacena el número de argumentos en la tabla `argv`.

El parámetro `argv` es una tabla de cadenas de caracteres (se verá más adelante), cada cadena de caracteres contiene un argumento pasado en la linea de comandos, la primera cadena contiene el nombre con el que fue invocado y las siguientes son los demás argumentos.

El parámetro `envp` es una tabla de cadenas, que pasa información sobre una variable de entorno del sistema operativo.

PASO DE ARGUMENTOS

Cuando hablamos de argumentos, nos referimos a los valores que aparecen en la llamada a la función. En el ejemplo anterior los argumentos fueron “dato1” y “dato2”. Y los parámetros son “a” y “b”, los que reciben los datos.

Notemos que el prototipo de la función tiene los nombres de los parámetros distintos a los de la definición, esto no afectará el comportamiento del programa, pero se vería mejor si fueran iguales. Lo que no puede cambiar es el tipo de datos que va a recibir.

En el caso del main, el paso de argumentos se realiza desde la línea de comandos, como ejemplo ponemos el caso de MS-DOS con el comando dir. Si invocamos a este comando escribimos dir /w /p; el argumento argc tiene el valor 3, y argv contiene 3 cadenas de caracteres: dir, /w, /p.

VALORES DE RETORNO

Una función puede regresar cualquier tipo de valor excepto tablas u otras funciones. Esta limitación podría resolverse utilizando estructuras o punteros, pero se verá más adelante.

MACROS

Una macro es una parte del código que puede parecer y actuar como una función, se define después de las librerías mediante un `#define`.

Se denominan instrucciones de preproceso, porque se ejecutan al comienzo de la compilación.

Sin embargo, no hay que abusar de ellas, porque podrían entorpecer el código y hacer lento el programa.

```
#include <iostream.h>
#define mayor(a,b)   (a>b)? a: b
int main(){
    int a,b;
    cout<<"teclea 2 numeros distintos"<<endl;
    cin>>a;
    cin>>b;
    cout<<"el mayor de esos numeros es: "
        <<(mayor(a,b))<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Debemos tener cuidado si vamos a usar macros, las macros, al compilar el programa, se sustituyen directamente en donde se invocan, es decir, que en este ejemplo, es como si se introdujera directamente los condicionales dentro del `cout`. Así, debemos tener cuidado en introducir bloques grandes de código en una macro (algo nada recomendable), sobre todo, cuidado con los puntos y comas.

Podemos ver que las macros al igual que las funciones pueden tener parámetros, sin embargo, tienen que escribirse con cuidado.

Por ejemplo, al hacer una macro de la siguiente manera:

```
#define curva(a) 1+2*a+a*a
```
[4]

si en nuestro código colocamos una instrucción

```
curva(1+3)
```

se sustituye por

```
1+2*1+3+1+3*1+3
```

contrario a lo que esperaríamos si invocáramos curva(4), 13 en lugar de 25. Habría que definir la macro como 1+2*(a)+(a)*(a).

El uso de macros hace más difícil la depuración de un programa, porque en caso de haber errores en la macro, el compilador nos avisará que hay errores, pero en la implementación de la macro. Además de que no realizará comprobaciones de tipo de datos ni conversiones.

[4] Ejemplo del libro Programación en C/C++, Alejandro Sierra Urrecho

En el lenguaje C, se acostumbra usar macros para definir constantes (porque a diferencia de C++, ahí no existe el tipo `const`. Por ejemplo:

```
#define PI 3.141592
```

cambiará todas las apariciones de la palabra “PI” por el valor 3.141592.

RECURSIVIDAD

La recursividad se presenta cuando una función se invoca a si misma. Distintamente a las iteraciones (bucles), las funciones recursivas consumen muchos recursos de memoria y tiempo.

Una función recursiva se programa simplemente para resolver los casos más sencillos, cuando se llama a una función con un caso más complicado, se divide el problema en dos partes, la parte que se resuelve inmediatamente y la que necesita de más pasos, ésta última se manda de nuevo a la función, que a su ves la divide de nuevo, y así sucesivamente hasta que se llegue al caso base. Cuando se llega al final de la serie de llamadas, va recorriendo el camino de regreso, hasta que por fin, presenta el resultado.

Un ejemplo clásico para este problema es calcular el factorial de un número, (n!). Conocemos los casos base, 0!=1, 1!=1. Sabemos que la función factorial puede definirse como $n! = n \cdot (n-1)!$. Entonces, 5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 120.

Nuestro programa quedaría como sigue:

```
#include <iostream.h>
long factorial(long numero);    //prototipo
int main(){
    long numero;
    cout<<"número para calcular el factorial:"<<endl;
    cin>>numero;
    cout<<"el factorial es: "<<factorial(numero)<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
long factorial(long numero){
     if(numero<=1)
                  return 1;
     else
         return numero*factorial(numero-1);
}
```

Nuestro programa llama a la función factorial con un número inicial, posteriormente, esta función se llama a sí misma cada vez con número más pequeño hasta que llega al caso base (numero<=1).

La recursividad es un tema importante en el mundo de la programación, la utilidad de este tipo de funciones se aprovecha en el diseño de algoritmos de criptografía, de búsqueda, entre otros. La implementación de la recursividad en nuestros programas debe de evaluarse con mucho cuidado, debemos de tratar de evitar que se usen funciones de complejidad exponencial, que se llamen a sí mismas una y otra vez sin que se tenga un control claro.

CAPÍTULO 6 ARRAYS Y CADENAS

Cuando declaramos una variable estamos apartando en memoria espacio para guardar sus posibles valores dependiendo del tipo de dato que se trata. Un array o arreglo son una serie de localidades en memoria consecutivas que están asignadas a un solo nombre y un mismo tipo de datos.

	Valor almacenado	Localidad en memoria
Arreglo[0]	1550	FFFB
Arreglo[1]	130	FFFC
Arreglo[2]	45	FFFD
Arreglo[3]	90	FFFE
Arreglo[4]	76	FFFF

Ya anteriormente hemos presentado ejemplos en los que se hace uso de los arreglos, y aunque no comprendimos exactamente que significado tenía la aplicación de los arreglos nos dio una idea de sus utilidades.

DECLARACIÓN

La forma de declarar un arreglo de cualquier tipo de datos es la siguiente:

tipo nombre [tamaño] ;

Por ejemplo, podemos declarar un arreglo de enteros con 12 elementos.

int MiArreglo [12] ;

El compilador se encargará de asignar la memoria requerida para almacenar determinados valores.

Cuando se declara un arreglo de caracteres se trata entonces de una cadena.

char nombre[20] ;

ASIGNACIÓN DE VALORES

Al momento de declarar un arreglo de cualquier tipo, podemos inicializarlo con los valores que queramos. Para inicializar un arreglo de enteros:

int MiArreglo[5] ={2,34,78,1,9};

Así, estos valores estarán almacenados en cada elemento del array. Es muy importante hacer notar que el primer elemento de un arreglo es el elemento 0, entonces, MiArreglo[0] contendrá el número 2, el segundo (MiArreglo[1]) contendrá el número 34 y así sucesivamente hasta MiArreglo[4] que es el último elemento del array. Si un arreglo cuenta con menos inicalizadores que elementos entonces el resto se inicializará a 0.

Y en caso de que se trate de una cadena de caracteres podemos hacerlo de 2 formas:

char MiCadena[13]= “hola a todos”;

o bien, declarar cada elemento

char MiArray[5]={'h','o','l','a','\0'};

Cuando se inicializa una cadena por el primer método, automáticamente se coloca un carácter de terminación de cadena (el carácter \0), en cambio, de la segunda forma debemos de ponerlo nosotros. También se puede excluir el tamaño del arreglo, el compilador lo determinará en la compilación.

Para acceder a cada elemento del arreglo debemos especificar su posición (la primera es la posición 0). En tiempo de ejecución podemos asignar valores a cada elemento cuidando siempre de no sobrepasar el tamaño del arreglo, si sobrepasamos ese tamaño se escribirán datos en un área de la memoria que no está asignada para ese array, puede escribir datos en un área en donde se almacenaba otra variable del programa o un componente del sistema, esto ocasionaría resultados no deseados.

```
#include <iostream.h>
#include <iomanip.h>     //para presentacion con formato
#include <stdlib.h>
#include <time.h>
int main(){
    const int tam_max=20;
    int aleatorios[tam_max];
    int i,suma=0;
    float promedio;
    srand((unsigned)time(NULL));
    for(i=0;i<tam_max;i++){
         aleatorios[i]=rand()%128;   //asigna el numero
         cout<<"aleatorio generado: "   //presenta el numero
             <<setw(5)<<aleatorios[i]<<endl;
    }
    for(i=0;i<tam_max;)
         suma+=aleatorios[i++];     //suma los elementos
    promedio=(float)suma/tam_max;
    cout<<endl<<"el promedio es: "<<promedio<<endl;
    cin.get();
    return 0;
}
```

El programa anterior muestra el uso de los arreglos. Ya anteriormente habíamos hecho un ejercicio que calculaba el promedio de una serie de números introducidos por el usuario, en este caso el programa guarda los números generados por la máquina en un arreglo para luego calcular el promedio.

Para asignar cada valor aleatorio generado, se utiliza un ciclo `for` con un contador “i” que va de 0 hasta el tamaño máximo que pudiera tener el array. El tamaño del array fue declarado como una constante al principio de la función principal, esto ayudará en caso de que se quiera aumentar el tamaño del arreglo, así no tendremos que modificar cada parte de nuestro código en la que se refiera al tamaño, bastará con cambiar el valor de la constante.

Dentro del ciclo se genera un número aleatorio con la función `rand()`, usamos el operador para obtener el resto de la división entera, así nos dará un número menor de 128. Previamente se utilizó la función `srand()` para establecer una semilla para generar los números aleatorios, esa semilla la tomamos llamando al reloj del sistema con la función `time()`. Al mismo tiempo imprimimos el número que fue generado.

Posteriormente, se hace de nuevo un ciclo para acceder a los valores almacenados y sumarlos en una variable, aunque esto se pudo haber hecho dentro del ciclo anterior servirá para mostrar las formas de asignación y consulta de valores. Además del uso del incremento después de la asignación.

Finalmente se calcula el promedio.

En el caso de las cadenas de caracteres también podemos acceder sus elementos de la misma manera, para la asignar valores es mejor usar las funciones para la lectura de caracteres, tales como `getline()`, porque nos permite tener un control de los caracteres que se introducirán y también introduce el carácter de terminación

de cadena, cosa que no es posible con `cin>>` directamente y además de que puede escribir en sectores de memoria no adecuados.

El siguiente programa es una muestra del uso de arreglos de caracteres.

```
#include <iostream.h>
int main(){
    const int tam_cad=20;
    char cadena[tam_cad];
    cin.getline(cadena,tam_cad,'\n');
    for(int i=0;(i<tam_cad)&&(cadena[i]!='\0');i++)
                cadena[i]+=5;
    cout<<cadena<<endl;
    cin.get();
    return 0;
}
```

¿Qué hace este programa?, sencillamente podemos decir que almacena una cadena introducida por el usuario, para luego modificar cada carácter por medio de un ciclo hasta que encuentre el carácter de terminación (\0) o llegue al limite de caracteres.

Quizá no parezca muy interesante este ejemplo, pero hay que recordar que con cosas pequeñas se construyen grandes cosas, adornemos un poco nuestro programa, supongamos que la cadena que introduce el usuario es un mensaje que debe ser secreto, en lugar de sumar 5 a cada carácter podemos sumarle el número que quiera el usuario, ¿por qué no una clave?.

El mismo programa pero un poco “adornado” o “presuntuoso” queda de la siguiente forma.

```
#include <iostream.h>
int main(){
    const int tam_cad=20;
    char cadena[tam_cad];
    int clave;
```

```
    cout<<"introduce el texto a cifrar"<<endl;
    cin.getline(cadena,tam_cad,'\n');
    cout<<"ahora dame tu clave (no mas de 3 digitos)"<<endl;
    cin>>clave;
    for(int i=0;(i<tam_cad)&&(cadena[i]!='\0');i++)
                 cadena[i]+=clave;
    cout<<"el texto cifrado es: "<<endl;
    cout<<cadena<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Como ejercicio sugiero que haga un programa que realice la acción contraria, de un texto cifrado saque el mensaje oculto, y ¿por qué no? intente descifrar un mensaje sin la clave.

La manipulación de cadenas es uno de los temas más extensos en cuestiones de programación y de los más importantes. Desde las acciones más básicas como la concatenación de cadenas, la ordenación y búsqueda en arreglos, hasta los complicados algoritmos de criptografía o la búsqueda a velocidades increíbles.

PASO DE ARGUMENTOS

Así como en ejemplos anteriores pasamos una variable a una función, también se puede pasar como argumento un arreglo. Para pasarlo es necesario hacer una llamada como la siguiente:

MiFuncion (arreglo);

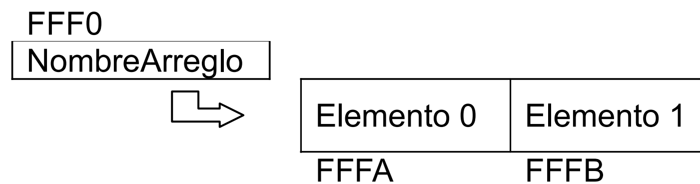
Y el prototipo de la función debe de ser de la siguiente forma:

tipoRetorno MiFuncion (tipoArreglo nombre [ ]);

No se requiere incluir el tamaño del arreglo dentro de los corchetes, si se escribiera el compilador lo ignora.

Cuando pasamos como argumento a una función una variable de tipo `int`, `float`, `char`, etc., estamos pasando el valor de esta variable a otra variable (podría decirse una copia) que se define en la función, esta forma de pasar argumentos se denomina "por valor". Cualquier modificación de valor a esta variable no tendrá efecto en la variable "original" que pasó su valor a la función. Esta última variable, al ser local, será destruida cuando termine de ejecutarse la función que la contiene.

En el caso del paso de arreglos, este no se hace por valor, sino que se hace un paso por referencia simulado. Cuando se declara un arreglo, el nombre de éste es una variable que apunta al primer elemento del arreglo, es decir que contiene su dirección en memoria.



Cuando pasamos esta dirección a una función, cualquier modificación que se haga dentro de esa función de nuestro arreglo tendrá efectos en la variable original.

El siguiente programa da un ejemplo:

```
#include <iostream.h>
void modifica(int arreglo[], int tamano);
int main(){
    const int tam_arr=5;
    int MiArreglo[tam_arr]={5,10,15,20,25};
    int i;
    modifica(MiArreglo, tam_arr);
    for(i=0;i<tam_arr;++i){
         cout<<"elemento "<<i<<"es "
             <<MiArreglo[i]<<endl;
    }
    cout<<"presiona enter para terminar"<<endl;
    cin.get();
    return 0;
}
void modifica(int arreglo[], int tam_arr){
     for(int i=0;i<tam_arr;++i)
             arreglo[i]*=2;
}
```

Se ha declarado e inicializado un arreglo de enteros de 5 elementos. Posteriormente se llama a la función modifica, la cual multiplica cada elemento del arreglo por dos, estos cambios de valor en los elementos del arreglo tienen efecto en “MiArreglo” aunque el nombre del parámetro de la función modifica sea diferente.

También se puede pasar a una función el valor de un elemento del array, sólo se pasará el valor del elemento, no se puede modificar el elemento original.

FuncionInvocada (array [elemento]);

En ocasiones puede resultar poco conveniente que alguna de las funciones que reciben un arreglo tenga derecho a modificarlo, a veces puede que cambie sus valores sin que nosotros nos percatemos inmediatamente. Para eso podemos incluir el calificador `const` dentro del prototipo y definición de la función, de esta manera no podrá modificarse nada de este arreglo.

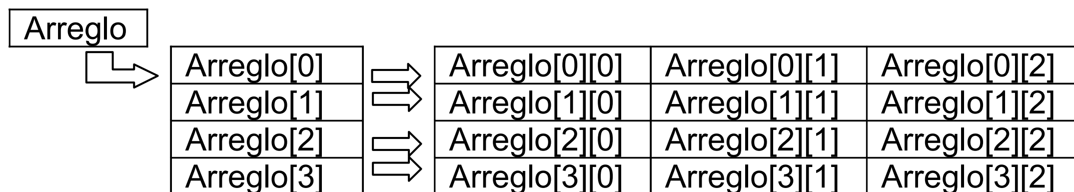
tipoRetorno nombreFuncion (const tipoArreglo nombre [ ]);

ARREGLOS MULTIDIMENSIONALES

Como sabemos, se pueden crear arreglos de cualquier tipo de datos, ahora veremos que también se pueden crear arreglos de arreglos. La forma de declararlos es con múltiples subíndices, cuantos se quieran tener, nosotros trabajaremos a lo más con dobles subíndices.

tipoArreglo nombre [subíndice1] [subíndice2] ;

La forma en que queda estructurado un arreglo de dos dimensiones es más o menos como se muestra en la figura.



El nombre del arreglo tiene la dirección del primer elemento (Arreglo[0]), y éste a su vez tiene la dirección del otro arreglo (Arreglo[0][0]), así para cada elemento de arreglo(0, 1, 2, 3).

Cuando se inicializa un arreglo multidimensional se inicializa entre llaves cada arreglo o bien, por medio de elemento seguidos:

Array1 [2] [3] = {1, 2, 3, 4, 5, 6};
Array2 [2] [3] = {{2, 4, 6}, {8, 10, 12}};

Al mandar un arreglo de dos dimensiones, debemos de especificar en el prototipo y definición de la función el subíndice2 de nuestro array. El siguiente ejemplo mostrará varios aspectos de ellos.

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <time.h>

const int vendedores=5, meses=4;
void inicializa(int ventas[][meses]);
void calculaTotal(const int ventas[][meses]);

int main(){
    int ventas[vendedores][meses];
    int i;

    //Las siguientes instrucciones solo presentan un
    //   encabezado para la tabla
    cout<<"ventas:"<<setw(23);
    for(i=1;i<=meses;++i)
        cout<<"   mes"<<i;
    cout<<endl;
    //fin de encabezado

    inicializa(ventas);
    calculaTotal(ventas);
    cout<<endl<<endl<<"presiona enter para terminar"<<endl;
    cin.get();
    return 0;
}
void inicializa(int ventas[][meses]){
     int i,j;     //contadores
     srand((unsigned)time(NULL));
     for(i=0;i<vendedores;++i){
          cout<<"ventas del vendedor "<<i+1<<": ";
          for(j=0;j<meses;++j){
               ventas[i][j]=rand()%50;
               cout<<setw(7)<<ventas[i][j];
          }
```

```
          cout<<endl;
     }
}
void calculaTotal(const int ventas[][meses]){
     int suma=0,total=0,i,j;
     cout<<endl<<endl;
     for(i=0;i<vendedores;++i){
          suma=0;
          for(j=0;j<meses;++j){
               suma+=ventas[i][j];
          }
          cout<<"el vendedor "<<i+1<<" vendio "
              <<suma<<" productos"<<endl;
          total+=suma;
     }
     cout<<endl<<endl<<"en total se vendieron: "
         <<total<<" productos en "<<meses<<" meses"<<endl;
}
```

Notemos primero cómo el prototipo de las funciones “inicializa” y “calculaTotal” debe incluir el segundo subíndice del arreglo. La función “calculaTotal” muestra un ejemplo del uso del calificador `const` para evitar que en esta función se modifique algún valor del arreglo.

La función “inicializa” llena el arreglo con valores aleatorios menores a 50, se utilizan dos contadores para dos ciclos `for`, “i” para el primer subíndice, “j” para el segundo. El ciclo recorre la primera “fila” (viendo el arreglo como una tabla de valores) usando el contador i y a cada celda, con ayuda de “j”, asigna valores, así cuando “j” llega al límite de celdas (meses) se aumenta una unidad el contador “i” para continuar con la otra fila y recorrerla de nuevo.

La función “calculaTotal” tiene una forma muy similar, pero en ésta no se modificarán valores, sino que se hará la suma de los elementos de cada fila para presentarlos en pantalla. Una variable “total” se encarga de ir sumando los productos vendidos cada

que se cambia de fila en la tabla, para después, al término de los ciclos presentar el total de artículos vendidos en esos meses.

Puede aumentarse el número de filas y columnas de la tabla modificando los valores de las constantes “vendedores” y “meses” respectivamente, definidas al comienzo del programa.

En cuanto al uso de arreglos cadenas de caracteres se dejará para el siguiente capítulo, porque es necesario considerar conceptos de direcciones de memoria, el uso de los apuntadores.

CAPÍTULO 7 APUNTADORES

OPERADORES REFERENCIA-DESREFERENCIA

Un apuntador se define como una variable que contiene una dirección de memoria. Para declarar un apuntador se hace uso del operador desreferencia o indirección (*), no se debe confundir con la multiplicación.

Al declarar un apuntador debemos de hacerlo de la siguiente forma:

tipo *nombre;

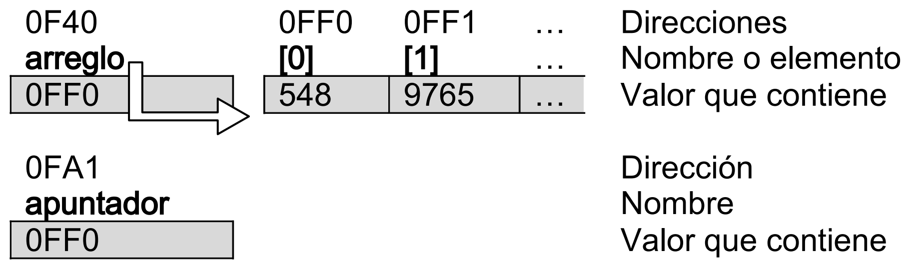
```
int *MiApuntador;
```

Ya habíamos dicho anteriormente que el nombre de un arreglo apunta al primer elemento de éste, es decir contiene la dirección en memoria del elemento 0. Tomando en cuenta esto, podemos ver que si hacemos:

```
int arreglo[10];
int *apuntador;

apuntador = arreglo;
```

sucede que, pasamos la dirección del primer elemento del arreglo a la variable “apuntador” que está determinada para almacenar una dirección de memoria.



Las variables también tienen una dirección en memoria, y para saber cual es debemos utilizar el operador referencia o dirección (&).

```
int MiVariable=20;
int *MiApuntador;

MiApuntador = &MiVariable;
```

Así "MiApuntador" contendrá la dirección de "MiVariable".

El uso de los apuntadores sirve para hacer uso de una variable con la cual no se tiene contacto directo, por medio del operador desreferencia. Por ejemplo, podemos asignar un nuevo valor a esta variable referenciada.

```
*MiApuntador=9;
```

por medio de la instrucción anterior se está asignando un valor de 9 a la variable que apunta "MiApuntador". Es decir que "MiVariable" ahora contiene un 9.

Un pequeño ejemplo nos mostrará lo dicho hasta el momento:

```
#include<iostream.h>
int main(){
    int variable=10;
    int *apuntador;
    cout<<"la variable contiene un "<<variable<<endl;
    apuntador=&variable;
    cout<<"variable esta en la direccion "
        <<apuntador<<endl;
    *apuntador=15;
    cout<<"la variable ahora tiene un "<<variable<<endl;
    cout<<"ahora tecle un numero"<<endl;
    cin>>*apuntador;
    cout<<"y ahora contiene un "<<*apuntador<<endl;
    cin.ignore();
    cin.get();
    return 0;
}
```

Como podemos ver podemos usar un apuntador para asignar cantidades (o lo que sea) a la variable referenciada ya sea por instrucción o por medio del teclado.

En la última instrucción `cout` se imprime el valor de “lo que apunta ‘apuntador’ ”.

ARITMÉTICA DE OPERADORES

Quizá ya se dio cuenta, así como la multiplicación es el inverso de la división, en el caso de los apuntadores sucede algo parecido, si se coloca una instrucción como la que sigue:

```
cout<< “la variable contiene ” <<&*variable;
```

o

```
cout<< “la variable contiene ” <<*&variable;
```

tendrán el mismo resultado que si no se colocaran los operadores * y &.

Un arreglo se puede considerar un apuntador (por las razones anteriormente mencionadas), y por lo tanto se puede aplicar la aritmética de operadores en ellos, en realidad no es muy diferente de cómo lo hacíamos ya antes, pero ahora utilizaremos el “ * ”.

Un array puede ser operado mediante incrementos y decrementos, por ejemplo:

```
MiArreglo [ 7 ];
```

También puede hacerse:

```
*(MiArreglo + 7);
```

Aunque no es nada recomendable utilizar este método para tratar los arreglos, nos da la idea de que se pueden hacer operaciones de suma y resta sobre los apuntadores. Si hacemos una instrucción como:

apuntador += 6;

la dirección de memoria almacenada cambia a 3 lugares adelante. Por consiguiente, también se puede aplicar una resta.

Cuando encontramos una instrucción del tipo:

*puntero = una_variable+6;

se está haciendo una asignación a “lo que apunta ‘puntero’”. Se modificará el valor de la variable referenciada por “puntero”. Igual pasa con asignaciones de otro tipo:
*puntero *=3;

que en este caso se hace una multiplicación de la variable referenciada por 3 y luego se vuelve a asignar.

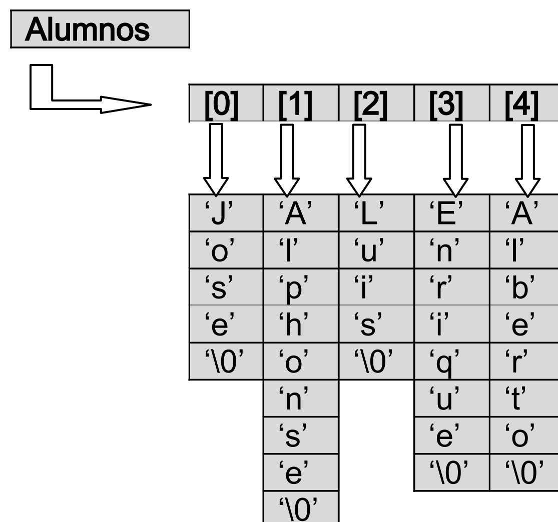
ARREGLOS DE APUNTADORES (ARREGLOS DE CADENAS)

Habíamos dejado pendiente un momento este tema, porque era mejor ver primero el concepto de apuntadores, ahora que ya sabemos en que consiste, podemos declarar e inicializar una cadena de caracteres de la siguiente forma:

char *cadena = “alguna_palabra”

Si quisiéramos declarar una cadena con un determinado número de elementos, es necesario utilizar uno de los métodos ya conocidos.

Un arreglo de cadenas en realidad se trata de un arreglo de apuntadores porque una cadena es un apuntador al primer carácter. Entonces, un arreglo de cadenas se vería de la siguiente forma.



La declaración del arreglo “Alumnos” sería de la siguiente forma:

```
char *Alumnos[4] = {“Jose”, “Alphonse”, “Luis”, “Enrique”, \
“Alberto”};
```

Notemos que de esta forma no es necesario colocar el carácter ‘\0’ al final de cada cadena, el compilador la coloca por sí sólo. También se pueden utilizar las otras formas, que ya conocemos, de inicializar un arreglo de dos dimensiones y se puede especificar el tamaño de las cadenas.

char MasAlumnos[4][10];

Veamos entonces que Alumnos[0] contiene la dirección en memoria que ocupa la letra ‘J’.

Y para acceder a cada carácter del arreglo declarado de cualquier forma, se utilizan los subíndices, por ejemplo, asignaremos con la siguiente instrucción un carácter contenido en el arreglo a otra variable de tipo char:

```
caracter = Alumnos [1][5];
```

Así asignamos el carácter ‘n’ a la variable “caracter”

Un ejemplo nos ayudará a visualizar su utilización. El siguiente programa se trata de un juego, o por lo menos la idea de cómo empezarlo. Consiste en teclear, lo más rápidamente, las palabras que vallan apareciendo en la pantalla, se cuenta con un determinado tiempo para hacerlo, si no se consigue dentro de ese intervalo continuará a la siguiente palabra. El programa termina cuando presionan la tecla “Esc”, pero bueno, aquí está el código.

```
#include<conio.h>
#include<stdlib.h>
#include<iostream.h>
#include<time.h>
int main(){
     int cont=0,i=0,num_aleat=0;
     char car;
    char *cadena[6]={"cinco","siete","texto","poema","radio",\
         "raton"};
     while(car!=27){
         srand((unsigned)time(NULL));
         num_aleat=rand()%6;
         cout<<endl<<endl<<cadena[num_aleat]<<endl;

for(cont=0,i=0;(cont<100000)&&(cadena[num_aleat][i]!='\0');++cont){
          if(kbhit()){
               car=getch();
                if(car==cadena[num_aleat][i]){
                  cout<<car;
                  ++i;
              }
                fflush(stdin);
          }
          }
    }
     cout<<endl<<"juego terminado"<<endl;
     cin.get();
     return 0;
}
```

El programa define primeramente una matriz de cadenas, para luego empezar el juego mediante un ciclo `while`. Repetirá todas las instrucciones dentro del bloque mientras no se oprima la tecla “Esc”. Se calcula un número aleatorio menor a 6 y se almacena en una variable tipo int, la cual servirá para que se mande a pantalla, mediante la instrucción `cout`, una cadena al azar, note que se le está enviando la dirección del primer carácter de la cadena.

Después se usa un ciclo `for`, en el cual se inicializa la variable que contará el tiempo (cont) y la variable de desplazamiento por los elementos de la tabla (i). Este ciclo se detendrá cuando el contador de tiempo llegue a 100000 o cuando se llegue al carácter '\0' en la cadena.

Dentro de ese ciclo `for` se encuentra la parte más interesante del programa, usamos una función (`kbhit`) que se encarga de detectar si se ha oprimido una tecla, esta tecla no aparecerá en la pantalla pero sí se quedará dentro del flujo, si detecta que se ha oprimido entonces, usando la función `getch()`, leerá y guardará en la variable car aquel carácter que se quedó dentro del flujo, pero sin presentarlo en pantalla, si el carácter leído corresponde con el carácter de la posición referida por "i", entonces si se presentará en pantalla y se continuará con el siguiente carácter (++i). Saliendo del condicional se limpia el flujo de entrada (función `fflush()` definida en `stdlib.h`) para evitar que se quede algún residuo o basura.

Las Funciones `kbhit()` y `getch()` no son parte de la librería estándar de C++, estas librerías son propiedad de la compañía Borland® y que se distribuyen en sus productos Borland/TurboC++. Sin embargo, debido a su amplio uso, están disponibles también versiones para otros productos como lo son el Visual C++ de Microsoft®, el Dev-C++ de Bloodshed, y por supuesto para el compilador GCC.

PASO DE ARGUMENTOS POR REFERENCIA

Anteriormente vimos como se hacia la modificación de un arreglo mediante el paso por referencias simulado. En el caso de las variables de tipo `int`, `float`, etc., también es posible hacer uso de ellas mediante el paso por referencia.

El paso por referencia consiste en pasar la dirección en memoria que ocupa esa variable para que la función que la recibe pueda hacer uso de ella, sin necesidad de hacer una copia como ocurre con el paso por valor.

Para mandar la dirección es necesario utilizar el operador & dentro de la invocación a la función. Y el prototipo y definición de la función debe especificar que será un apuntador mediante el uso del *.

tipo_que_regresa nombre_funcion (tipo_apuntador *nombre, otro_tipo nombre, …);

Lo anterior es la forma de una definición que hace uso de apuntadores. La función puede devolver o no un valor mediante una instrucción return(). Puede contener varios parámetros apuntador o no apuntador.

Cuando se va a hacer uso de esa función, la invocación es de la siguiente forma:

nombre_funcion (&nombre, …);

En el caso de los arreglos recordemos que estos son la dirección, y no es necesario poner el operador &.

nombre_funcion (arreglo[numero_cadena]);

La instrucción anterior, siendo “arreglo” un array de cadenas, envía la dirección del primer carácter de la cadena numero_cadena.

El siguiente ejemplo mostrará estos aspectos.

```
#include<iostream.h>
void aMayusculas(char *cadena);
void cambiaNum(int *numero);
int main(){
    char palabra[10];
    int numero;
    cout<<"teclea una cadena "<<endl;
    cin.getline(palabra,10);
    aMayusculas(palabra);
    cout<<endl<<palabra<<endl;
    cout<<"ahora teclea un numero"<<endl;
    cin>>numero;
    cambiaNum(&numero);
    cout<<"este numero se cambio a: "<<numero;
    cin.ignore();
    cin.get();
    return 0;
}
void aMayusculas(char *cadena){
    for(int i=0;cadena[i]!='\0';++i)
             cadena[i]+=('A'-'a');
}
void cambiaNum(int *numero){
     *numero *= *numero;
}
```

El código no es difícil de entender, quizá la parte en la que cusa un poco de confusión podría ser la función “cambiaNum”, simplemente lo que hace es multiplicar “lo que apunta ‘numero’” por sí mismo, es decir lo eleva a cuadrado. Revise la tabla de precedencia de operadores si se tienen más dudas sobre esto.

Gracias a las referencias de C++ es posible hacer una función `cambiaNum()` con una sintaxis más sencilla y de una manera más amigable, utilizando el '&'.

```
void cambiaNum(int & numero){
     numero *= numero;
}
```

de esta manera surtirá el mismo efecto que la que habiamos definido, cuando se invoque sólo se hará:

```
cambiaNum(numero);
```

como si fuese una llamada normal.

CAPÍTULO 8 ESTRUCTURAS

En muchos lenguajes, se le permite al programador definir sus propios tipos de datos a partir de los tipos atómicos. En C y C++ pasa lo mismo, y además existen de antemano los tipos union, enumeración y estructura. Estos tipos definidos forman parte del estandar de C, y por consiguientede C++, en éste último existe también el tipo class.

Para los objetivos de este trabajo excluiremos el tipo class (clase), pero a lo largo del capítulo mencionaremos en varias ocasiones este concepto para tener una idea de lo que se trata.

ENUMERACIONES

En ocasiones habremos de declarar constantes enteras una tras otra con valores consecutivos, por ejemplo:

```
const int a=0,b=1,c=2,d=3;
```

para evitar este tipo de declaraciones una tras otra, existe el tipo enumerativo. La declaración anterior se puede hacer de una mejor forma:

```
enum {a,b,c,d};
```

Una enumeración puede ser global o local. Además de que puede tener un nombre que los identifique, podemos indicar desde donde empieza la numeración.

```
enum meses { enero=1, febrero, marzo, abril, mayo, junio,
julio, agosto, septiembre, octubre, noviembre, diciembre};
```

Los valores que se le asignarán empiezan por 1 a enero, 2 a febrero y así sucesivamente. Si se le asignase un 5 a marzo, entonces abril tendría un 6 y así continuaría la numeración.

Después de hacer esta enumeración, se puede hacer una nueva a partir del nombre.

```
meses calendario;
```

Un ejemplo sencillo ilustrará nuestras ideas.

```
#include<iostream.h>
enum colores{rojo,amarillo,verde};
int main(){
        int i=0,contador=0;     //unos contadores
        colores semaforo;
        cout<<"este programa mostrara el tipo enumerativo,"
            <<" por medio de un semaforo de crucero"
            <<" simulado"<<endl;
        while(contador<3){
        semaforo=verde;
        cout<<"semaforo en verde, el peaton no pasa"<<endl;
        cout<<"el peaton pide el paso con un boton"
            <<"(presione enter)"<<endl;
        cin.get();
        semaforo=amarillo;
        cout<<"semaforo en amarillo, precaucion"<<endl;
        for(i=0;i<500000000;i++);
        semaforo=rojo;
        cout<<"semaforo en rojo, ALTO, pasa el peaton"<<endl;
        for(i=0;i<500000000;i++);
```

```
        contador++;
        }
        return 0;
}
```

Debemos tomar en cuenta que al declarar un tipo de variable en base a una enumeración, en este caso una variable semaforo de tipo colores, sólo se podrá asignar valores declarados dentro de esta, para el ejemplo anterior sólo puede tomar valores rojo amarillo o verde.

UNIONES

El tipo de datos union puede contener datos de tipos y tamaños diferentes, esta variable almacenará cualquier tipo de dato en una única localidad en memoria. Por ejemplo:

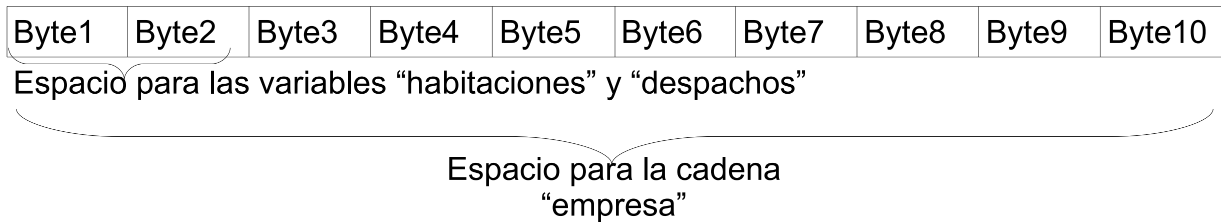
```
union edificio{
     int habitaciones;
     int despachos;
     char empresa[10];
};
```

declara un nuevo tipo de dato llamado "edificio", para usar una variable de tipo edificio bastará con hacer una declaración como:

```
edificio casa;
```

la variable casa tendrá el suficiente espacio en memoria para almacenar una cadena (el tipo de dato de mayor longitud dentro de la union). Es responsabilidad del programador extraer adecuadamente el tipo de dato deseado.

La forma en que está estructurada en la memoria nuestra variable es como la que sigue:



Cuando se almacena un entero y posteriormente una cadena, ésta última se escribe sobre el espacio de memoria asignado al entero, y por lo tanto, el contenido se pierde.

Practique con el siguiente programa.

```
#include<iostream.h>
#include<iomanip.h>

union edificio{
        int habitaciones;
        int despachos;
        char empresa[10];
};

int main(){
        edificio casa, agencia, gobierno;
        cout<<"cuantas habitaciones tiene tu casa?"<<endl;
        cin>>casa.habitaciones;
        cout<<"escribe el nombre de la agencia en la\
        que trabajas"<<endl;
        cin.ignore();
        cin.getline(agencia.empresa,10);
        cout<<"cuantos despachos tiene la oficina de gobierno \
        de tu localidad?"<<endl;
        cin>>gobierno.despachos;
     cout<<"cuantos despachos tiene el edificio de la agencia "
            <<agencia.empresa<<"?"<<endl;
```

```
        cin>>agencia.despachos;
        cout<<"edificio"<<setw(25)<<"habitaciones/despachos"
            <<endl;
        cout<<"casa"<<setw(18)<<casa.habitaciones<<endl;
        cout<<"agencia"<<setw(15)<<agencia.despachos<<endl;
        cout<<"gobierno"<<setw(14)<<gobierno.despachos<<endl;
        cin.ignore();
        cin.get();
        return 0;
}
```

Para accesar a los miembros de una variable de tipo `union` es necesario utilizar el operador punto (.), seguido de la variable a extraer. En este ejemplo hemos extraído un entero de la variable “casa” por medio de `casa.habitaciones`, si se hace una extracción como `casa.empresa` no sabemos exactamente que es lo que aparecerá porque pueden existir otros datos en esa área de la memoria asignada a esta cadena; y si se hiciese lo contrario, asignar una cadena y luego extraer un entero, seguramente no obtendremos lo que deseamos. Como con cualquier otro tipo de datos, podemos realizar operaciones con el contenido de los miembros de la union, operaciones básicas, envío como argumentos, conversiones, etc.

El uso de las variables de tipo `union` no es muy común, alguien seguramente habrá querido extraer un tipo entero, luego asignar una cadena y posteriormente utilizar de nuevo ese entero. Que útil seria que dentro de una sola variable podamos tener oportunidad de extraer tanto un tipo como otro sin perder los datos, ¿se imagina poder regresar mediante la instrucción `return` más de un dato?.

ESTRUCTURAS

Las estructuras son colecciones de elementos, los cuales pueden ser de diferente tipo, y a diferencia de las uniones, en éstas no se comparte la misma área de memoria para los elementos.

La forma de declarar una estructura es más o menos como sigue:

```
struct nombre_estructura{
        tipo nombre_elemento1;
        tipo nombre_elemento2;
};
```

Es entonces cuando podemos declarar una variable del tipo de la estructura, funcionando el nombre de la estructura como un nuevo tipo de datos, de esta manera, la nueva variable tendrá cada uno de los elementos declarados en ella.

```
nombre_estructura     nueva_variable;
```

Para acceder a cada uno de sus elementos usamos el operador punto (.), seguido del nombre del nombre del elemento, de la misma forma que con las uniones.

Una estructura puede ser declarada de forma global o local, pero es más común hacerlo fuera de toda función.

El siguiente es un ejemplo de su uso.

```
#include<iostream.h>
#include<math.h>
struct proyectil{
        float VelocidadInicial, angulo;
        float PosX, PosY;
};
float calculaCaida(proyectil cosa);
int main(){
        proyectil misil;
        misil.PosX=misil.PosY=0;
        float distancia;
        cout<<"Este programa calculará la distancia, en linea\
             recta,en la que caerá un proyectil"<<endl;
        cout<<"escribe la velocidad inicial para lanzarlo: ";
        cin>>misil.VelocidadInicial;
        cout<<"escribe ahora el angulo de lanzamiento: ";
        cin>>misil.angulo;
        distancia=calculaCaida(misil);
        cout<<"la distancia a la que caerá el proyectil es: "
                <<distancia<<" metros aproximadamente"<<endl;
        cin.ignore();
        cin.get();
}
float calculaCaida(proyectil cosa){
        int t=1;                //tiempo
        const float grav=9.81;  //gravedad
        cosa.angulo/=57.2957;   //transformamos a radianes
        do{
          cosa.PosX=(cosa.VelocidadInicial*cos(cosa.angulo)*t);
          cosa.PosY=(cosa.VelocidadInicial*sin(cosa.angulo)*t)\
                -(0.5*grav*t*t);
             ++t;
        }while(cosa.PosY>0);
        return(cosa.PosX);
}
```

Debemos hacer notar que las estructuras no se pueden inicializar en su declaración.

Aunque las estructuras son una buena manera de mandar bloques de datos a una función, y es posible regresar estructuras (por la instrucción `return`), el coste es

alto, el programa se hace lento, gasta CPU y memoria. Para ello, y gracias a que se trata como un nuevo tipo de variable, es posible hacer una paso por referencia, así evitaríamos que se hiciese una copia de toda la estructura, y sólo se pasara la dirección.

La forma de hacer un paso por referencia es igual que con cualquier otro tipo de variables, sin embargo, al accesar a sus miembros debemos de hacerlo por medio del operador flecha (->). Usando nuestro ejemplo anterior, simplemente cambiarían unas lineas:

```
.
.
cosa->angulo/=57.2957;  //transformamos a radianes
do{
 cosa->PosX=(cosa->VelocidadInicial*cos(cosa->angulo)*t);
 cosa->PosY=(cosa->VelocidadInicial*sin(cosa->angulo)*t)\
       -(0.5*grav*t*t);
   ++t;
}while(cosa->PosY>0);
return(cosa->PosX);
.
.
```

Y gracias a las nuevas referencias de C++ que describimos en el capítulo anterior, podemos simplemente incluir un carácter más en nuestro código para utilizarlo como referencia, en nuestra función (y la declaración por supuesto) incluimos un '&':

```
float calculaCaida(proyectil& cosa){
.
.
```

y bastará para que se trate como una referencia sin necesidad de usar el operador flecha.

Las estructuras también pueden tener por miembros a funciones, en cuyo caso se invoca de la misma forma en la que se accesa a una variable. Le sugiero animarse a usarlas, intente hacer unos ejercicios con ellas. Sin embargo, para esos casos es mejor y más útil utilizar otro tipo de datos definido en C++, se trata de las clases (class), para los fines de este trabajo las clases quedan fuera, debido a que se requiere describir y practicar bastante con la programación orientada a objetos. El lector debe de tener en cuenta su existencia, para que en un futuro, si se está interesado, se adentre más en el mundo de la programación, en una de mejor nivel, con mejores técnicas, y por ende con más capacidades.

Aquí muestro un programa en donde se observa la utilización de funciones como miembros de una estructura, y también un ejemplo de las referencias C++.

```
#include<iostream.h>
#include<iomanip.h>
const int NumMaterias=11;
struct alumno {
        int boleta, semestre;
        int calificaciones[NumMaterias];
        void constancia(alumno & este){
                cout<<"constancia impresa"<<endl;
                cout<<"alumno con boleta: "<<este.boleta;
                cout<<setw(50)<<"del semestre numero: "
                        <<este.semestre<<endl;
                cout<<endl<<"CALIFICACIONES DEL SEMESTRE: "
                        <<endl;
                for(int i=0;i<NumMaterias;++i){
                        cout<<"asignatura "<<i+1<<" : "
                                <<este.calificaciones[i]<<endl;
                }
        }
        void inicia(alumno & este){
                for(int i=0;i<NumMaterias;++i)
                        este.calificaciones[i]=0;
                este.boleta=0;
                este.semestre=0;
        }
};
```

```
int main(){
        alumno alan;
        alan.inicia(alan);
        cout<<"alan creado"<<endl;
        alan.boleta=2005630170;
        alan.semestre=5;
        alan.constancia(alan);
        return 0;
}
```

CAPÍTULO 9 ENTRADA Y SALIDA POR ARCHIVOS

La entrada y salida por archivos es uno de los temas más importantes para la programación, es necesario para poder programar desde una pequeña agenda hasta una completa base de datos, pero antes de entrar de lleno a este tema, y aprovechando los conocimientos adquiridos hasta el momento, quiero mostrar el uso de la memoria dinámica.

MEMORIA DINÁMICA

Hasta el momento no contamos con una forma de "administrar" la memoria utilizada en nuestros programas, cuando declaramos una variable se asigna memoria para almacenar datos dentro de ella, y ésta no se destruye hasta que termina el bloque en el que fue declarada, se crea y se destruye cada que se pasa por ese bloque, quizá puede parecer poco importante, pero cuando se cuentan con grandes cantidades de datos, es necesario tener un mejor control de lo que se pone en memoria.

Una de las formas en que podemos asegurarnos que una variable declarada dentro de un bloque no sea borrada al término de éste, es mediante la utilización del calificador `static`. De ésta manera, la variable perdurará hasta el término de todo el programa.

static tipo_variable mi_variable;

Pero en algunos casos esto nos será insuficiente, vamos a necesitar “destruir” variables antes de terminar el programa, para hacer esto contamos con los operadores `new` y `delete`.

Se puede utilizar el operador `new` para crear cualquier tipo de variables, en todos los casos devuelve un puntero a la variable creada. En el momento en que se quiera borrar esta variable deberemos utilizar el operador `delete`, todas las variables creadas mediante el operador `new` deben de ser borradas con el otro operador.

En el siguiente ejemplo utilizaremos estos dos operadores para mostrar sus ventajas.

```
#include<iostream.h>
int main(){
        char *cadena2;
        int fincontador=10;
        cout<<"programa de prueba"<<endl;
        for(int i=0;i<fincontador;++i){
                cout<<"contador funcionando "<<i<<endl;
        }
        int respuesta;
        cout<<"crear variable?, 1 para si";
        cin>>respuesta;
        char cadena[3]="hi";
        if(respuesta==1){
                int LongCad;
                cout<<"variable creada"<<endl;
                cout<<"longitud de cadena: ";
                cin>>LongCad;
                cadena2=new char[LongCad];
                cout<<"escribe: "<<endl;
                cin.ignore();
                cin.getline(cadena2,LongCad);
                cout<<"cadena escrita: "<<cadena2<<endl;
        }else{
                char cadena[10]="rayos, no";
        }
        cout<<cadena<<endl;
```

```
        cout<<cadena2<<endl;
        delete cadena2;
        cin.ignore();
        cin.get();
        return 0;
}
```

Las partes que nos interesan de este programa son la declaración de la variable tipo apuntador que se encuentra al principio del `main`, la asignación de espacio dentro del bloque `if`, y por último la destrucción de nuestra variable lineas antes del término del programa.

Si hubiésemos declarado el apuntador dentro del bloque `if`, cuando se saliera de éste entonces la variable ya no existiría y habría un error en tiempo de compilación.

El uso de la memoria dinámica nos será muy útil en muchos casos, en otros quizá prefiramos hacerlo de la manera más común. Estos operadores funcionan tanto para variables simples como para compuestas (definidas por el programador). De nueva cuenta debo mencionar a las clases, porque en éstas se maneja mucho este concepto, y de nuevo invito al lector a que, terminando esta lectura, se adentre más en el mundo de la programación consultando otra bibliografía.

Si se anima a trabajar con clases lo felicito, antes quisiera darle una idea manejando estructuras con funciones. El siguiente programa es una modificación del último que vimos en el capítulo anterior, la diferencia está en el manejo de los operadores descritos hace unos párrafos, y por ende en el manejo de apuntadores a estructuras.

```
#include<iostream.h>
#include<iomanip.h>
const int NumMaterias=11;
struct alumno {
        int boleta, semestre;
        int calificaciones[NumMaterias];
        void constancia(alumno * este){
```

```
                cout<<"constancia impresa"<<endl;
                cout<<"alumno con boleta: "<<este->boleta;
                cout<<setw(50)<<"del semestre numero: "
                        <<este->semestre<<endl;
                cout<<"\nCALIFICACIONES DEL SEMESTRE: "<<endl;
                for(int i=0;i<NumMaterias;++i){
                        cout<<"asignatura "<<i+1<<" : "
                              <<este->calificaciones[i]<<endl;
                }
        }
        void inicia(alumno * este){
                for(int i=0;i<NumMaterias;++i)
                        este->calificaciones[i]=0;
                este->boleta=0;
                este->semestre=0;
        }
};
int main(){
        alumno *alan =new alumno;
        alan->inicia(alan);
        cout<<"alan creado"<<endl;
        alan->boleta=2005630170;
        alan->semestre=5;
        alan->constancia(alan);
        delete alan;
        alumno *alfonse=new alumno;
        alfonse->inicia(alfonse);
        alfonse->constancia(alfonse);
        delete alfonse;
        alan->constancia(alan);
        cin.get();
        return 0;
}
```

Como en el ejemplo anterior, declaramos una variable de tipo apuntador (apuntador a alumno) y luego le asignamos espacio en memoria. Cuando se invoca a la función miembro “inicia”, se envía como argumento el nombre del apuntador, y como este contiene una dirección de memoria, pues entonces la función que lo recibe esta preparada para recibirlo y manejarlo adecuadamente con el operador flecha.

Note que después de la destrucción de la variable “alan” se crea otra de nombre “alfonse” que también se destruye, y posteriormente se invoca de nuevo a la función “constancia” de “alan”, dependiendo del compilador utilizado puede haber distintos resultados, pero es lógico que no se obtendrá lo que deseamos porque esa variable ya no existe.

ARCHIVOS

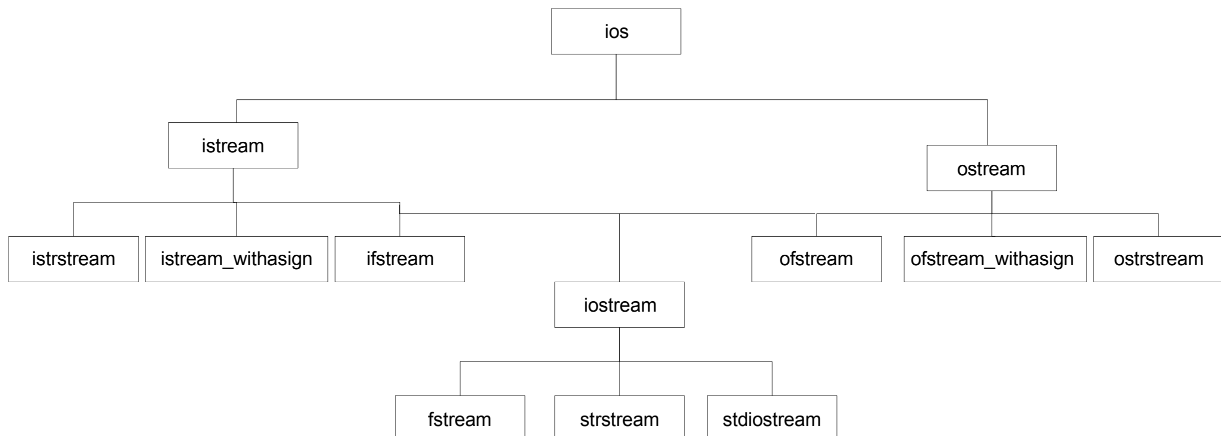
Para empezar con el manejo de archivos es necesario recordar el concepto de flujo, el cual se define como un dispositivo que consume o produce información. En nuestros programas hechos hasta el momento hemos utilizado los flujos estándar `cin`, `cout` y `cerr`, el resto de los flujos que se deseen deberán ser creados por el programador. Todos los flujos se comportan de forma análoga, independientemente del dispositivo que se trate.

Para poder usar un flujo estándar basta con incluir la biblioteca `iostream.h` como lo hemos hecho hasta ahora. Cuando decidimos utilizar la función `cin.get()` no sabíamos exactamente el porque de la utilización del punto(.), ahora que hemos visto un poco el concepto de estructura podemos decir que se trata de la invocación a una función miembro del flujo `cin`. Sí, el flujo es un objeto, viéndolo desde la perspectiva de las estructuras y no precisamente de las clases como debería de ser, se trata de un tipo de dato que contiene variables y funciones que pueden ser invocadas.

A lo largo de este capítulo hablaremos de los flujos como estructuras, y no como clases para tener una mejor idea de lo que se tratan.

Las estructuras (en realidad clases) para la entrada y salida de datos tienen un orden jerárquico, en ellas existe la herencia que básicamente consiste en que una de orden

inferior obtiene las mismas variables y funciones que la de orden mayor, además de que puede agregar más.



Para poder trabajar los ficheros como flujos es necesario incluir la librería `fstream.h`, y según la utilización que queramos dar a este fichero (lectura o escritura) deberemos declarar el tipo de flujo.

Para crear un archivo de salida declaramos una variable de tipo `ofstream`, el cual ya está declarado dentro de nuestra librería. Es mejor ver un ejemplo de base.

```
#include<fstream.h>
int main(){
        ofstream archivo;
        archivo.open("miarchivo.txt");
        archivo<<"hola desde este archivo\n";
        archivo<<"ya he escrito algo\n";
        archivo.close();
}
```

Aquí declaramos a “archivo” como variable tipo ofstream, y posteriormente utilizamos su función miembro open para asociarla a un archivo, se pude asociar directamente en la declaración de la siguiente manera:

```
ofstream archivo(“miarchivo.txt”);
```

Tanto la primera como la segunda forma admiten un segundo argumento que especifica el modo de apertura de un archivo. Los modos disponibles se muestran en la siguiente tabla y pueden ser utilizados incluyendo la librería `iostream.h`.

ios::app	Se escribe al final de archivo
ios::out	El archivo se abre para escritura
ios::trunc	Si el archivo existe se eliminará su contenido
ios::in	El archivo se abre para lectura, el archivo original no será modificado
ios::binary	El archivo se abre en modo binario

Tabla 12 Modos de apertura de archivo

Con el archivo creado en el ejemplo anterior utilizaremos el siguiente programa para escribir al final de él.

```
#include<iostream.h>
#include<fstream.h>
int main(){
        ofstream archivo("miarchivo.txt", ios::app);
        archivo<<"hola desde este archivo de nuevo\n";
        archivo<<"ya he escrito algo de nuevo\n";
        archivo.close();
}
```

El método para abrir un archivo en modo lectura es muy similar, pero en este caso utilizaremos `ifstream`. Para tener el control del fichero, aparte de conocer los modos de apertura de un archivo, debemos de conocer el delimitador, así como en las cadenas existe el carácter de fin de cadena('\0'), en los archivos está el fin de archivo (EOF).

El siguiente programa lee caracteres de un archivo y los imprime en pantalla hasta llegar al fin de éste.

```
#include<iostream.h>
#include<fstream.h>
int main(){
        char caracter;
        ifstream archivo("miarchivo.txt", ios::in);
        while(!archivo.eof()){
                archivo.get(caracter);
                cout<<caracter;
        }
        archivo.close();
}
```

El programa abre el archivo en modo de lectura, inmediatamente el indicador de posición se coloca en el primer caracter del flujo (el archivo), la función `eof()` devuelve verdadero en caso de que en la posición en la que está el indicador esté el fin de archivo, nosotros hacemos una comprobación de ésto, y mientras (bucle `while`) no se llegue al final de archivo se leerá un caracter de éste flujo, al hacer esto el indicador de posición avanzará, posteriormente se imprime el caracter leído en pantalla y continua el ciclo.

¿Que pasaría si el archivo no existiese?, el programa entraría en un ciclo infinito, por eso debemos de asegurarnos de que el flujo se ha creado bien.

```
#include<iostream.h>
#include<fstream.h>
int main(){
        char caracter;
        ifstream archivo("miarchivo2.txt", ios::in);
        if(archivo){
                while(!archivo.eof()){
                        archivo.get(caracter);
                        cout<<caracter;
                }
                archivo.close();
        }else{
                cerr<<"el archivo no existe"<<endl;;
                return 1;
```

```
        }
        return 0;
}
```

Para el manejo de caracteres desde un archivo podemos utilizar las funciones miembro `get`, `getline`, `read`, `write`, `ignore`, `gcount` con las que ya tenemos un poco de experiencia.

Además de éstas, existen otras funciones que nos serán muy útiles para que no sea tan secuencial la forma en la que leemos o escribimos el archivo.

tellg()	Devuelve la posición de lectura
seekg()	Se coloca dentro del flujo en la posición pasada como argumento para poder leer.
tellp()	Devuelve la posición de escritura
seekp()	Se coloca dentro del flujo en la posición pasada como argumento para poder escribir.

Ya sabrá de la importancia de los archivos, pero quizá piensa en cosas demasiado grandes, pensar en una base de datos aveces puede causar que nuestro interés se deteriore con el avance del proyecto, esto claro, si no se tiene claramente definido el objetivo de ésta. Para practicar con el manejo de archivos piense en algo pequeño, algo que le llame la atención, vuelva a recordar que con cosas pequeñas se construyen cosas más grandes.

Hasta aquí el tema de manejo de archivos creo que nos ha dado las suficientes herramientas para desarrollar cosas mejores.

¡¡Ohh, la computadora tiene un espíritu dentro!!!, quizá somos muchas las personas que hemos visto a alguien decir eso, en el bachillerato llegué a ver varias personas un poco asustadas por ello. Para practicar un poco más, y utilizar algunas funciones vistas hasta ahora le muestro el siguiente programa.

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
int main(){
        char texto_preg[300],texto_resp[300],*compara=NULL;
        ifstream archivo("arch.txt", ios::in);
        if(!archivo){
                 cerr<<"error al abrir el archivo";
                return 1;
        }
        cout<<"* hola"<<endl;
        cout<<"> ";
        cin.getline(texto_resp,290);
        while(texto_resp[0]!='\0'){
        archivo.seekg(0);
        do{
                archivo.getline(texto_preg,290);
                compara=strstr(texto_resp,texto_preg);
                if(archivo.eof()){
             cout<<"* no se contestar, dimelo por favor"<<endl;
                        cin.getline(texto_preg,290);
                        archivo.close();
                        ofstream archivoS("arch.txt",ios::app);
                        archivoS<<texto_resp<<endl;
                        archivoS<<texto_preg<<endl;
                        archivoS.close();
                        archivo.open("arch.txt", ios::in);
                }
        }while(compara==NULL);
        cout<<"> ";
        archivo.getline(texto_preg,290);
        cout<<"* "<<texto_preg<<endl;
        cout<<"> ";
        cin.getline(texto_resp,290);
        }
        archivo.close();
        return 0;
}
```

Se trata de un simple programa de conversación, donde el elemento principal es el archivo “arch.txt”, el cual contiene una conversación en él:

```
hola
como estas?
bien
que haces?
nada
pues ponte a hacer algo, a que te dedicas?
estudio
y que estudias?
informatica
ha que bueno!!, de donde eres?
ixtapaluca
que interesante, hay algo que quieras preguntarme?
como te llamas?
acaso eso importa?
como te llamas
pregunta de nuevo
```

El programa inicia abriendo el archivo en modo lectura, si existe un error el programa termina. Empieza su conversación diciendo “hola”, y espera a que el usuario responda, atrapa su respuesta y la guarda en una cadena, si tecleó algo que no fuera simplemente un enter entonces continua dentro del bucle.

Se posiciona al inicio del archivo, y, dentro de otro bucle, lee la cadena de caracteres del archivo con un máximo de 290 caracteres y la almacena en “texto_preg”. Utilizando la función `strstr()` de la biblioteca `string.h`, compara la cadena introducida por el usuario con la cadena que acaba de leer del archivo, la función `strstr()` devuelve un apuntador a la primera localización de la cadena “texto_preg” en “texto_resp”, si no se encuentra devuelve NULL.

Si se ha llegado al final del archivo, es decir, no se encontró que contestar, pide a la persona que le diga que debe de contestar, entonces cierra el archivo y lo vuelve a abrir pero esta vez en modo de escritura para añadir la pregunta del usuario con su respectiva respuesta, luego cierra el archivo de salida y lo abre en modo de lectura de nuevo.

Estos pasos de buscar y, si no encuentra entonces preguntar, continuarán mientras no se encuentren coincidencias entre lo escrito y lo que está dentro del archivo.

Después, continuando con el bucle de mayor nivel, vuelve a leer del archivo una cadena de caracteres, y empieza la conversación de nuevo. Para terminar el programa simplemente el usuario debe teclear enter sin ningún otro caracter.

Como ya se habrá dado cuenta, la parte principal de este programa es el archivo, en él está la base de la conversación, y si se quiere una plática más amena pues hay que modificar el archivo y aumentar su repertorio de respuestas. Note que al final del archivo hay una linea en blanco para el correcto funcionamiento del programa.

APÉNDICE

BIBLIOTECA ESTÁNDAR DE C++

Seguramente en el compilador que ha estado utilizando, sea Visual C++, Dev C++ o cualquier otro que no sea muy viejo, es decir anterior al año de 1998, le ha estado marcando algunos "Warnings" a la hora de compilar su código fuente, trate de corregirlos. Uno de ellos que quizá no sepa a que se refiere es:

…#warning This file includes at least one deprecated or antiquated header... To disable this warning use -Wno-deprecated.

Este warning está presente en el compilador GCC, y aparece debido a los archivos de cabecera que hemos empleado hasta este momento.

En el estándar de C++ las bibliotecas dejan de utilizar el ".h", es decir que en lugar de utilizar `<iostream.h>`, ahora tendrá que escribirse `<iostream>`, en vez de `<iomanip.h>` será `<iomanip>`.

A pesar que esto tiene varias ventajas, que tal vez no vea en este momento, se tendría que colocar "std" antes de cada llamada a una función de la biblioteca, debido a que toda la biblioteca estándar de C++ está definida en el espacio de nombres llamado `std`. Con esto, tendríamos que hacer una llamada a escribir en pantalla con la instrucción `std::cout`.

Para evitar esto, y por consiguiente hacer menos extenso el código a escribir, podemos emplear la directriz `using`, indicando al compilador el espacio de nombres donde está referenciado.

Por ejemplo, el siguiente programa es la nueva versión de uno de nuestros primeros programas escritos.

```
#include <iostream>
using namespace std;
int main(){
     cout<< “adios a todos” <<endl;
     return 0;
}
```

Para tener una idea de las capacidades aportadas por la biblioteca estándar, presentamos la siguiente tabla en donde se clasifican las librerías deacuerdo a su funcionalidad.

ENTRADA/SALIDA	
<cstdio>	E/S de la biblioteca de C
<cstdlib>	Funciones de clasificación de caracteres
<cwchar>	E/S de caracteres extendidos
<fstream>	Flujos para trabajo con ficheros en disco
<iomanip>	Manipuladores
<ios>	Tipos y funciones básicos de E/S
<iosfwd>	Declaraciones adelantadas de utilidades de E/S
<iostream>	Objetos y operaciones sobre flujos estándar de E/S
<istream>	Objetos y operaciones sobre flujos de entrada
<ostream>	Objetos y operaciones sobre flujos de salida
<sstream>	Flujos para trabajar con cadenas de caracteres
<streambuf>	Búferes de flujos
CADENAS	
<cctype>	Examinar y convertir caracteres
<cstdlib>	Funciones de cadena estilo C
<cstring>	Funciones de cadena estilo C
<cwchar>	Funciones de cadena de caracteres extendidos estilo C

<cwctype>	Clasificación de caracteres extendidos
<string>	Clases para manipular cadenas de caracteres
CONTENEDORES	
<bitset>	Matriz de bits
<deque>	Cola de dos extremos de elementos de tipo T
<list>	Lista doblemente enlazada de elementos de tipo T
<map>	Matriz asociativa de elementos de tipo T
<queue>	Cola de elementos de tipo T
<set>	Conjunto de elementos de tipo T (contenedor asociativo)
<stack>	Pila de elementos de tipo T
<vector>	Matriz de elementos de tipo T
ITERADORES	
<iterator>	Soporte para iteradores
ALGORITMOS	
<algorithm>	Algoritmos generales (buscar, ordenar, contar, etc.)
<cstdlib>	bsearch y qsort
NÚMEROS	
<cmath>	Funciones matemáticas
<complex>	Operaciones con números complejos
<cstdlib>	Números aleatorios estilo C
<numeric>	Algoritmos numéricos generalizados
<valarray>	Operaciones con matrices numéricas
DIAGNÓSTICOS	
<cassert>	Macro ASSERT
<cerrno>	Tratamiento de errores estilo C
<exception>	Clase base para todas las excepciones
<stdexcept>	Clases estándar utilizadas para manipular excepciones
UTILIDADES GENERALES	
<ctime>	Fecha y hora estilo C
<functional>	Objetos función
<memory>	Funciones para manipular bloques de memoria

<utility>	Manipular pares de objetos
LOCALIZACIÓN	
<clocale>	Control estilo C de las diferencias culturales
<locale>	Control de las diferencias culturales
SOPORTE DEL LENGUAJE	
<cfloat>	Limites numéricos en coma flotante de estilo C
<climits>	Limites numéricos estilo C
<csetjmp>	Salvar y restaurar el estado de la pila
<csignal>	Establecimiento de manejadores para condiciones excepcionales (también conocidos como señales)
<cstdarg>	Lista de parámetros de función de longitud variable
<cstddef>	Soporte de la biblioteca al lenguaje C
<cstdlib>	Definición de funciones, variables y tipos comunes
<ctime>	Manipulación de la hora y fecha
<exception>	Tratamiento de excepciones
<limits>	Limites numéricos
<new>	Gestión de memoria dinámica>
<typeinfo>	Identificadores de tipos durante la ejecución

Biblioteca estándar de C++[5]

[5] Francisco Javier Ceballos, “Enciclopedia del lenguaje C++”.

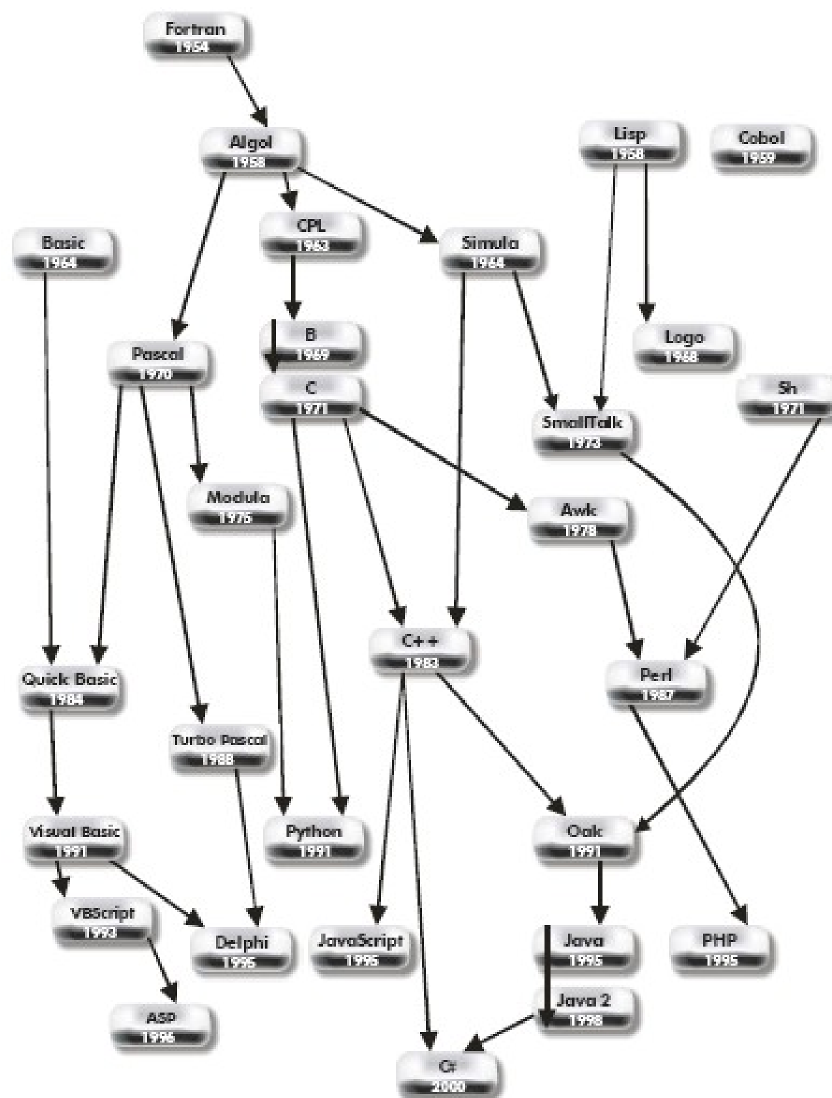


Ilustración 1: Evolución de algunos lenguajes

RECOMENDACIONES Y/O SUGERENCIAS

La forma en la que he desarrollado este manual no permite del todo consultar una parte sin haber leído una anterior. En algunas ocasiones encontrará comentarios o referencias a ejemplos vistos en una parte precedente al texto en cuestión.

Por eso recomiendo que, con el interés de aprender el lenguaje de programación C++, siga este manual de principio a fin, tal y como si se tratara de un tutorial. Por supuesto, para su mejor aprovechamiento es sugerente consultar otra bibliografía, así como la ayuda proporcionada con el compilador que utilice para hacer los ejemplos.

A una persona que ya tenga conocimientos de este lenguaje puede resultarle más rápida su consulta si presta especial atención a las tablas proporcionadas y a los ejemplos mostrados, para ver parámetros, definiciones, declaraciones, e implementación del código.

¿Qué es lo que sigue?

Al finalizar la lectura y estudio de este manual se contará con los elementos para desarrollar programas estructurados. La principal característica de C++ es que es un lenguaje orientado a objetos, es por eso que, si se tiene el interés de tener un mejor nivel de programación y hacer cosas mejores, es necesario que se estudie la programación orientada a objetos.

El lenguaje C++ ha sido una base para otros lenguajes de programación, es por ello que encontramos varias similitudes con otros. Y, siendo un lenguaje vigente para el desarrollo de aplicaciones, resulta muy útil aprenderlo. Sin embargo, considero que no es absolutamente necesario aprender a programar con objetos en C++ antes que en cualquier otro lenguaje, de hecho, personalmente recomiendo adentrarse desde

ahora en el lenguaje Java, con el cual, además de que proporciona varias ventajas que en otros lenguajes difícilmente tenemos (como que los punteros no existen), se ha convertido en uno de los lenguajes con más amplia "cobertura", refiriéndome con esto a que las implementaciones de este lenguaje prácticamente no tienen límites, pudiendo encontrar aplicaciones desarrolladas con él en cualquier lado.

BIBLIOGRAFÍA

Deitel, Harvey M.; Deitel, Paul J.
C++, cómo programar
Edit. Pearson Educación.
4ª ed., 2003. 1320 pp

Sánchez, Jorge.
Java 2
www.jorgesanchez.net
2004, 312pp.

Schildt, Herbert.
C, Manual de referencia
Edit. Osborne McGraw-Hill.
4ª ed., 2000. 709 pp.

Sierra Urrecho, Alejandro; Alfonseca Moreno, Manuel
Programación en C/C++. Edit. Anaya Multimedia.
Madrid, España, 1999. 351 pp.

Stevens, Al; Walnum, Clayton.
Programación con C++. Edit. Anaya Multimedia.
Madrid, España, 2000. 832 pp.