Programación para físicos

Juan Carlos Córdova, Enrique Pazos

21 de febrero de 2012

Índice general

Experimento #1:

Preparando un programa

Objetivos

- Comprender los pasos involucrados en la creación de un programa.
- Escribir, guardar, compilar y ejecutar un programa sencillo.

Descripción

En este experimento se desea escribir sobre la pantalla de la computadora una frase sencilla, "Hola mundo!". Si bien esto no resuelve ninguna pregunta trascendental, sí ejemplifica el mecanismo de producción de cualquier programa usando una plataforma dada.

Procedimiento

Paso 1 Utilizando un editor adecuado, escriba el siguiente programa y guardelo como un archivo de tipo ASCII (American Standard Code for Information Interchange) que termine con una extensión ".c". (Por ejemplo, el nombre del archivo podría ser "hola.c")

```
#include <stdio.h>
/* Este programa envia un saludo
al mundo */

void main()
{
   printf("Hola mundo!\n");
}
```

Paso 2 Compile el programa. El procedimiento específico depende de la plataforma que se esté utilizando. En Linux o en Unix existen varias posibilidades, de las cuales dos se muestran a continuación:

```
cc hola.c -o hola  <Enter>
```

o

```
gcc hola.c -o hola  <Enter>
```

Paso 3 Ahora ejecute el programa:

```
hola <Enter>
```

Es posible que necesite escribir

```
./hola <Enter>
```

Paso 4 Si nos interesara guardar los resultados del programa en un archivo, digamos `hola_out.txt`, podríamos usar el comando de redirección de Unix o Linux de la siguiente forma:

```
./hola > hola_out.txt <Enter>
```

De esta manera se ejecuta el programa y la salida (si hay alguna) va a dar al archivo llamado `hola_out.txt`. Para verificar el contenido de este archivo puede usarse el comando cat de Unix (type en DOS), o se podría abrir el archivo usando un editor de texto convencional.

En el caso de usar el comando cat:

```
cat hola_out.txt <Enter>
```

esto desplegaría, sobre la pantalla del monitor, el contenido del archivo.

Impresión

1. Imprima el programa.
2. Imprima la salida del programa.

Preguntas

1. El símbolo `#` se utiliza en C para indicarle al compilador que haga algo antes de proceder a compilar el programa. Así `#` introduce una directiva al compilador. En particular, cuál es la función de la directiva `#include`?
2. Qué diferencia hay entre `#include <stdio.h>` e `#include "stdio.h"`?
3. Deben las directivas terminarse con un ";"?
4. Qué funciones están definidas en el encabezado stdio.h?
5. Explique el uso del comando printf().

6. En el comando de linea `cc hola.c -o hola`, cuál es el objeto de la opción `-o`? Y si hubiese una opción `-l`?

7. Investigue el propósito y uso de los siguientes operadores, funciones y palabras clave:

int	long	float	double	boolean	char
sizeof()	scanf()	void	return	=	==
¿	¡	!=	!	clock()	

8. Investigue el uso de la construcción `if, elseif, else`.

9. Investigue el uso del iterador `for`.

10. Investigue el uso de `while`.

11. Qué objeto tienen los símbolos `/*` y `*/`?

Experimento #2:

Entrada, salida, tiempo

Objetivos

- Determinar algunos parámetros del compilador de C.
- Aprender a ingresar información del teclado.
- Familiarizarse con algunas funciones de tiempo.
- Aprender a determinar tiempos de porciones específicas de un programa.

Descripción

En este experimento tenemos tres cosas por resolver: determinar algunos parámetros del compilador de C, calcular el cuadrado de un número, determinar el tiempo que le toma a la máquina obtener este resultado.

Aquí se utilizan varias de las funciones investigadas en el experimento anterior, en la sección de preguntas, para determinar algunos parámetros de operación del compilador de C. Estamos interesados en particular en el número de bytes que una variable de los siguientes tipos ocupa: char, int, long, boolean, float, double, y punteros.

A diferencia del programa del Experimento 1, este programa solicita información del teclado, la procesa, e imprime el resultado. De esta manera tenemos un ejemplo sencillo de lo que la mayor parte de programas hacen: tomar información, manipularla, y dar a conocer los resultados de la manipulación.

Además de ésto, estamos interesados en saber en cuánto tiempo se realiza el procesamiento para lo cual haremos uso de la función `clock()` definida en el archivo *time.h*. Esta función da el tiempo en "pulsaciones de reloj", para relacionar esto con segundos se debe saber cuántas "pulsaciones" caben en un segundo. El símbolo `CLOCKS_PER_SEC` definido en *time.h* nos da esta información.

Así, lo que deseamos es un programa que, en la primera parte, imprima el número de bytes ocupados por variables de tipo char, int, long, boolean, float, double, y punteros; que imprima también cuántas pulsaciones de reloj componen un segundo; que ingrese un entero del teclado, lo eleve al cuadrado, imprima el resultado y el tiempo que le toma a la máquina en calcularlo.

Procedimiento

Quizá resulte conveniente dividir este problema en varios programas que resuelvan distintos aspectos que aquí nos competen. Por ejemplo, *parametros.c* podría imprimir los parámetros en los que estamos interesados ya que esto, en sí, no tiene relación directa con el cálculo del cuadrado de un entero.

Otro programa, *granularidad.c*, puede utilizarse para determinar la resolución de la función `clock()`, de esta manera sabríamos cuál es el tiempo mínimo que tenemos esperanza de medir.

Por último podríamos escribir *cuadrado_int.c, cuadrado_long.c, cuadrado_float.c* y *cuadrado_double.c*, que aceptan un número (de tipo int, long, float, y double respectivamente), lo elevan al cuadrado, imprimen el número original, su cuadrado, y el tiempo que le toma (tanto en pulsos de reloj como en segundos) hacer esta operación.

Luego, podríamos proceder como se describe a continuación:

Paso 1 Escriba *parametros.c*, que imprime el número de bytes ocupados por variables de tipo char, int, long, boolean, float, double, y punteros; y también cuántas pulsaciones de reloj componen un segundo; Las principales funciones usadas en este programa son `printf()` y `sizeof()`.

Paso 2 Escriba *granularidad.c* de manera que se pueda determinar el número más pequeño de pulsaciones a las que es sensible la función `clock()`. Esto se puede lograr usando un enunciado `while` que continuamente encuesta `clock()` hasta que encuentra un valor nuevo: la diferencia entre el valor de clock() justo antes de comenzar el `while` y el valor al salir de éste es la granularidad. Imprima ésta.

Paso 3 Escriba cada una de las funciones *cuadrado_int.c, cuadrado_long.c, cuadrado_float.c* y *cuadrado_double.c*, que aceptan un número (de tipo int, long, float, y double respectivamente), lo elevan al cuadrado, imprimen el número original, su cuadrado, y el tiempo que le toma (tanto en pulsos de reloj como en segundos) hacer esta operación.

Para hacer ésto, y en base a la granularidad de la función `clock()` es muy posible que sea necesario repetir la operación de elevar al cuadrado un número suficiente de veces como para que se tomen un tiempo suficientemente grande que la función `clock()` pueda cronometrar. Si la resolución de `clock()` es de por ejemplo 10000 pulsos, repita la operación de elevar al cuadrado (o en general cualquier función que nos interese cronometrar) tantas veces como para requerir un tiempo unas 10 veces mayor, de esta manera el tiempo total empleado es exacto hasta en un 10 %. Luego, al dividir este tiempo por el número de veces que se repitió la operación se obtiene el tiempo que buscamos.

En este último programa, se hará uso de `clock()` para tomar tiempos y de un iterador `for` para repetir la operación que buscamos cronometrar, además

de `scanf()` para obtener el número que deseamos cuadrar y `printf()` para imprimir resultados.

La diferencia entre los programas *cuadrado_XXXX.c* es simplemente el tipo de las variables que contienen al número original y su cuadrado. A grandes rasgos, estos programas tienen la forma:

```
/* Incluir encabezados necesarios */
#include <stdio.h>
etc.

/* Definir el cuerpo del programa
principal */
void main()
{
/* Declarar variables necesarias
para aceptar un numero, para
contener su cuadrado, y
variables para almacenar el
tiempo to antes de iniciar el
procesamiento, y el tiempo tf al
finalizar el mismo. */


/* Pedir informacion al usuario */


/* Marcar el tiempo antes del procesamiento */
to = clock();

/* Repetir el procesamiento un numero apreciable de veces */
for ( )
{




}

/* El tiempo en segundos por operacion es, aproximadamente,
(tf-to)  dividido por CLOCKS_PER_SEC */



/* Imprimir resultados */
}
```

Impresión

1. Imprima cada uno de los programas.
2. Haga una tabla que muestre los parámetros del sistema.
3. Haga una tabla que muestre el tiempo de ejecución para cada programa que eleva al cuadrado un número.

Preguntas

1. Depende el tiempo de elevar al cuadrado del número que se está cuadrando?
2. Depende el tiempo de elevar al cuadrado del tipo de la variable que se esta cuadrando?
3. Cuál es la granularidad de `clock()` en segundos?
4. Dado que la granularidad de `clock()` (en segundos) es G, que una operación O se repitió R veces y que tomó T segundos; cuál es la incerteza D (en segundos) del tiempo de una sola operación O?
5. La iteración `for` toma, de por si, tiempo. Cómo eliminaría ese tiempo para obtener un valor más exacto del tiempo que le toma a la computadora el elevar un número al cuadrado?
6. Qué otros factores pueden introducir incertezas en el tiempo de procesamiento neto calculado de esta manera?
7. Investigar que es una función en C y cómo se declara una.
8. Qué es un prototipo?
9. Qué es recursión? De tres ejemplos de fórmulas recursivas que haya encontrado en física o en matemática.

Experimento #3:

Funciones en C

Objetivos

- Aprender a definir funciones en C.
- Usar prototipos para mejorar la legibilidad de un programa.
- Encapsular en una sola unidad un bloque de procesamiento de uso frecuente.

Descripción

Supongamos que deseamos elevar un número, digamos un entero, a una potencia también entera. Esta es una operación que podríamos encontrar con frecuencia en un programa que resuelva un problema físico. Podríamos, en principio, crear código (un fragmento de programa) que implemente ésto y luego, donde lo necesitaramos, copiarlo y quizá cambiar el nombre de las variables involucradas para corresponder a aquellas que estamos usando en esa parte del programa.

Si bien es cierto que ésto funciona (bien y rápido), consume tiempo de programador el repetir esas líneas de código, hace el programa fuente más largo y, peor aún, introduce la posibilidad de error al no cambiar los nombres de todas las variables por ejemplo. Así que aunque ésta es una posibilidad, debe pesarse cuidadosamente la ventaja en tiempo contra las mencionadas desventajas.

En general, si un fragmento de código puede considerarse como una sola unidad procesadora, y si además es de uso frecuente, conviene más el encapsularla toda en una sola función que haga el procesamiento y regrese el resultado esperado. A esta función se le da un nombre y en el programa principal se le llama por nombre cada vez que se desee realizar el procesamiento contenido en ella.

La función así definida se convierte en una especie de "caja negra" que realiza un trabajo específico. Para poder operar, sin embargo, esta "caja" necesita saber sobre qué ha de operar, y para que el programa que la invocó sepa los resultados la caja ha de proveerlos de alguna forma.

Se llaman parámetros de entrada al conjunto de datos (variables, constantes, etc.) que se le entregan a la función para que ésta opere. La función, a su vez,

entrega sus resultados a través de un enunciado return que se ejecuta justo antes de que la función finalice. El enunciado `return` es seguido inmediatamente por una expresión o variable que constituye un parámetro de salida del tipo apropiado.

Por ejemplo, una función que pretenda elevar una base entera a un exponente no-negativo también entero podría llamarse `elevar`. Para operar, la función precisa de dos datos: la base y el exponente, ambos de tipo entero. Como resultado la función entrega también un número entero. El prototipo de la función (una copia del encabezado de la función) se vería de esta manera:

```
int elevar(int base, int exponente);
```

El prototipo sólo declara a la función. Con ésto el compilador sabe que en alguna parte del programa fuente (o de los encabezados) encontrara la definición de la función, i.e., el encabezado y el cuerpo de la función.

En el caso del prototipo mostrado, éste indica que hay (en alguna parte del programa fuente) una función llamada `elevar`, que acepta dos parámetros de tipo entero y cuyo resultado es un número entero. Lo que está entre paréntesis se llama la lista de argumentos; en este caso base y exponente son argumentos formales. El `int` que antecede el nombre de la función es el tipo de la función, i.e., la clase de dato que regresa la función como resultado de sus cálculos.

La lista de argumentos puede contener cero, uno, o más argumentos. En caso de cero argumentos aún deben usarse parentesis, pero sin nada en ellos o con la palabra reservada `void` indicando la inexistencia de argumentos.

Si la función no regresa nada, el tipo de la función deberá ser `void` sin que exista la posibilidad de omitir esa declaración (algunos compiladores toman la ausencia de tipo como un `void` implícito, pero ésto no está garantizado y no corresponde al estándar ANSI, así que no será permitido en nuestros programas).

Llamaremos parámetros a los valores actuales con los que se invoca (no declara, ni define) la función.

En este experimento queremos un programa que declare (con un prototipo) la función `elevar()` para base entera y exponente entero y no-negativo. El programa además tomará el tiempo de ejecución de esta función y lo imprimirá.

El lenguaje C define en la librería *math.h* la función `pow()` que hace lo mismo que nuestra `elevar()` pero sobre números de tipo real (por extensión también lo hace sobre enteros). Así que también estamos interesados en cronometrar ésta con el propósito de comparar su desempeo contra nuestra función.

Procedimiento

Paso 1 Escriba una función elevar que eleve una base entera a un exponente entero y no-negativo.

Para hacer ésto el siguiente pseudo-código puede ser de utilidad:

```
/* Chequee que el exponente sea valido */
if (Y)
```

```
{
   /* Si no lo es indiquelo y regrese 0 */
   printf(Y);
   return 0;
}

/* Debido al return en el condicional anterior, si llegamos
hasta esta linea es que NO se cumplio el condicional */

/* Declare todas las variables locales que sean necesarias */

/* Prepare el resultado */
resultado = 1;

for (contador entre 1 y exponente)
{
   /* Multiplique por la base */
   resultado = resultado*base;
}

return resultado;
```

Nótese que verificamos que el exponente sea válido. Aunque ésto no es estrictamente necesario, es muy importante hacerlo porque hace a nuestra función robusta: la función opera aún cuando sea usada en forma inapropiada y además nos hace saber del mal uso. Como norma, debemos programar "a prueba de bobos".

Otra cosa importante es la inclusión de comentarios. Aunque no ayudan al procesamiento, los comentarios mejoran la legibilidad del programa y ésto es un gran ventaja.

Ahora puede escribirse en el programa principal todo aquello que necesitamos para tomarle a tiempo a la funcion `elevar()`.

Tome en cuenta que el tiempo de ejecución probablemente dependerá de la base y el exponente escogidos, así que automatíce la elección de éstos en el programa para descubrir si existe tal dependencia.

Paso 2 Escriba un programa, *pow.c*, que cronometre la función `pow()` de C.

Impresión

1. Imprima los dos programas elevar.c y pow.c.
2. Haga una tabla para cada programa que muestre los tiempos de ejecución en función de la base y el exponente escogidos.

Preguntas

1. Depende el tiempo de ejecución de `elevar()` o de `pow()` de la base o del exponente? Si sí, explique cómo. Si no, explique por qué no.

2. Se puede escribir `elevar.c` sin incluir un prototipo para la función `elevar()`? Explique.

3. Si D es el tiempo de ejecución de una multiplicación entera, halle una fórmula en base a la tabla de la sección anterior para el tiempo de ejecución de `elevar()` y `pow()` en función de los parámetros con que se invocan estas funciones.

4. Puede justificarse su resultado en base al algoritmo empleado?

Experimento #4:

Funciones recursivas en C

Objetivos

- Entender qué es recursión y por qué ciertos algoritmos son más fáciles de escribir como funciones recursivas.
- Distinguir entre recursión directa e indirecta.
- Comprender las ventajas y desventajas de una implementación recursiva.

Descripción

Muchos problemas pueden resolverse con el uso de unidades de procesamiento que nosotros llamamos funciones. Estas funciones, a su vez pueden hacer uso de otras funciones que estén definidas en el sistema (ya sea por ser parte integral del lenguaje, o porque fueron explícitamente escritas por el usuario, o porque pertenecen a una librería importada en el programa). Cuando el código dentro de una función eventualmente invoca a la función misma, se dice que la función es recursiva.

Un ejemplo de una función recursiva sería la siguiente:

```
int recursiva_1(int p)
{
   /* Verifique entrada valida */
   if (p<0)
{

printf("Advertencia: recursiva_1 requiere un parametro
no-negativo y entero!\n");
printf("             regresando 0 por entrada invalida.\n");
return 0;
}

/* Chequee caso terminal */
if (p ==0)  return 0;

/* De lo contrario, prosiga con la recursion */
```

```
return p + recursiva_1(p-1)
}
```

Puede verificarse que esta función regresa la suma de los números entre 0 y p. Hay varias cosas qué notar en esta implementación: 1) se asegura que la función trabaje con parámetros válidos; 2) se incluye un caso para el cual la función no se invoca a si misma, a esto se le llama un caso terminal; se especifica el procesamiento en los casos no- terminales.

La inclusión de un caso terminal es de importancia extrema, porque de lo contrario la recursión seguiría por siempre (hasta los límites de memoria de nuestra computadora) y eventualmente el programa colapsaría por agotamiento de recursos.

Para éste programa en particular, `recursiva_1(0)` evalúa a 0 por ser el caso terminal; `recursiva_1(1)` evalúa a $1 +$ `recursiva\_1(0)` $= 1 + 0 = 1$; etc.

Obviamente, el resultado de `recursiva_1(p)` (la suma de los enteros positivos entre 0 y p, podría obtenerse usando un método iterativo:

```
int iterativa_1(int p)
{
   int j, acumulado =0;
   /* Verifique entrada valida */
   if (p<0)
{

printf("Advertencia: recursiva_1 requiere un parametro
no-negativo y entero!\n");
printf("             regresando 0 por entrada invalida.\n");
return 0;
}

/* Realice el calculo por medio de una iteracion */
for (j=0; j<=p; j++)
{
acumulado = acumulado + j;
}

return acumulado;
}
```

De esta forma `iterativa_1()` hace lo mismo que `recursiva_1()`, pero sin emplear recursión. Hay otra forma también bastantante evidente de realizar este cálculo, y es por medio de la fórmula:

$$S = \frac{p(p+1)}{2}$$

Así podríamos escribir:

```
int formula_1(int p)
{
/* Verifique entrada valida */
if (p<0)
{
printf("Advertencia: recursiva_1 requiere un parametro
no-negativo y entero!\n");
printf("             regresando 0 por entrada invalida.\n");
return 0;
}

/* Realice el calculo por medio de una formula */

return (p*(p+1))/2;
}
```

Cuál forma escoger para realizar esta operación? Esto depende en parte de consideraciones de estilo, de eficiencia y de legibilidad. Ciertos problemas son más fáciles de codificar en una de estas tres formas y por lo tanto una primera solución sería adoptar la forma que más rapidamente produzca un programa que resuelva el problema. Luego, guiados quizá por razones de eficiencia, podríamos tratar de implementar la solución no en la forma en que sea más obvia, sino en aquella en que sea más eficiente (en tiempo o en espacio).

Por ejemplo, de las tres funciones que calculan la suma de enteros entre 0 y p que hemos descrito, la última es la más rápida y más compacta, pero no es tan obvia (por eso incluiríamos un comentario al principio de la función indicando qué hace). La versión iterativa es obvia, pero ineficiente. La recursiva es más críptica y quizá la más ineficiente de las tres.

Sin embargo, las soluciones recursivas son muchas veces la única forma "evidente" de proceder y poseen cierta elegancia intrínseca. Toda función recursiva puede expresarse como una solución iterativa, generalmente más eficiente, aunque el proceso de conversión no es obvio ni está sistematizado.

Descripción

En este experimento se desea codificar una version recursiva de la función `elevar()` del Experimento #3. Esta versión pretende reducir el número de multiplicaciones para aumentar la velocidad de procesamiento.

Llamaremos a la versión recursiva `elevar_r()` y, al igual que `elevar()`, entrega un número entero como resultado y toma dos enteros (una base y un exponente no-negativo) como parámetros.

La función iterativa `elevar()` realiza un número de multiplicaciones igual al exponente. Esto puede mejorarse de la siguiente manera:

```
int elevar_r(int base, int exponente)
{
```

```
   /* Chequear validez de los argumentos */


    /* Examinar el exponente */

   /* Si es par, calcular base^(exponente/2) y regresar este numero al
   cuadrado. Esto debe emplear un numero de multiplicaciones menor o
   igual que exponente/2+1 */

   /* Si es impar, calcular base^(Parte Entera de (exponente/2)) y regresar
   este numero al cuadrado multiplicado por base. Esto requiere de un
   numero de multiplicaciones menor o igual a exponente/2+2 */
}
```

El caso terminal, por supuesto, es cuando el exponente es 0 y el resultado, por lo tanto, es 1.

Impresión

1. Imprima el programa que incluye `elevar_r()`.
2. Imprima una tabla que compare los tiempos de ejecución de esta función y los de sus contrapartes `elevar()` y `pow()`.

Preguntas

1. Qué hace su programa si el exponente es un número negativo?
2. Qué hace su programa si el exponente es un número real?
3. Concuerdan los tiempos de su tabla con lo que esperaría dada la reducción en el número de operaciones?
4. Cómo podría averiguar cuánto tiempo le toma a su computadora el llamar a una función recursiva (el tiempo que le toma en guardar datos en la pila, ejecutar el salto a donde está la función y el regresar)?
5. Es este tiempo distinto para funciones iterativas o una función en general?

Experimento #5:

Estructuras y funciones relacionadas

Objetivos

- Aprender cómo un lenguaje puede aumentarse con la definición de nuevos tipos creados por el usuario.
- Entender la diferencia entre funciones de acceso y de procesamiento.
- Comprender en qué consiste el sobrecargar una función.

Descripción

En este experimento crearemos un nuevo tipo en C, un tipo definido por el usuario, que nos permita declarar variables que representen números complejos. Asociado a este nuevo tipo escribiremos funciones que nos permitan manipular nuestras variables complejas.

No existe en C ningún tipo Complejo, i.e., un tipo que contenga una parte real y otra imaginaria. Sin embargo, en física, es muy común trabajar con variables o funciones cuyos valores puedan ser cantidades complejas.

Una opción para llevar cuenta de cantidades complejas sería declarar variables por pares asociados. Así una variable compleja A podría declararse en C realmente como dos variables de tipo double (o float), digamos Ax y Ay, una conteniendo la parte real y otra la imaginaria de A. Si bien ésto es posible, no sólo es engorroso sino también propende a la equivocación.

Una mejor manera de resolver el problema es definir una estructura por medio de la palabra clave `struct` y luego declarar variables del tipo recién creado. Por ejemplo:

```
struct Complejo
{
   double x; /* un tipo real para
      contener la parte real del numero
      complejo  */
   double y; /* un tipo real para
      contener la parte imaginaria del
```

```
      numero    */
};
```

Ahora es posible declarar variables de este tipo:

```
Complejo A;
```

La estructura `Complejo` (y por lo tanto las variables que se declaren de este tipo) tiene dos campos: x e y. Estos campos pueden accederse por medio del operador de acceso de campo: el punto.

Por ejemplo, la parte real de A puede modificarse de la siguiente forma:

```
A.x = 5.3;
```

O podría consultarse de la misma manera, por ejemplo en un enunciado como el siguiente:

```
printf("La parte real es %f y la parte imaginaria es %f\n",A.x, A.y);
```

Esto es un acceso directo a los campos de la variable. La variable pudo inicializarse al declararla:

```
Complejo B={1.0,2.0};
```

crearía una variable B de tipo `Complejo` cuya parte real sería inicializada a 1.0 y la imaginaria a 2.0.

Los campos de una estructura sólo pueden inicializarse al declarar una variable. Sería incorrecto intentar lo siguiente:

```
struct Complejo
{
   double x = 0.0;
   double y = 0.0;
};
```

de hacerlo el compilador reportaría un error sintáctico y el programa no compilaría.

En el caso de números complejos tenemos también la opción de representarlos como una combinación de magnitud y argumento. Podríamos crear un nuevo tipo, quizá llamado `CPolar`, pero sería muy similar a `Complejo` y posiblemente innecesario. Una opción es usar los mismos campos x e y para representar o las componentes rectangulares de un complejo, o su magnitud y ángulo. Sin embargo, de hacer ésto, es necesario informar a quien esté usando la variable (el programa principal, una función dentro del programa, el programador mismo) de qué representación se trata. Para ello agregamos un campo más que nos indica ésto.

```
struct Complejo
{
   char r; /* r == 0 implica representacion rectangular*/
               /* r == 1 implica  representacion polar */

   double x; /*un tipo real para contener la parte real o la magnitud*/
   double y; /*un tipo real para contener la parte imaginaria o el
                  argumento */
};
```

Así una función puede verificar el significado de x e y en una variable consultando primero el campo r de la variable. Si bien es cierto que los campos de estructuras son accesible directamente, hay razones para evitar el acceso directo. En cambio se crea un grupo de funciones de acceso que son las únicas que manipulan directamente los campos de una variable. Otras funciones, llamadas funciones de procesamiento son las encargadas de procesar las variables, pero cuando necesitan un campo específico recurren a las funciones de acceso.

De esta manera se gana una abstracción respecto a la forma particular en la que un tipo se representa en C. Luego, si el programador decide que existe una forma más rápida o eficiente de representar el tipo puede ir y modificarlo a conveniencia, modificar las funciones de acceso y nada mas. Ya que todas las demás funciones hacen uso de las funciones de acceso cuando requieren información el cambio de la forma del tipo no las afecta y les resulta transparente. Las particularidades de la implementación del tipo están escondidas para las funciones de procesamiento. Esto permite que el programador de estas funciones se desentienda de aspectos muy particulares y se concentre en los aspectos de alto nivel (algoritmos para elevar a una potencia en vez de detalles de a qué parte de la memoria va ir a dar un resultado, por ejemplo).

Las funciones de acceso se dividen en funciones de consulta y funciones de modificación. Obviamente las primeras reproducen el valor de algún campo de la variable y las últimas modifican campos dentro de la variable.

Por ejemplo, en el caso de nuestros números complejos podríamos consultar de qué representación se trata, la parte real del número, la imaginaria, la magnitud, o el argumento.

```
bool EsPolar(Complejo X)
{
   /* Regresa verdadero si la variable X esta en representacion
      polar y falso de lo contrario */

if (X.r == 1) return true;
   return false;
}
```

Esto podría también haberse logrado con un simple `return X.r`; en vez del condicional.

Si ya existe una función que accede a un campo en particular es buena práctica de programación dejar que esa función sea la única que acceda a ese campo y otras funciones (aún si son de acceso) que consulten ese mismo campo lo hagan a través de esa función.

Por ejemplo, aunque podríamos escribir

```
bool EsRect(Complejo X)
{
   /* Regresa verdadero si la variable X esta en representacion
      rectangular y falso de lo contrario */

   if (X.r == 0) return true;
      return false;
}
```

Es mejor escribir:

```
bool EsRect(Complejo X)
{
   /* Regresa verdadero si la variable X esta en representacion
      rectangular y falso de lo contrario */

   return !EsPolar(X);
}
```

Donde el `!` es el operador de negación lógica. De esta manera no hay dos funciones consultando el mismo campo para realizar su trabajo.

Estas reglas de "buena programación" introducen robustez, legibilidad, y orden al programa, pero a costo de un incremento en el tiempo de procesamiento. Sin embargo, para la mayor parte de aplicaciones este incremento es despreciable y es sólo en casos de muy alto rendimiento (procesamiento en tiempo real de señales como ejemplo) es admisible el violarlas en favor de mayor velocidad de procesamiento.

Sería común modificar la parte real, la parte imaginaria, convertir de rectangular a polar o viceversa, etc., y ésto sería realizado por funciones de modificación:

```
Complejo Polar(Complejo X)
{
   /* Convierte de rectangular a polar */
   if (EsPolar)
   {
      /* Nada que hacer, ya es polar */
      return X;
   }

   /* Si el programa llega a este punto es que la condicion anterior
```

```
      no se cumplio */

   Complejo T; /* Definir una variable temporal */
   T.r = 1; /* Indicar que T sera polar */

   T.x = sqrt(X.x*X.x+X.y*X.y);
   T.y = atan2(X.y/X.x); /* Aqui deberia chequearse que el
      denominador no es 0 */
   return T;
}
```

Como ejemplo de funciones de procesamiento podríamos tener alguna que sume dos complejos, otra que efectúe restas, multiplicaciones, etc. En el caso de multiplicar, por ejemplo, podríamos escribir:

```
Complejo multiplicar(Complejo A, Complejo B)
{
   Complejo P, Q, R;
   Q=Polar(A);
   R=Polar(B);
   P.r=1; /* regrese el resultado en representacion polar */
   P.x=Q.x*R.x;
   P.y=Q.y+R.y;
   return P;
}
```

Pero hay que recordar que multiplicar un complejo por un número real también tiene sentido. Sin embargo, la función anterior no es capaz de hacer eso. Por supuesto, existe la posibilidad de escribir una función como:

```
Complejo multiplicar_real(Complejo A, double B)
{
   Complejo R;
   R.r = 0; /* esta y las siguientes asignaciones deberian hacerse
               a traves de una funcion de modificacion adecuada,
               digamos DefinirComplejo(), pero ya que aun no hemos
               escrito tal funcion, haremos un acceso directo a los
               campos de R */
   R.x  = B;
   R.y = 0;

   return multiplicar(A,R);
}
```

Sin embargo, también tenemos la opción de *sobrecargar* la función multiplicar, de manera que no creamos un nuevo nombre (multiplicar_real, por ejemplo), sino una nueva versión de una funcíon existente que sepa qué hacer con un argumento complejo y uno real:

```
Complejo multiplicar(Complejo A, double B)
{
   Complejo R;

   R.r = 0; /* esta y las siguientes asignaciones deberian hacerse
               a traves de una funcion de modificacion adecuada,
               digamos DefinirComplejo(), pero ya que aun no hemos
               escrito tal funcion, haremos un acceso directo a los
               campos de R */
   R.x  = B;
   R.y = 0;

   return multiplicar(A,R);
}
```

No hay ambigüedad porque el compilador sabe cuál función invocar en base a la lista de argumentos. Por ejemplo, si multiplicamos dos complejos la primera versión de la función se usa, si multiplicamos un complejo con un real, la segunda se aplica; ésta convierte el real en un complejo y luego invoca a la primer versión.

Para cubrir el caso de un real multiplicando a un complejo podríamos sobrecargar la función nuevamente:

```
Complejo multiplicar(double A, Complejo B)
{
   return multiplicar(B,A);
}
```

Nótese lo fácil que fue sobrecargar esta versión. Siempre que sea posible es buena práctica hacer uso de código ya escrito, en vez de recodificar. A esto se le llama reuso de código y ahorra tiempo de programación.

Descripción

Defina un tipo `Complejo` capaz de representar una variable compleja en representación polar o rectangular.

Incluya funciones de acceso que permitan manipular los contenidos para modificarlos o para consultarlos. Entre las funciones de consulta tendremos al menos:

```
Re() Im() Mag() Arg()
```

Entre las funciones de modificación estarán por lo menos:

```
Definir() FijarRe() FijarIm()
EsPolar() EsRect()
```

Y entre las funciones de procesamiento estarán:

```
Rect() Polar() Conj() Inv()
sumar() restar() multiplicar()
dividir() elevar()
```

Agregue las funciones que hagan falta y asegúrese de sobrecargar las funciones apropiadas para cubrir todos los casos de interés.

Impresión

1. Imprima su programa.
2. Imprima sólo los prototipos de las funciones que manejan números complejos.

Ejecución

1. Escriba su programa de tal forma que al ser ejecutado pida al usuario dos números complejos A, B y un número real s y un número entero n. Utilizando las funciones que manejan números complejos, haga que el programa saque a la pantalla el resultado de las siguientes operaciones:
   a) $A + B$
   b) $B - A/s$
   c) sAB
   d) B/A
   e) $1/(A - B)$
   f) B^n/s
   g) A^nB
2. Envíe el código fuente de su programa por correo electrónico.

Preguntas

1. Cuáles fueron las funciones de consulta asociadas a la estructura `Complejo`?
2. Cuáles las de modificación?
3. Cuáles las de procesamiento?
4. Qué funciones debieron sobrecargarse?

Experimento #6:

Sobrecarga de operadores y programación modular

Objetivos

- Aprender a agrupar unidades funcionales en archivos de encabezado.
- Sobrecargar operadores en C++ para facilitar la programación.

Descripción

En el experimento anterior se creó una pequeña librería de funciones útiles para manipular variables de tipo complejo. Una vez probadas y verificadas, estas funciones (y los tipos y constantes que hayamos definido) podrían utilizarse en otros programas. Podríamos, por ejemplo, guardar todas las estructuras, constantes, prototipos y definiciones en un sólo archivo para que cualquier programa que incluyera a ese archivo (usando una directiva `#include`) pudiera hacer uso de ellas. Esto se logra simplemente guardando nuestras funciones, etc., en un archivo terminado en ".h". Luego, una vez escritas y probadas, podemos crear nuestro archivo de encabezado (así se llaman los archivos terminados en ".h") simplemente eliminando la parte que ejercita nuestras funciones (todo lo que habíamos escrito en main() para probar nuestras funciones) y guardando el resultado en un archivo de nombre *Complejo.h* por ejemplo.

Es conveniente que *Complejo.h* tenga un comentario indicando qué hay contenido en él, y contenga además los prototipos y luego las definiciones de todas las funciones.

```
/* Complejo.h: este archivo contiene la definicion de la estructura
   Complejo y las funciones de acceso y modificacion pertinentes. */

#ifndef Complejo_H
#define Complejo_H

/* Aqui vendria la definicion de la estructura Complejo */

...
```

```
/* Aqui vendrian los prototipos de las funciones */

...

/* Aqui estarian las definiciones de las funciones de acceso */

/* Primero funciones de consulta */

...

/* Segundo funciones de modificacion */

...

/* Tercero funciones de procesamiento */

#endif
```

Nótese la inclusión de una directiva de compilación condicional (el `#if` cerrado por el `#endif`) que evita que el archivo pueda incluirse más de una vez en el programa fuente.

Parte de este experimento es encapsular, por así decirlo, nuestro módulo de complejos en un archivo de encabezado. Sin embargo, queremos también ampliar el módulo.

En estos momentos, si quisiéramos sumar un par de número complejos A y B y guardar el resultado en C tendríamos que invocar a la función sumar de la siguiente manera:

```
C = sumar(A,B);
```

No sería mejor poder escribir "C=A+B;" por ejemplo? Sin embargo esto no es posible usando el "+" de C++ porque éste no opera con números complejos... a menos que sobrecarguemos este operador!

Para sobrecargar un operador en C++ debemos redefinir la función que éste efectúa. El nombre formal del operador comienza con `operator`, así podríamos sobrecargar el "+" de la siguiente manera:

```
Complejo operator+(Complejo A, Complejo B)
{
   return sumar(A,B);
}
```

Claro, tendríamos que hacer definiciones similares para los casos de un real sumado a un complejo, o un complejo sumado a un real. Pero esto nos permitiría escribir cosas como:

```
D = A+B+C;
```

Donde A, B, C, y D son variables complejas. La razón es que la expresión se evalúa por pares de derecha a izquierda:

```
D= A + (B+C)
```

Así se efectúa la suma de B y C (ambos complejos) primero, y el resultado es complejo, después se hace A + (un complejo) que resulta en un complejo que finalmente se asigna a D.

Descripción

Sobrecargue todos los operadores aritméticos de C++ para que puedan operar sobre números complejos.

Produzca por separado un archivo de encabezado *complejo.h* y un ejercitador (el programa que verifica la operación de sus subrutinas) *complejo.c*. En el ejercitador trate además de incluir *complejo.h* varias veces para verificar que ésto no produce resultados adversos.

Impresión

1. Imprima su archivo de encabezado.
2. Imprima su programa fuente.
3. Imprima una ejecución típica del ejercitador.

Preguntas

1. Qué operadores no pueden sobrecargarse en C++?
2. Puede una función sobrecargada diferir de otra versión de sí misma sólo en el tipo del resultado que regresa?
3. Qué pasa si dos funciones con el mismo nombre poseen además la misma lista de argumentos?
4. Puede una función que no es `void` ser invocada sin que su resultado sea utilizado para nada? Si la respuesta es afirmativa, indique cuál sería el propósito de invocar una función si se va a deshechar su resultado. Si la respuesta es negativa indique que tipo de error reporta el compilador o el programa.
5. Explique por qué las directivas `#if` y `#endif` previenen la doble inclusión de *complejo.h*.

6. Cómo se incluye *complejo.h* en el programa? Si un programa incluye dos encabezados, *encabezado1.h* y *encabezado2.h*, y ambos definen un símbolo llamado SIM con distintos valores, describa cómo reacciona el compilador y el programa (se ejecuta, se reporta un error, etc.).

Experimento #7:

Punteros, vectores y asignación dinámica de memoria

Objetivos

- Comprender el uso de punteros.
- Aprender a declarar y usar vectores.
- Aprender a declarar vectores de tamao variable.

Descripción

Hasta ahora hemos declarado y usado variables de tipos predefinidos y de tipos definidos por el usuario. Los nombres de estas variables, físicamente, corresponden a localidades (direcciones) de memoria cuyo contenido es precisamente el contenido de la variable en cuestión. La ubicación exacta de una variable puede obtenerse a través del operador &. Este operador está sobrecargado y también sirve para efectuar una operacion AND, bit por bit, entre dos operandos del mismo tipo entero. Sin embargo, asociado al nombre de una variable, regresa la dirección en memoria de esta variable. Por ejemplo:

```
int A,B;
```

declara dos variables enteras. La operación

```
&A;
```

regresa la dirección en memoria donde los cuatro bytes de la variable entera *A* comienzan. En general esta dirección variará en diferentes ejecuciones del programa. Podemos asignar esta dirección a un tipo especial de variable, llamada un puntero. Se declara una variable de tipo puntero a través del operador *, de esta manera tenemos que el enunciado

```
int *pentero;
```

declara una variable llamada pentero capaz de contener la dirección de memoria donde una variable entera comienza.

Ahora es posible no sólo guardar la dirección de A sino también manipular su contenido a través del puntero. De esta forma el siguiente fragmento de código:

```
pentero = &A;
*pentero = 5;
pentero = &B;
*pentero = 7;
```

es totalmente equivalente a:

```
A = 5;
B = 7;
```

Por qué, entonces, tomarse la molestia de usar punteros? Bien, existen ocasiones en las que queremos crear un número de variables y manipularlas sin poner atención específica a sus nombres (de hecho, muchas veces ni siquiera tienen nombres!) y para ello podríamos usar un puntero que, al cambiar su dirección, automáticamente asume la "personalidad" de la nueva variable.

Un caso muy relacionado con punteros es la declaración de vectores. El enunciado:

```
int vectorentero[10];
```

declara a un vector llamado *vectorentero* con diez componentes que van desde `vectorentero[0]` a `vectorentero[9]`. Cada uno de estos elementos es en sí una variable entera y por lo tanto puede tratarse como tal. El nombre vectorentero en sí es un puntero, aunque de tipo especial ya que su contenido (la dirección donde comienzan las 10 variables enteras) no puede cambiarse. Sin embargo, sería perfectamente válido hacer lo siguiente:

```
pentero = vectorentero;
```

Y en estos momentos tanto *pentero, como pentero[0], vectorentero[0] apuntan al mismo punto: el primer elemento del vector.

Para acceder al siguiente miembro, vectorentero[1], podríamos hacer uso de *(pentero+1), o pentero[1]. La primera forma se llama indexado por puntero y desplazamiento, la segunda se denomina indexado por puntero y subíndice.

Los vectores pueden inicializarse:

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

declara a vectorentero con cinco elementos: vectorentero[0] es igual a 5, ..., vectorentero[4] es 1.

Una pequea función que regresa la suma de los elementos de un vector de variables enteras podría escribirse como:

```
int sumacomponentes(int *vec, int n)
{
   /* El "*vec" podria sustituirse por "vec[]", la variable n indica el
      tamao del vector */
   int contador,resultado=0;

   for (contador = 0; contador<n; contador++)
   {
      resultado = resultado + vec[contador];
      /*  otras posibilidades para esta linea son:
            resultado += vec[contador];
            o
            resultado += *(vec+contador);
            o
            resultado += *vec;
            vec++;
            estas ultimas dos lineas podrian condensarse en una:
            resultado +=*(vec++); */
   }

   return resultado;
}
```

Nótesen las distintas maneras de realizar la suma, incluyendo la posibilidad de modificar el puntero vec mismo. También nótese que uno de los parámetros de la función es el tamao del vector. Esto es necesario porque los vectores no guardan, en ninguna parte, su tamao. Quizá esto implique que sea conveniente definir un tipo, digamos Vector que sí lo haga, por ejemplo:

```
struct Vector
{
   int t; /* tamao del vector */
   int elemento[t];    /* el vector mismo   */
};
```

Desafortunadamente ésto no es posible: no se pueden definir vectores de tamao variable dentro de una estructura. Lo que sí es posible sería definir un tamao máximo MAX, y que el campo t indique cual es el número real de elementos.

```
#define MAX 5000

struct Vector
{
   int t; /* tamao del vector */
   int elemento[MAX];    /* el vector mismo   */
};
```

Pero hacer ésto equivale a desperdiciar elementos dentro del vector si $t <$ MAX y a no poder representar vectores con mas de MAX elementos.

Afortunadamente este problema puede resolverse mediante el uso de punteros. El procedimiento es el de crear un puntero y asignarle memoria después según sean las necesidades.

Así una mejor manera de definir nuestro tipo Vector sería:

```
struct Vector
{
   int n; /* tamao del vector */
   int *elemento;    /* el vector mismo   */
};

void definir(int tamao, Vector &V)
{
   V.n = tamao;
   V.elemento = malloc(sizeof(int)*tamao); /* Reservar el espacio necesario*/
   for (int i=0; i<tamao; i++)
   {
      /* inicialice los elementos a 0 */
      V.elemento[i] = 0;
   }
}

void borrar(Vector &V)
{
   V.n = 0;
   free(V.elemento);     /* Devolver el espacio usado */
   V.elemento = NULL;
}
```

A estas funciones se les agregarían las acostumbradas funciones de acceso y procesamiento.

Las instrucciones `malloc` y `free` solicitan y liberan memoria respectivamente. Alternativamente pueden usarse `new` y `delete`. De hecho el uso de estas últimas se recomienda en programas de C++ por razones que escapan el alcance de este curso. Nosotros nos contentaremos con usar el primer juego.

Procedimiento

Escriba una librería llamada *vector.h* que permita la definición de variables de tipo Vector de tamao variable. Las siguientes operaciones deben ser parte de la librería:

ppunto	el producto punto
mag	la magnitud de un vector
+ y -	suma y resta de vectores

No iremos mas allá en el caso de vectores porque realmente estamos interesados en desarrollar una estructura capaz de representar matrices (de las cuales los vectores son sólo un caso especial) que será más completa y útil.

Escriba también un programa que ejemplifique y verifique la operación de su librería.

Impresión

1. Imprima vector.h.
2. Imprima el programa ejercitador.
3. Imprima los resultados de ejecutar el ejercitador.

Preguntas

1. Investigue el formato de `malloc` y `free`.
2. Investigue `new` y `delete`.
3. Defina una estructura capaz de representar funciones de tiempo, incluya los prototipos de las funciones asociados a esta estructura.
4. Defina una estructura capaz de representar matrices, incluya los prototipos de las funciones asociados con esta estructura.

Experimento #8:

Introducción a clases en C++

Objetivos

- Aprender a declarar clases en C++.
- Entender las ventajas de una implementación utilizando clases.
- Comprender el uso y la necesidad de constructores. Entender los conceptos de encapsulación, ocultación de información, y abstracción.

Descripción

Hasta ahora hemos recurrido al uso de estructuras para definir nuevos tipos cuando los tipos preexistentes no han sido suficiente para englobar toda la información que queremos asociar a una variable. También hemos visto que podemos, a través del uso de arreglos, definir una colección de variables de un mismo tipo para automatizar su manipulación y/o procesamiento.

Cuando hemos creado una estructura también creamos funciones de acceso y procesamiento para manipular sus componentes. Sin embargo, el acceso directo a los campos de una estructura aún era permitido. Las funciones que creamos (globales), frecuentemente hacían accesos directos y otras funciones (o el programa principal) podían hacer lo mismo. Esto crea la posibilidad de usar "atajos" para modificar la información contenida en una variable, con cierta ganancia quizá en velocidad, pero a costa de integridad en la información ya que al evitar el uso de la función de acceso adecuada se pierde la verificación que ésta hacía sobre la consistencia y validez de los datos.

Las funciones de acceso y procesamiento que creamos en asociación a estructuras estan débilmente ligadas a éstas: son funciones globales. Podrían aparecer en módulos diferentes del programa (aunque ésto es una práctica pobre de programación) y no sabríamos ni cuáles son, ni cuántas hay. Los datos miembro de una estructura pueden ser modificados a voluntad del programador sin intervención de las funciones de acceso, y los detalles de la implementación son visibles a todos los programas (denominados clientes) que hacen uso de la estructura. Todas estas son, obviamente, desventajas.

Afortunadamente en C++ estamos en capacidad de corregir la situación a través del uso de clases. Una clase es también un tipo definido por el usuario y pueden declararse variables de este tipo. Estas variables reciben el nombre de textitobjetos. La clase se declara por medio de la palabra clave `class` y al igual que las estructuras contiene campos llamados datos miembro, pero también incluye a sus funciones de acceso y procesamiento (llamadas funciones miembro). Una clase capaz de representar vectores podría ser escrita como:

```
// Clase para manejar y operar vectores en tres dimensiones

class Vector
{
private:
   double x;  // componente x
   double y;  // componente y
   double z;  // componente z

public:
   // Constructores
   Vector();  // para cuando un objeto se declara sin argumentos
   Vector(double a, double b, double c); //cuando se declara con argumentos

   // Funciones de acceso
      // Las tres funciones siguientes devuelven una componente del vector
   double comp_x();
   double comp_y();
   double comp_z();
   double mag();  // Retorna la magnitud del vector
   double mag(int);  // Retorna la magnitud elevada a una exponente entero

   // Funciones de modificacion
   void fijar_x(double);
   void fijar_y(double);
   void fijar_z(double);

   // Funciones de procesamiento
   Vector operator+(Vector);
   Vector operator-(Vector);
   Vector operator*(double);
   Vector operator/(double);
   double ppunto(Vector);
   Vector pcruz(Vector);
};
```

La definción de la clase vector consta de dos partes: `private:` y `public:`. Cualquier función colocada después de `public:` es accesible a cualquier pro-

grama que tenga acceso a un objeto de la clase vector y son conocidas como funciones miembros de la clase. Cualquier dato declarado después de `private:` puede ser accesado únicamente por las funciones miembros de la clase. Esto implica que para poder manipular las componentes de nuestros vectores se debe implementar funciones de acceso y modificación. Esto hace que la parte privada de la clase permanezca oculta para el programador, no teniendo que preocuparse por los detalles de cómo esta estructurado el objeto que se está manejando.

La parte pública de la definición de la clase contiene los prototipos de las funciones que pueden ser utilizadas por cualquier otro programa, así como los constructores de la misma. Estas funciones son los servicios que provee la clase o la interface de la clase. El código propio de cada una de estas funciones se escribe a continuación de la definición de la clase. Teniendo en mente anteponer el nombre de la clase seguido por el operador :: al nombre de la función.

El constructor de una clase es una función especial cuyo nombre es el mismo que el de la clase. El constructor para la clase Vector sería:

```
Vector::Vector()
{
   x = 0;
   y = 0;
   z = 0;
}
```

Es importante notar que esta función no regresa ningún dato y que no va precedida por la palabra `void`. El constructor se encarga de inicializar las variables de la parte privada de la clase al momento en que una variable es declara como un objeto de tipo vector. Esto se hace de la misma forma en que se declara una variable de tipo `int, double, char`, etc. Es decir que para declarar una variable de tipo vector lo que debemos hacer es escribir:

```
Vector A;
```

Cuando se alcanza esta línea en la ejecución del programa, se ejecuta el código del constructor de la clase. En este caso las componentes del vector son inicializadas a cero.

Pudiera darse el caso en que quisieramos inicializar un vector con componentes específicas. Esto también es tarea del constructor y lo podemos realizar sobrecargando el constructor de la clase:

```
Vector::Vector(double a, double b, double c)
{
   x = a;
   y = b;
   z = c;
}
```

En este caso, la declaración del objeto no es simplemente `Vector A`, sino que sería:

```
Vector A(1, 5, -7);
```

el constrctor inicializa la componente x como 1, la componente y como 5 y la componente z como -7. Nuestra clase tendría entonces dos constructores, uno para cuando un vector es creado sin especificar los valores de sus componentes y otro cuando las componentes son conocidas. Los prototipos de cada constructor deben estar en la parte pública de la definición de la clase.

La forma de accesar a la funciones públicas de la clase es a través del punto. Por ejemplo, el producto punto entre dos vectores se calcula con la función `ppunto`. Si nuestros vectores son A y B para calcular el producto punto escribimos:

```
Vector A(4, 5, -1), B(-7, 0, 2); //creamos dos vectores
double producto;

producto = A.ppunto(B);  //realizamos el producto punto
```

El código para calcular el producto punto sería:

```
double Vector::ppunto(Vector q)
{
   double w;
   w = x * q.x + y * q.y + z * q.z;
   return w;
}
```

Es importante notar que la función recibe únicamente un parámetro que es el vector B. El vector A pasa por referencia y sus componentes dentro de la función miembro ppunto se conocen simplemente como x, y, z. En cambio, las componentes del vector B (dentro de la función) se conocen como $q.x, q.y, q.z$. Todas las funciones miembros de la clase tienen esta característica. El objeto que antecede al punto, pasa por referencia y cualquier otro objeto pasa como un parámetro de la función.

En el caso de la sobrecarga de operadores sucede algo similar, el objeto que antecede al operador pasa por referencia y el segundo es un parámetro de dicha función. Así, por ejemplo, la suma de dos vectores puede escribirse sobrecargando el operador $+$, de la siguiente forma

```
Vector Vector::operator+(Vector q)
{
   Vector w;  //declaramos un vector auxiliar

   // efectuamos la suma componente por componente
   w.x = x + q.x;
   w.y = y + q.y;
   w.z = z + q.z;
```

```
   return w;  // retornamos el vector conteniendo la suma calculada
}
```

La función anterior nos permite hacer una suma de vectores simplemente escribiendo `A + B`. Donbre A y B son objetos definidos como vectores. En el código de la función las componentes del primer vector son conocidas simplemente como x, y, z en cambio las del segundo vector son conocidas teniendo que especificar el objeto al cual nos referimos.

Procedimiento

Escriba una librería llamada *vector.h* que permita la defínción de vectores en tres dimensiones. Las operaciones que deben estar incluidas en la librería son todas aquellas que se pueden efectuar con un vector: suma, resta, multiplicación por escalar, división por escalar, magnitud, producto punto y producto cruz. Además se debe implementar las funciones de acceso y modificación para cada una de las componentes de un vector. Adicionalmente la librería debe contener todas las sobrecargas necesarias de las funciones u operadores para el producto de vector por escalar y división por escalar, así como la sobrecarga del operador de inserción para desplegar adecuadamente un vector en la pantalla.

Escriba también un programa ejecitador que ejecute y verifique las funciones de la librería.

Impresión

1. Imprima la *vector.h*.
2. Imprima el programa ejecitador.
3. Imprima los resultados de ejecutar el ejercitador.

Preguntas

1. ¿Cuáles son las funciones públicas de la clase?
2. ¿Pertenecen todas las funciones de su librería a la parte pública de la dinición de la clase?
3. ¿Qué funciones no pertenecen a la parte pública de la clase?
4. ¿Por qué la función que sobrecarga el operador de inserción no pertenece a la clase?

Experimento #9:

Operaciones con matrices

Objetivos

- Construir una clase capaz de realizar operaciones con matrices.
- Comprender la necesidad de constructores y destructores.
- Emplear los conceptos de asignación dinámica de memoria.

Descripción

Ahora que hemos aprendido cómo se define e implementa una clase, deseamos crear un objeto que pueda almacenar la información contenida en una matriz. La forma en la que esto se puede lograr no es única. Lo primero que vendría a nuestra mente sería utilizar un arreglo de dos dimensiones de variables de tipo `double`, de tal forma que un elemento de la matriz sea por ejemplo: `A[3][6]`. Esto tiene algunos inconvenientes. Debido a que no todas las matrices tienen el mismo tamaño, debemos utilizar la asginación dinámica de memoria. Para un arreglo unidimensional (de un solo índice) el procedimiento es directo. Sin embargo, para un arreglo de dos dimensiones (dos índices) que tiene m filas y n columnas; primero debemos asignar espacio en memoria para m punteros de tipo `double*` y luego asignar el espacio necesario para que cada uno de estos punteros contenga n variables de tipo `double`. De esta forma, la matriz es creada como una colección de m vectores cada uno con n componentes. El inconveniente de esta forma de representar una matriz, consiste en que las componentes de la misma se almacenan en bloques separados de memoria lo que complicaría el acceso a las componentes a través de punteros. Para evitar esto y tener las componentes de la matriz en un solo bloque de memoria, utilizaremos un arreglo unidimensional. Si la matriz tiene m filas y n columnas, el tamaño del arreglo será de $m \times n$. Las primeras n posiciones del arreglo corresponden a los elementos de la primera fija de la matriz, es decir, desde 0 hasta $n-1$. Los elementos de la segunda fila van desde la posición n hasta la $2(n-1)$. De esta forma el elemento que se encuentra en la fija i, columna j se encuentra en la posición $(i-1)*n+(j-1)$ del arreglo. Adicionalmente el objeto que represente una matriz debe guardar el tamaño de la misma, es decir, el número de filas y de columnas; pues el arreglo no guarda su tamaño en ninguna parte.

La definición de la clase matriz sería:

```
class matriz
{
private:
   int n_fil;  // numero de filas;
   int n_col;  // numero de columnas;
   double *elemento; //puntero al que se debe asignar el espacio
                     //necesario para representar una matriz

public:
   matriz(int, int);  //constructor de la clase
   ~matriz();         //destructor de la clase
   void dimensionar(int, int);
   void borrar();

};


matriz::matriz(int fil, int col)
{
   //quitamos cualquier dato que pueda estar contenido en las variables
   //privadas de la clase
   n_fil = 0;
   n_col = 0;
   elemento = NULL;

   //asignamos el numero filas y columnas y tambien el espacio necesario
   //para "elemento"
   dimensionar(fil, col);
}


matriz::~matriz()
{
   borrar();
}


// esta funcion nos permitira inicializar una matriz que esta siendo creada y
// tambien una matriz que ya ha sido creada con anterioridad.
void matriz::dimensionar(int fil, int col)
{
   //si la matriz ya existia borramos lo que contenga y si no existia
   //la funcion borrar() no tiene ningun efecto.
   borrar();
```

```
   //verificamos que el numero de filas y columnas sean enteros
   //positivos
   if ( (fil >= 0) && (col >= 0) )
   {
      //asignamos espacio al puntero
      elemento = new double[fil*col];

      //definimos el numero de filas y columnas
      n_fil = fil;
      n_col = col;

      //inicializamos cada elmento a cero
      for(int k = 0; k <= fil*col-1; k++)
         elemento[k] = 0;
   }

   //si fil y col no son enteros positivos desplegamos el error en la
   //pantalla y terminamos la ejecucion del programa
   else
   {
      cout<<endl;
          <<"Error! el numero de filas o columnas no es positivo"
          <<endl;
      exit( 1 );
   }
}


//borra el contenido de una matriz
void matriz::borrar()
{
   if (elemento != NULL)
      delete [] elemento;

   n_fil = 0;
   n_col = 0;
   elemento = NULL;
}
```

En la definición de la clase ahora aparece la función `~matriz` que es el *destructor de la clase*. El destructor es un función especial cuyo nombre es igual al nombre de la clase antecedido por el símbolo ~. Al igual que el constructor, no devuelve nada y no lleva la palabra `void`. El destructor no recibe parámetros. Su propósito es, como su nombre lo indica, eliminar o borrar los objetos

(matrices, en este caso) que ya no se utilizarán más en una función. Este es el caso de los objetos que se declaran localmente en las funciones. Una vez que la ejecución del programa alcanza el final de una función, todas las variables localmente definidas se eliminan. El destructor es invocado por el programa de forma automática cada vez que se necesite eliminar un objeto de la clase. El destructor es importante en las clases cuyos objetos hacen uso de asignación dinámica de memoria, ya que el espacio reservado mediante `new` o `malloc` debe ser liberado con `delete` o `free`, respectivamente. El tener un destructor apropiado garantiza que los recursos de memoria utilizados por el programa son devueltos al sistema al término de la ejecución. De lo contrario cada vez que el programa corre estaría ocupando nuevo espacio, lo cual; eventualmente terminaría con la memoria del sistema. En el experimento anterior no necesitamos un destructor ya que no asignamos memoria de forma dinámica.

Es importante notar que el constructor de la clase hace uso de la función `dimensionar` para reservar el espacio del puntero `elemento` e inicializar las variables `n_fil` y `n_col`. La creación de esta función permitirá reajustar el espacio concedido a una matriz ya existente. Por ejemplo, en el caso de la multiplicación de matrices, el tamaño de la matriz producto depende del tamaño de las matrices factores.

Procedimiento

Escriba una librería llamada *matriz.h* que permita la definición de matrices reales de m filas por n columnas. El mínimo conjunto de funciones que se debe implementar son:

1. Las funciones de acceso que permitan obtener:
   - el número de filas de la matriz
   - el número de columnas de la matriz
   - el ij-ésimo elemento de la matriz
2. Una función que permita modificar el ij-ésimo elemento de la matriz.
3. Una función que sobrecargue el operador de inserción para desplegar una matriz en pantalla.
4. Las funciones de procesamiento que permitan manejar el álgebra de matrices, es decir, que las operaciones que deben estar definidas son:
   - suma de matrices
   - resta de matrices
   - producto de matrices
   - producto de matriz por escalar
   - producto de escalar por matriz

- cociente de matriz entre escalar
- elevar una matriz a una potencia entera

5. Las funciones de procesamiento que realizarán operaciones propias de las matrices:
   - determinante
   - matriz transpuesta
   - matriz inversa
   - matriz menor

Escriba también un programa ejercitador que ejecute y verifique la funciones de su librería.

Impresión

1. Imprima *matriz.h*.
2. Imprima el programa ejercitador.
3. Imprima la salida del programa ejercitador.

Preguntas

1. ¿Cuáles son las funciones que no son miembros de la clase?
2. ¿Es necesario implementar una función que sobregarque el operador = (igual) ? Explique.
3. ¿Es necesario crear otro constructor para la clase matriz, aparte del que se discutió anteriormente?, Explique.
4. ¿Cuáles de las funciones pueden ser implementadas en forma recursiva?