Práctica 4:
Tipos de datos abstractos

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

Tipo de dato abstracto El alumno aprenderá a crear estructuras en lenguaje C para modelar tipos de datos abstractos y a utilizarlos junto con las estructuras de datos lineales.

2. Material

Se asume que el alumno cuenta con un una computadora con arquitectura Intel x86 o compatible, sistema operativo Linux/Unix, y compilador `gcc` instalado.

3. Instrucciones

1. Lea detenidamente los antecedentes teóricos sobre la construcción de estructuras para manejo de datos abstactos y nuevos tipos de datos en la Sección 4.
2. Realice cada una de las actividades detalladas en la Sección 5 apoyándose en el código de ejemplo del Apéndice A y responda las preguntas de las mismas.

4. Antecedentes[1]

Una estructura es una colección de una o más variables que pueden ser de diferentes tipos, todas ellas agrupadas bajo un solo nombre para un manejo conveniente. Las estructuras ayudan a organizar datos complejos, particularmente en programas grandes, porque permiten que un grupo de variables relacionadas se trate como una unidad en lugar de como entidades separadas.

Un ejemplo tradicional de una estructura es el registro de nómina: un empleado se describe mediante un conjunto de atributos como nombre, dirección, número de seguro social, salario, etc. Algunos de estos, a su vez, podrían ser estructuras: un nombre tiene varios componentes, como hace una dirección y hasta un salario. Otro ejemplo, más típico de C, proviene de los gráficos: un punto es un par de coordenadas, un rectángulo es un par de puntos, etc.

El principal cambio realizado por el estándar ANSI es definir la asignación de estructura: las estructuras pueden copiarse y asignarse, pasarse a funciones y devolverse mediante funciones. Esto es soportado por la mayoría de los compiladores desde hace años, pero las propiedades ahora están definidas con precisión. Las estructuras y matrices automáticas ahora también se pueden inicializar.

4.1. Estructuras básicas

A continuación se definirán algunas estructuras adecuadas para gráficos. El objeto básico es un *punto*, que asumiremos que tiene una coordenada x y una coordenada y, ambas enteras. Los dos componentes se pueden colocar en una estructura declarada así:

[1] *Esta sección, en su totalidad, es una traducción del capítulo 6 y sección 6.1 del libro «The C Programming Language» de B. Kernighan y D. Ritchie.*

```
struct point {
    int x;
    int y;
};
```

La palabra clave `struct` introduce una declaración de estructura, que es una lista de declaraciones entre llaves. Un nombre opcional llamado *structure tag*, o etiqueta de la estructura, puede seguir a la palabra `struct` (la palabra `point` en el ejemplo). La etiqueta nombra este tipo de estructura y se puede utilizar posteriormente como abreviatura de la parte de la declaración entre llaves.

Las variables nombradas en una estructura se denominan *miembros*. Un miembro de estructura o etiqueta y una variable ordinaria (es decir, no miembro) pueden tener el mismo nombre sin generar conflictos ya que siempre se pueden distinguir por contexto. Además, los mismos nombres usados por los miembros de una estructura pueden ser reutilizados en otras estructuras difrentes, aunque, por cuestión de estilo, normalmente se utilizarían los mismos nombres sólo para objetos estrechamente relacionados.

Una declaración `struct` define un tipo. La llave derecha que termina la lista de miembros puede ir seguida de una lista de variables, al igual que para cualquier tipo básico. Eso es,

```
struct { \dots{} } x, y, z;}
```

es análogo a

```
int x, y, z;
```

en el sentido de que ambas declaran a x, y y z como variables del tipo mencionado y hace que se les reserve un espacio.

Una declaración de estructura que no va seguida de una lista de variables no reserva memoria, sino que simplemente describe una plantilla: la forma de la estructura. Sin embargo, si la declaración está etiquetada, la etiqueta se puede usar más adelante en las definiciones de instancias de la estructura. Por ejemplo, dada la declaración del punto anterior,

```
struct point pt;
```

define una variable pt que es una estructura de tipo struct point. Una estructura se puede inicializar siguiendo su definición con una lista de inicializadores, cada uno una expresión constante, para los miembros:

```
struct maxpt = {320, 200};
```

Una estructura automática también se puede inicializar por asignación o llamando a una función que devuelve una estructura del tipo correcto. En una expresión se hace referencia a un miembro de una estructura particular mediante una construcción de la forma `nombre-estructura.miembro`

El operador de miembro de estructura «`.`» Conecta el nombre de la estructura y el nombre del miembro. Para imprimir las coordenadas del punto pt, por ejemplo,

```
printf("%d, %d", pt.x, pt.y);
```

o para calcular la distancia desde el origen (0,0) a `pt`,

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Las estructuras se pueden anidar. Una representación de un rectángulo es un par de puntos que denotan las esquinas diagonalmente opuestas:

```
struct rect {
  struct point pt1;
  struct point pt2;
};
```

La estructura `rect` contiene dos estructuras tipo `point`. Si se declara `screen` como

```
struct rect screen;
```

enconces `screen.pt1.x` hará referencia a la cordenada x de $pt1$, miembro de `screen`.

5. Desarrollo de la práctica

Lea cuidadosamente cada una de las actividades propuestas antes de realizar el programa o las modificaciones indicadas.

5.1. Actividad 1: Estructuras vs tipos

Los programas de los Apéndices A.2 y A.3 suman números complejos y son, en principio, equivalentes.
Compile y ejecute los programas de los Apéndices A.2 y A.3 con

```
./add_scomplex 1 -1+2i
./add_tcomplex 1 -1+2i
```

y compare los resultados obtenidos para los mismos conjuntos de datos. ¿Son iguales? Posteriormente analice el código de cada uno de los programas y estudie las diferencias. ¿En qué se diferencian? ¿Qué programa es más fácil de leer? Explique [2 puntos].

5.2. Actividad 2: Producto de números complejos

Con base en los programas de los Apéndices A.2 y A.3 desarrolle un programa que tome como parámetros dos números complejos y devuelva como resultado el producto de los mismos. Compile su programa y pruébelo. ¿Qué programa se tomó como base? ¿Qué modificaciones se realizaron? Explique [1 punto].

5.3. Actividad 3: Suma de vectores en 2D

El programa del Apéndice A.4 calcula la suma de dos vectores de dimensión 2.
Compile y ejecute el programa del Apéndice A.4 con

```
./add_v2 1,0 -1,-1
```

Posteriormente analice el código y compárelo con el de los los programas de los Apéndices A.2 y A.3. ¿Qué similitudes encuentra? ¿En qué se diferencian? Explique [2 puntos].

5.4. Actividad 4: Suma de vectores en 3D

Modifique una copia del programa del Apéndice A.4 para que calcule la suma de dos vectores de dimensión 3 recibidos por línea de comandos. Nombre a su nuevo programa `src/add_v3.c`. Compile, ejecute y verifique correctez.

¿Qué modificaciones se realizaron? Explique [2 puntos].

5.5. Actividad 5: Suma normalizada de vectores en 3D

Modifique una copia del programa anterior para que calcule la suma **NORMALIZADA** de dos vectores de dimensión 3 recibidos por línea de comandos. La normalización deberá realizarse mediante una función con prototipo:

```
void vector3_normalize(vector3 *v);
```

Nombre a su nuevo programa `src/addn_v3.c`. Compile, ejecute y verifique correctez.

¿Qué modificaciones se realizaron? ¿Cómo se implementó la función `vector3_normalize`? Explique [3 puntos].

A. Código de ejemplo

A.1. Archivo `src/Makefile`

src/Makefile

```
CC       = gcc
CFLAGS   = -Wall -O3 -std=c99 -pedantic
SILENT   = @
PROGRAMS = add_scomplex add_tcomplex add_v2
.PHONY: all clean $(PROGRAMS)

all: $(PROGRAMS)

$(PROGRAMS): %: %.c
  $(SILENT) $(CC) $(CFLAGS) -o $@ $<

clean:
  rm -f *.o
```

A.2. Archivo `src/add_scomplex.c`

src/add_scomplex.c

```c
#include <stdlib.h>
#include <stdio.h>

struct tcomplex{
  double re; /** The real part of the complex number */
  double im; /** The imaginary part of the complex number */
};

int main(int argc, char** argv);
struct tcomplex complex_add(struct tcomplex a, struct tcomplex b);
int complex_parse(char* s, struct tcomplex *out);
char* complex2str(struct tcomplex c);

int main(int argc, char** argv){
  struct tcomplex  a, b, c;

  if ( (argc < 3) || /* Read numbers. Quit on fail. */
    !complex_parse(argv[1], &a) ||
    !complex_parse(argv[2], &b) ) return -1;

  // Sum
  c = complex_add(a, b);
  // Print result
  printf("%s + %s = %s\n", complex2str(a), complex2str(b), complex2str(c));

  return 0;
}

struct tcomplex complex_add(struct tcomplex a, struct tcomplex b){
  struct tcomplex sum;
  sum.re = a.re + b.re;
  sum.im = a.im + b.im;
  return sum;
}

int complex_parse(char* s, struct tcomplex *out){
  char* bcc = s;
  char* cc;
  // init out
  out->re = 0; out->im = 0;
  if (s == NULL) return 0;
  // Convert real part
  out->re = strtod(bcc, &cc);
  if(*cc == '\0') return 1;
  // Convert imaginary part
  if ((*cc != '+') && (*cc != '-')) return 0;
  bcc = cc;
  out->im = strtod(bcc, &cc);
  if(*cc == 'i') return 1;
  return 0;
}

char* complex2str(struct tcomplex c){
  char *s = calloc(50, sizeof(char));
  sprintf(s, "%0.2f%+0.2fi", c.re, c.im);
  return s;
}
```

A.3. Archivo `src/add_tcomplex.c`

src/add_tcomplex.c

```c
#include <stdlib.h>
#include <stdio.h>

struct {
  double re; /** The real part of the complex number */
  double im; /** The imaginary part of the complex number */
} typedef complex;

int main(int argc, char** argv);
complex complex_add(complex a, complex b);
int complex_parse(char* s, complex *out);
char* complex2str(complex c);

int main(int argc, char** argv){
  complex  a, b, c;

  if ( (argc < 3) || /* Read numbers. Quit on fail. */
    !complex_parse(argv[1], &a) ||
    !complex_parse(argv[2], &b) ) return -1;

  // Sum
  c = complex_add(a, b);
  // Print result
  printf("%s + %s = %s\n", complex2str(a), complex2str(b), complex2str(c));

  return 0;
}

complex complex_add(complex a, complex b){
  complex sum;
  sum.re = a.re + b.re;
  sum.im = a.im + b.im;
  return sum;
}

int complex_parse(char* s, complex *out){
  char* bcc = s;
  char* cc;
  // init out
  out->re = 0; out->im = 0;
  if (s == NULL) return 0;
  // Convert real part
  out->re = strtod(bcc, &cc);
  if(*cc == '\0') return 1;
  // Convert imaginary part
  if ((*cc != '+') && (*cc != '-')) return 0;
  bcc = cc;
  out->im = strtod(bcc, &cc);
  if(*cc == 'i') return 1;
  return 0;
}

char* complex2str(complex c){
  char *s = calloc(50, sizeof(char));
  sprintf(s, "%0.2f%+0.2fi", c.re, c.im);
  return s;
}
```

A.4. Archivo `src/add_v2.c`

src/add_v2.c

```c

#include <stdlib.h>
#include <stdio.h>

struct {
  double v1; /** The first component of the vector */
  double v2; /** The second component of the vector */
} typedef vector2;

int main(int argc, char** argv);
vector2 vector2_add(vector2 a, vector2 b);
int vector2_parse(char* s, vector2 *out);
char* vector2_tostr(vector2 c);

int main(int argc, char** argv){
  vector2  a, b, c;

  if ( (argc < 3) || /* Read numbers. Quit on fail. */
    !vector2_parse(argv[1], &a) ||
    !vector2_parse(argv[2], &b) ) return -1;

  // Sum
  c = vector2_add(a, b);
  // Print result
  printf("%s + %s = %s\n", vector2_tostr(a), vector2_tostr(b), vector2_tostr(c));

  return 0;
}

vector2 vector2_add(vector2 a, vector2 b){
  vector2 sum;
  sum.v1 = a.v1 + b.v1;
  sum.v2 = a.v2 + b.v2;
  return sum;
}

int vector2_parse(char* s, vector2 *out){
  char* bcc = s;
  char* cc;
  // init out
  out->v1 = 0; out->v2 = 0;
  if (s == NULL) return 0;
  // Convert component 1
  out->v1 = strtod(bcc, &cc);
  printf("%p,%p => '%c', '%c'\n", cc, bcc, *cc, *bcc);
  if((cc == bcc) || (*cc != ',')) return 0;
  // Skip comma and convert component 2
  bcc = ++cc;
  out->v2 = strtod(bcc, &cc);
  printf("%p,%p => '%c', '%c'\n", cc, bcc, *cc, *bcc);
  if(cc == bcc) return 0;
  return 1;
}

char* vector2_tostr(vector2 c){
  char *s = calloc(50, sizeof(char));
  sprintf(s, "(%0.4f,%0.4f)", c.v1, c.v2);
  return s;
}
```

B. Reporte Escrito

El reporte de la práctica deberá ser entregada en un archivo en formato `PDF` siguiendo las siguientes especificaciones:

- La primera página del documento deberá ser la carátula oficial para prácticas de laboratorio disponible en `lcp02.fi-b.unam.mx/`
- El nombre del documento `PDF` deberá ser `nn-XXXX-L04.pdf`, donde:
  - `nn` es el número de lista del alumno a dos dígitos forzosos (ej. 01, 02, etc.).
  - `XXXX` corresponden a los cuatro primeros caracteres de su RFC o CURP.
- El reporte consiste en un documento de redacción propia donde el alumno explica de forma concisa y a detalle las actividades realizadas en la práctica, y reportando los resultados obtenidos.
- La longitud del reporte no deberá exceder las 3 páginas, sin contar la carátula.
- El reporte deberá seguir todos los lineamientos para documentos escritos establecidos al inicio del curso.
- Todas las referencias deberán estar debidamente citadas.

IMPORTANTE: No se aceptan archivos en otros formatos ni con nombres distintos a los especificados.