Tema 5. Estructuras

Metodología de la programación I

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Curso 2005-06

Índice

Motivación

A veces resulta útil tener una colección de valores de diferentes tipos y tratar la colección como una sola cosa.

Ejemplos

- Información sobre un estudiante (NIF, nombre, curso, grupo).
- Definición de un punto en el plano (con coordenadas x, y).
- Definición de una fecha (día, mes, año).
- Definición de una hora (hora, minutos, segundos, AM o PM).

Introducción

Definición

Las estructuras o registros son tipos de dato compuestos que se definen a partir de elementos de otros tipos.

sintaxis

```
struct <nombre_estructura> {
  <tipo1> <miembro 1>;
  <tipo2> <miembro 2>;
  ...     ...
  <tipon> <miembro n>;
};
```

La estructura tiene un nombre `<nombre_estructura>` que es el nombre del tipo de dato. Cada miembro tiene un nombre asociado `<miembro X>` que nos permitirá referenciarlo.

Ejemplos

- Información sobre un alumno (NIF, nombre, curso, grupo).

```
struct Alumno{
  char NIF[10];
  char nombre[200];
  int curso;
  char grupo;
};
```

- Definición de un punto en el plano (con coordenadas x, y).

```
struct Punto{
  double x;
  double y;
};
```

Ejemplos

- Definición de una fecha (día, mes, año).

```
struct Fecha{
  int dia;  // en el rango [1,31]
  int mes;  // en el rango [1,12]
  int anio;
};
```

- Definición de una hora (hora, AM o PM, minutos, segundos).

```
struct Tiempo{
  int hora;     // en el rango [0,11]
  bool am;      // true para AM y false para PM
  int minuto;   // en el rango [0,59]
  int segundo; // en el rango [0,59]
};
```

En la definición de `Tiempo` como la estructura

```
struct Tiempo{
  int hora;
  bool am;
  int minuto;
  int segundo;
};
```

- `Tiempo` es el nombre de la estructura, y define un nuevo tipo de dato. Así, se pueden definir nuevas variables de este tipo usando la sintaxis normal de declaración de variables:

  ```
  Tiempo ahora, vectorTiempo[10], matrizTiempo[5][7];
  ```

- `hora`, `am`, `minuto`, `segundo` se conocen como campos o valores miembro de la estructura. Los miembros de una estructura pueden ser de cualquier tipo y una estructura puede contener miembros de muchos tipos diferentes (incluso otras estructuras).
- Los identificadores de los miembros una estructura deben ser únicos, pero dos estructuras diferentes pueden tener, sin conflictos, miembros con el mismo nombre.

Índice

Inicialización

Una estructura se puede inicializar en la declaración usando la misma notación que para inicializar vectores.

Ejemplo

```
Tiempo ahora = { 10, false, 15, 20};

Punto origen = {0, 0};

Alumno estudiante = {"24236946Y", "Juan Lopez", 1, 'B'};

Fecha examen = {10, 2, 2006};
```

Operadores de acceso a valores miembros de la estructura

Los miembros de un estructura se acceden mediante:

- El operador punto (.)
  Accede a un miembro a través del nombre de la variable del tipo de la estructura.

```
Tiempo ahora;
.............
cout << ahora.hora;
```

- El operador flecha (->)
  El operador flecha accede a un miembro a través de la dirección de memoria de una variable del tipo de la estructura. Este operador se aplica sobre variables que almacenan la dirección de memoria donde se encuentra la estructura.
  Como el uso y manejo de los punteros será motivo de estudio del tema siguiente, en dicho tema se verá su aplicación como operador de acceso de estructuras. En este tema únicamente se usará como operador de acceso el operador punto.

Operación de asignación

Dicha operación se puede realizar:

- De forma individual sobre cada uno de los miembros de la estructura, combinando las operaciones de acceso con el operador de asignación

  ```
  <OperacionAcceso> = <expresion>;
  ```

  Para que la operación sea correcta, el tipo del miembro al que se accede debe coincidir o ser compatible con el tipo que devuelve la expresión.

  ```
  Tiempo.ahora;
  ..................
  ahora.hora = 1;
  ahora.am = false;
  ahora.minuto = (48+15)%60;
  ```

  En general, `<variable>.<miembro>` es una variable y se comporta como cualquier variable.

Operación de asignación

Dicha operación se puede realizar:

- de forma completa, asignando a una estructura otra estructura del mismo tipo.

```
Tiempo ahora, despertador;
.........................

ahora.hora = 8;
ahora.am = true;
ahora.hora = 0;
ahora.minuto = 0;
.................
despertador = ahora;
```

Operaciones de entrada y salida

En principio, se deben realizar individualmente sobre cada valor miembro de la estructura, y consiste en combinar las operaciones de entrada y salida con las operaciones de acceso a valores miembro.

Ejemplo

```
Tiempo ahora;
....
cout << "Introduzca la hora (0-23): ";
cin >> ahora.hora;
ahora.am = ahora.hora < 12;
if (!ahora.am)
  ahora.hora = ahora.hora - 12;
cou << "Introduzca los minutos (0-59): ";
cin >> ahora.minuto;
cout << "Introduzca los segundos (0-59): ";
cin >> ahora.segundo;
....
```

Ejemplo (continuación)

```
....
cout << ahora.hora << ":" << ahora.minuto <<
     ":" << ahora.segundo;
if (ahora.am)
  cout <<" AM";
else
  cout <<" PM";
cout << endl;
```

Estructuras y modularización

Las estructuras se comportan en C++ como si fueran tipos de datos básicos cuando se utilizan como argumento de los módulos:

- Se pueden pasar por valor:

```
void ImprimirUniversal (Tiempo t);
```

- Para pasarlos por referencia a un módulo es necesario anteceder su identificador con el operador `&`:

```
void LeerIntervaloTiempo (Tiempo &inicio, Tiempo &fin);
```

- Las funciones pueden devolver estructuras:

```
Tiempo DiferenciaTiempo (Tiempo actual, Tiempo hasta);
```

¡Cuidado!

Cuando se pasa una estructura que no se va a modificar en un módulo, de forma natural, lo pasaríamos por valor. Pero a diferencia de los tipos de datos básicos, las estructuras pueden ocupar mucha memoria, y el paso por valor implica hacer una copia de la estructura en la pila.

Para evitar esto, se usa el PASO POR REFERENCIA CONSTANTE.

```
const <tipo_estructura> &<parametro_formal>
```

Por ejemplo,

```
void ImprimirUniversal (const Tiempo &t);

Tiempo DiferenciaTiempo (const Tiempo &actual,
                         const Tiempo &hasta);
```

El paso por referencia constante se comporta distinto a un paso por valor. El comportamiento en un paso por valor es el siguiente:

```
void ImprimirUniversal (Tiempo t){
   .........
   t.minuto=17; // Permitido
   .........
```

mientras que en un paso por referencia constante,

```
void ImprimirUniversal (const Tiempo & t){
   .........
   t.minuto=17; // NO Permitido
   .........
```

y el compilador detecta que esa operación es errónea. Pero recordemos: NO QUEREMOS MODIFICARLO.
El paso por referencia constante, pasa una referencia sobre el dato con el que se quiere trabajar (con lo que evitamos una copia que puede ocupar mucha memoria) pero lo protege para que no se pueda modificar el dato original.

¡Cuidado!

¡Cuidado! al hacer un `return` también se copia la estructura por lo que se suele devolver un parámetro por referencia en su lugar.

Por ejemplo,

```
Tiempo DiferenciaTiempo (const Tiempo &desde,
                         const Tiempo &hasta);
```

sería preferible escribirlo como

```
void DiferenciaTiempo (const Tiempo &desde,
                       const Tiempo &hasta,
                       Tiempo &diferencia);
```

Un ejemplo de uso del tipo Tiempo

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

struct Tiempo{
  int hora;
  bool am;
  int minuto;
  int segundo;
}
/* Prototipos */
void PonerHora(int hora, bool am, int minuto, int segundo,
               Tiempo &t);
void segundosATiempo(int segundos, Tiempo &t);
void CogerTiempoActual(Tiempo &t);
void DiferenciaTiempo(const Tiempo &desde,
                      const Tiempo &hasta,
                      Tiempo &diferencia);
void ImprimirUniversal(const Tiempo &t);
```

```
int main(){
  Tiempo ahora, cena, que_queda;

  CogerTiempoActual(ahora);
  cout << "Son las ";
  ImprimirUniversal(ahora);

  PonerHora(10, false, 0, 0, cena)
  DiferenciaTiempo(ahora, cena, que_queda);
  cout << ". Aun quedan "
  ImprimirUniversal(que_queda);
  cout << " hasta la cena\n";
  return 0;
}

void PonerHora(int hora, bool am, int minuto, int segundo,
               Tiempo &t){
  t.hora = hora;
  t.am = am;
  t.minuto = minuto;
  t.segundo = segundo;
}
```

```
void segundosATiempo(int segundos, Tiempo &t){
  t.segundo = segundos % 60;
  segundos = segundos / 60;
  t.minuto = segundos % 60;
  t.hora = segundos / 60;
  t.am = t.hora <= 12;
  if (!t.am)
    t.hora = t.hora - 12;
}

CogerTiempoActual(Tiempo &t){
  const int SEG_DIA=86400; //60*60*24
  time_t horaSistema;

  horaSistema=time(0); // Segundos desde 1/1/1970 0:0:0
  segundosATiempo(horaSistema % SEG_DIA, t);
}
```

```
void DiferenciaTiempo(const Tiempo &desde,
                      const Tiempo &hasta,
                      Tiempo &diferencia);
  int seg_desde, seg_hasta, resta;
  seg_desde=desde.hora*3600 + desde.minuto*60 + desde.segundo;
  if (!desde.am)
    seg_desde=seg_desde + 12*3600;
  seg_hasta=hasta.hora*3600 + hasta.minuto*60 + hasta.segundo;
  if (!hasta.am)
    seg_hasta=seg_hasta + 12*3600;

  resta=seg_hasta-seg_desde;

  if (resta<0)
    ponerHora(0, true, 0, 0);
  else
    segundosATiempo(resta, diferencia);
}
```

```
void  ImprimirUniversal(const Tiempo &t){
  if (t.hora<10)
    cout << "0" << t.hora << ":":
  else
    cout << t.hora << ":";

  if (t.minuto<10)
    cout << "0" << t.minuto << ":":
  else
    cout << t.minuto << ":";

  if (t.segundo<10)
    cout << "0" << t.segundo:
  else
    cout << t.segundo;

  if (t.am)
    cout << " AM";
  else
    cout << " PM";
}
```

Estructuras y matrices

Una matriz de cualquier dimensión (y como caso particular un vector) puede ser un tipo asociado a un miembro de una estructura.

El acceso a los elementos de la matriz, se realiza mediante la combinación de las operaciones de acceso a valores miembros y las operaciones de acceso a los elementos de la matriz.

ejemplo

Concebir un vector como una estructura con dosmiembros:

```
struct vector50_int{
  int util; // num de elementos usados
  int vector[50]; // los datos del vector
};

vector50_int v;
```

- Puedo asignar el valor 3 a la componente 11 del vector

```
v.vector[5]=3;
v.util = 12;
```

- Puedo inicializar la estructura completa en la declaración

```
vector50_int v = {4, {3, 4, 5, 6}};
```

- Puedo leer el valor de una componente del vector

```
cin >> v.vector[7];
```

- Puedo usar funciones definidas para trabajar con vectores

```
Ordena(v.vector, v.util);
```

Trabajar con vectores o matrices incluidas como miembros de una estructura tiene la siguiente peculiaridad:
Si definimos dos vectores `v1` y `v2` de la forma normal, es decir,

```
int v1[50], v2[50];
```

e intentamos hacer una copia de `v1` en `v2` de la siguiente forma:

```
v2=v1;
```

se produce un error ya que las copias de vectores se deben hacer componente a componente.
Sin embargo, si utilizamos una definición como la anterior

```
vector50_int v1, v2;
```

la operación de asignación `v2 = v1` realiza una asignación completa entre las dos estructuras, incluyendo la copia de los vectores.

Se pueden definir vectores y en general matrices de estructuras. El acceso a los diferentes miembros de las estructuras, se realiza combinando el acceso a los elementos de la matriz con las operaciones de acceso a los miembros de la estructura .

```
struct Alumno{
  int codigo;
  int pract_realizadas;
  double nota_practicas[3];
  double nota_teoria;
};

int main(){
  Alumno ListaAlumnos[100];
  .........
```

Dada estas definiciones, las siguientes operaciones son correctas:

```
ListaAlumnos[0].codigo=1;
ListaAlumnos[10].pract_realizadas=1;
cin >> ListaAlumnos[3].nota_teoria;
cout << ListaAlumnos[5].practicas[0];
ListaAlumnos[1]=ListaAlumnos[0];
```

Estructuras de estructuras

Como caso particular, un valor miembro de una estructura puede ser a su vez otra estructura.

Ejemplo

Supongamos que se está desarrollando un programa para controlar el acceso de una serie de usuarios a un determinado edificio a través de una puerta con un lector de tarjetas. La información relevante para la gestión es la siguiente:

Código de usuario	Un entero [0,3000]
Fecha de uso	Dias, mes y año
Hora de uso	Hora, AP o PM, minuto y segundo
Salida o Entrada	Fijar si entra o sale del edificio

Con esta descripción, podemos definir una estructura que mantenga la información relativa al control de entradas y salidas del edificio con la siguiente definición:

```
struct Fecha{
  int dia; // 1-31
  int mes; // 1-12
  int anio; // 2005-9999;
};

struct ControlEntrada{
  int codigo;
  Fecha dia_uso;
  Tiempo hora_uso; // Definida anteriormente
  bool entrada; // true: entrada, false: salida
};
```

La estructura `ControlEntrada` define una agrupación de distintos datos en una única estructura de datos, entre los hay dos valores miembro que a su vez son estructuras.

El acceso a los miembros de las estructuras que son estructuras, se realizan con las operaciones de acceso, es decir, con el operador punto y el operador flecha. Así, si se define

```
ControlEntrada controlador;
```

Las siguientes operaciones de acceso se interpretan como:

```
controlador.codigo
```

Accede al miembro `codigo` de la variable controlador

```
controlador.hora_uso.hora
```

Accede al miembro hora de la estructura `hora_uso` (de tipo `Tiempo`) de la variable `controlador`.

```
controlador.HoraUso.mes
```

ERROR, no hay ningún miembro `mes` que pertenezca a una estructura de tipo `Tiempo`

Un ejemplo de uso

Se define un diccionario español-inglés como una estructura:

```
const int MAX_NUM_PALABRAS=1000000;
struct diccionario{
    entrada ingles[MAX_NUM_PALABRAS];
    entrada espanol[MAX_NUM_PALABRAS];
    int num_palabras;
};
```

`num_palabras` contiene el número de palabras que hay actualmente en el diccionario, y `entrada` es la estructura:

```
struct entrada{
    char palabra[40];
    int traduccion;
};
```

donde se almacena, junto a cada palabra (`palabra`) que es una cadena de caracteres de C (cstring), el índice (`traduccion`) de la entrada de su traducción en el otro vector del diccionario.

Realizar la función que, dada una palabra en español almacenada en un cstring y un diccionario, devuelve en un cstring con la traducción de dicha palabra al inglés. Si la palabra no estuviese en el diccionario, el módulo devuelve un código de error `false` y si la traducción ha tenido éxito, se devuelve un valor `true`.

```
bool traduce_palabra_esp_ing(const char palabra[],
                             const diccionario &dic,
                             char traduccion[]){
  int pos = busca(palabra, dic.espanol, dic.num_palabras);
  if(pos == -1)
    return false;
  else{
    int trad = dic.espanol[pos].traduccion;
    strcpy(traduccion, dic.ingles[trad].palabra);
    return true;
  }
}
```

Ejercicio

Realizar una función que busque una `palabra` en un vector de `entrada` y devuelva la posición donde se encuentra la palabra o -1 si no está.

Imprimir el contenido del diccionario

```
void imprime(const diccionario &dic){
    cout << "Contenido del diccionario:\n";
    cout << "Número de palabras: " << dic.num_palabras;
    cout << "\nPalabras en español y traducción en inglés\n";
    for (int i=0; i< dic.num_palabras; i++)
      cout << dic.espanol[i].palabra << "\t"
           << dic.ingles[dic.espanol[i].traduccion].palabra
           << endl;
}
```

Añadir una pareja de palabras en el diccionario.

```
bool anade_al_diccionario(const char esp[], const char ing[],
             diccionario &dic){
  if(busca(esp, dic.espanol, dic.num_palabras)==-1
     && busca(ing, dic.ingles, dic.num_palabras)==-1){
     // Nota: tenemos que buscar en los dos diccionarios
     // antes de añadir para evitar tener entradas repetidas

     // Insertamos las palabras y sus traducciones
     strcpy(dic.espanol[dic.num_palabras].palabra, esp);
     strcpy(dic.ingles[dic.num_palabras].palabra, ing);
     dic.espanol[dic.num_palabras].traduccion =
                                          dic.num_palabras;
     dic.ingles[dic.num_palabras].traduccion =
                                          dic.num_palabras;
     dic.num_palabras++; // tenemos una palabra más
     return true;
  } else // alguna palabra ya está en el diccionario
    return false;
}
```

Resumen

- Se han presentado las estructuras como un tipo de dato capaz de agrupar información de diferentes tipos de datos.
- Se han introducido las operaciones básicas con estructuras.
- Se han visto los problemas que supone pasar estructuras por valor a funciones y sus soluciones.
- Se han creado tipos de datos complejos y se ha visto la forma de usarlos.