Universidad LOYOLA

Departamento de Métodos Cuantitativos

Tema 4. Tipos de Datos Compuestos

Alfonso Carlos Martínez Estudillo (acme@uloyola.es)

Fundamentos de Informática I

1º de Grado en Ingeniería Informática y Tecnologías Virtuales

Curso 2021-2022

Índice de contenidos

Índice de contenidos

Tipos de datos compuestos

- En casi todos los problemas, es necesario mantener una relación entre variables diferentes o almacenar y referenciar variables como un grupo.
- Los lenguajes ofrecen tipos más complejos para tal fin.
- Un **tipo de dato compuesto** es una composición de tipos de datos simples o compuestos, caracterizado por:
  - La organización de sus datos.
  - Las operaciones que se definen sobre él.

Tipos de datos compuestos

Distinguimos:

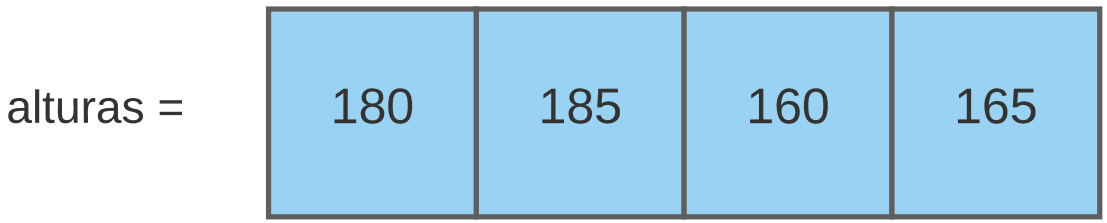
- **Homogéneos**:
  - Varias componentes del mismo tipo simple.
  - **Arrays** (vectores, matrices, n-dimensionales).
- **Heterogéneos**:
  - Varias componentes de diferentes tipos simples.
  - **Estructuras**, uniones, ...

Ejemplo

- Imaginemos que queremos calcular la altura media de los alumnos de un grupo formado por tres personas.
  - En este caso necesitamos tres variables para almacenar la altura.
  - Una variable si queremos almacenar la media.
- Supongamos ahora que el grupo está formado por 30 alumnos.
  - Esto supondría tener que declarar un total de 30 variables.
  - Sería tedioso y código bastante extenso.
- Y si el grupo es de cien personas o queremos aplicar el código a varias clases...
  - Deberíamos cambiar el código cada vez para cada caso.

Solución

Se introduce un nuevo tipo de dato que permita representar dichas variables en una única estructura de datos, reconocida bajo un nombre único.



Arrays: definición

- Tipo de dato compuesto homogéneo, es decir, todas las componentes son del mismo tipo.
  - int, float, double, char, ...
- Número fijo de componentes.
  - Tamaño del array.
- Tienen un nombre único.
- Tipos:
  - **Vectores**: arrays de una dimensión.
  - **Matrices**: arrays de dos dimensiones.
  - **Arrays de n-dimensiones**.

Índice de contenidos

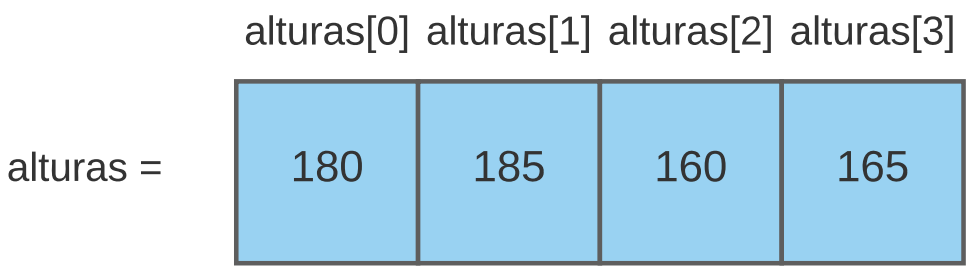
Declaración de un vector

<tipo> <identificador> [<tamaño>]

```
1 int notas[4];
```

Donde distinguimos:

- **<tipo>**: indica el tipo de dato de todos los elementos del array.
- **<identificador>**: nombre por el que se referencian.
- **<tamaño>**: número de elemento del array.
- El índice de la primera posición del array es 0.
- El índice de la última posición del array es <tamaño> - 1.

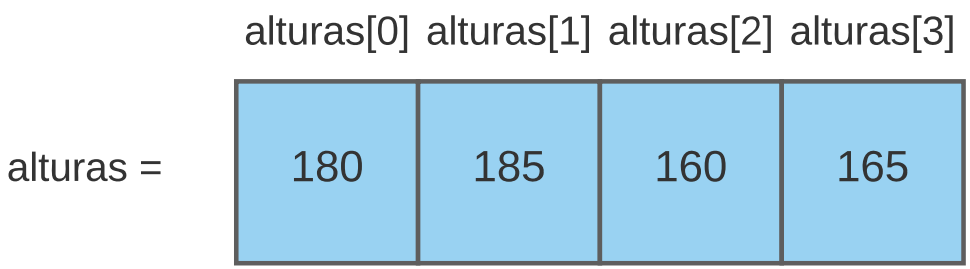


Almacenamiento en memoria

Una variable de tipo vector:

- Ocupa una cantidad fija de memoria, que NO es posible alterar durante la ejecución del programa.
- Esta cantidad de memoria debe conocerse antes de la ejecución del programa.
  - Número de elementos conocidos en tiempo de compilación.
  - Se pueden utilizar constantes.
- Las posiciones están contiguas en memoria.

```
1 int notas[4];
```



Tamaño de los vectores

- El tamaño de un vector: número de elementos que puede contener.
- Tamaño de los vectores en memoria: se usa el operador **sizeof**.
- Si usamos **sizeof** para un vector, este devuelve el tamaño en bytes para el vector completo.

```
1 int notas[4];
2 n=sizeof(notas);
3 //Sabiendo que un int ocupa 4 byte, n tomará el valor de 16.
```

- Podemos solicitar el tamaño en memoria de un elemento concreto del array.

```
1 int notas[4];
2 n=sizeof(notas[1]);
3 //n tomará el valor de 4.
```

Acceso

<identificador> [<índice>]

```
1 notas[2];
```

- Se puede acceder a cada elemento utilizando un índice entero en el nombre del vector.
- El índice tomará valores desde 0 hasta el número de elementos menos 1.
- El índice o posición de un elemento es siempre el mismo que el número de pasos desde el elemento inicial.
- Supongamos el siguiente vector:

```
1 float v[3];
```

  - Cada componente es de tipo **float**.
  - Se puede acceder desde v[0] hasta v[2].
  - Fuera de ese rango, no son accesos correctos.

Acceso

- C++ no comprueba que los índices a los que se accede estén en el rango definido y, por lo tanto, no dará un error al compilar, pero sí lo dará al ejecutar.
- En este caso acceder a la posición 20 del vector anterior daría un fallo en la ejecución.
- Se pueden referenciar elementos utilizando fórmulas para los subíndices, siempre que el resultado sea un entero.

```
1 v[3+4];
2 v[t+5];
3 v[mes];
4 ...
```

Asignación

<identificador> [<índice>]=<Expresión>

- **<Expresión>**: ha de ser del mismo tipo que el definido en la declaración del vector o al menos compatible.
- La asignación se realiza componente a componente.

```
1 v[3]=10*5;
```

- No se permiten asignaciones globales sobre todos los elementos del vector, salvo al declarar la variable.

Inicialización

- Se deben asignar valores a los elementos de un vector antes de utilizarlo.
  - Asignar valores elemento a elemento.

```
1 int v[2];
2 v[0]=0;
3 v[1]=1;
```

  - Asignar valores en una única sentencia.

```
1 int v[2] = {0,1} //Inicializa v[0]=0, y v[1]=1
2 int v[] = {0,1,2} //El compilador asume que la longitud
      es 3
```

  - Asignar valores en un bucle (for, while, do-while).

```
1 for(int i=0; i<3; i++)
2 {
3   v[i]=i;
4 }
```

Lectura y escritura

- Se realiza componente a componente.
- Se utiliza un bucle para leerlas.

```
1 #define DIM 20
2 ...
3 int v[DIM];
4 for(int i=0; i<DIM; i++)
5 {
6   cout << "Introduzca el valor de la posicion " << i << ": ";
7   cin >> v[i];
8   cout << "El valor introducido es: " << v[i] << endl;
9 }
```

Recorrido de un vector

Ejemplo (media de los valores de un vector):

```
1  #include <iostream>
2  #define NUM_ALUMNOS 10
3  using namespace std;
4  int main(){
5    int alturas[NUM_ALUMNOS];
6    float media=0.0;
7    for(int i=0; i<NUM_ALUMNOS; i++){
8      cout << "Introduce altura " << i << ": ";
9      cin >> alturas[i];
10   }
11   for(int i=0; i<NUM_ALUMNOS; i++)
12   {
13     media = media + alturas[i];
14   }
15   media=media/NUM_ALUMNOS;
16   return 0;
17 }
```

Recorrido de un vector

- En nuestros programas sería conveniente definir vectores con un tamaño medianamente grande.
- Luego utilizaremos sólo aquellas primeras posiciones que sean necesarias.
- Para ello, definiremos otra variable que controle el número de posiciones útiles.

Recorrido de un vector

Ejemplo:

```
1  #include <iostream>
2  #define NUM_ALUMNOS 100
3  using namespace std;
4  int main(){
5    int alturas[NUM_ALUMNOS];
6    int n;
7    do{
8      cout << "Introduzca el número de alumnos: ";
9      cin >> n;
10   }while(n<0 || n>NUM_ALUMNOS);
11   for(int i=0; i<n; i++)
12   {
13     cout << "Introduce altura " << i << ": ";
14     cin >> alturas[i];
15   }
16   return 0;
17 }
```

Tratamiento selectivo

```
1  #include<iostream>
2  #define DIM_MAX 100
3  using namespace std;
4  int main(){
5    int v[DIM_MAX];
6    int suma = 0, n=50;//Longitud util de nuestro vector
7    for(int i=0; i<n; i++){
8      v[i]=i;
9    }
10   for(int i=0; i<n; i++){
11     if(v[i]%2==0){
12       suma=suma+v[i];
13     }
14   }
15   cout << "La suma de los pares es: " << suma << endl;
16   return 0;
17 }
```

Búsqueda secuencial

```
1  #include<iostream>
2  #define DIM_MAX 100
3  using namespace std;
4  int main(){
5    float v[DIM_MAX], elemento=5;
6    int i=0, posicion=-1, n=20,
7    bool encontrado=false;
8    ...
9    while((i<n)&&(!encontrado)){
10     if(v[i]==elemento){
11       posicion=i;
12       encontrado=true;
13     }
14     i++;
15   }
16   ...
17 }
```

Búsqueda secuencial

```
1  #include<iostream>
2  #define DIM_MAX 100
3  using namespace std;
4  int main(){
5    float v[DIM_MAX], elemento=5;
6    int i=0, posicion=-1, n=20;
7    bool encontrado=false;
8    ...
9    for(i=0; (i<n)&&(!encontrado); i++){
10     if(v[i]==elemento){
11       posicion=i;
12       encontrado=true;
13     }
14   }
15   ...
16 }
```

Índice de contenidos

Arrays multidimensionales

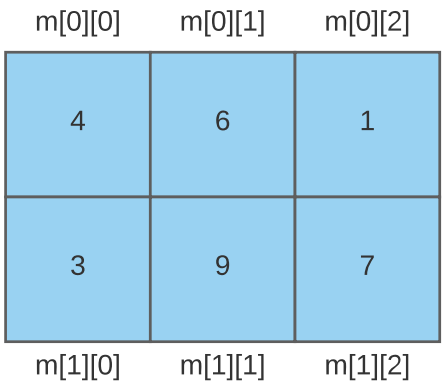
- Declaración:
  <**tipo**> <**identificador**> **[d1][d2]...[dn];**.
- Todas las dimensiones han de ser de tipo entero, y empiezan a enumerarse en cero, como en los vectores.

```
1 int m[3][5];
2 float m2[3][5][7];
```

- Un array bidimensional es una matriz:
  <**tipo**> <**identificador**> **[n_filas][n_columnas];**.

Almacenamiento en memoria

- Una matriz int[2,3] puede verse como:



- Todas las posiciones están realmente contiguas en memoria.
- En C++ la representación exacta sería:

Acceso

$$<\textbf{identificador}> [\textbf{ind}_1][\textbf{ind}_2]...[\textbf{d}_{n-1}];.$$

- Ejemplos:

```
1 float m[2][3][4];
2 m[1][2][3]; m[0][0][0] // son correctos
3 m['2'][0][0], m[0][0][4] // son incorrectos
```

- La forma de acceder a un elemento será por las coordenadas, en una matriz sería:

```
1 m[fila][columna]
```

Inicialización

- Como ocurre con los vectores, podemos inicializar una matriz en la misma declaración. La forma segura es poner entre llaves los valores de cada fila.
- Ejemplo:

```
1 int m[2][3] = {{1,2,3},{4,5,6}}.
```

- También se pueden inicializar elemento a elemento, recorriendo la matriz con bucles anidados.

Buscar un elemento

```
1  int i, j;
2  bool encontrado = false;
3  float elemento, m[MAX_F][MAX_C];
4  ...
5  i=0;
6  while((!encontrado)&&(i<filas)){
7    j=0;
8    while((!encontrado) && (j<columnas)){
9      if(m[i][j]==elemento){
10       encontrado=true;
11     }
12     j++
13   }
14   i++
15 }
16 ...
```

Buscar un elemento

```
1  bool encontrado = false;
2  float elemento, m[MAX_F][MAX_C];
3  ...
4
5  for(int i=0; i<filas; i++){
6    for(int j=0; j<columnas; j++){
7      if(m[i][j]==elemento){
8        encontrado=true;
9      }
10   }
11 }
12 ...
```

Lectura en array 3-dimensional

```
1  ...
2  int m[MAX_FIL][MAX_COL][MAX_PROF];
3  int filas, columnas, profundidades;
4  for(i=0; i<filas; i++){
5    for(j=0; j<columnas; j++){
6      for(k=0; k<profundidades; k++){
7        cout << "Elemento[" << i << "][" << j << "]["<< k << "]: ";
8        cin >> m[i][j][k];
9      }
10   }
11 }
12 ...
```

Multiplicación de matrices

```
1  int A[MAX_FIL][MAX_COL],B[MAX_FIL][MAX_COL],R[MAX_FIL][MAX_COL];
2  int filasA, columnasA, filasB, columnasB;
3  for(int i=0; i<filasA; i++)
4  {
5    for(int j=0; j<columnasB; j++)
6    {
7      R[i][j]=0;
8      for(int k=0; k<columnasA; k++)
9      {
10       R[i][j]=R[i][j]+A[i][k]*B[k][j];
11     }
12   }
13 }
```

Recorrido de una matriz por filas

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4    int filas=3, columnas=3;
5    int  m[filas][columnas]={{1,2,3},{4,5,6},{7,8,9}};
6    for(int i=0; i<filas; i++)
7    {
8      for(int j=0; j<columnas; j++)
9      {
10       cout << "Matriz[" << i << "][" << j <<"]: " << m[i][j];
11       cout << endl;
12     }
13   }
```

Recorrido de una matriz por columnas

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4    int filas=3, columnas=3;
5    int  m[filas][columnas]={{1,2,3},{4,5,6},{7,8,9}};
6    for(int j=0; j<columnas; j++)
7    {
8      for(int i=0; i<filas; i++)
9      {
10       cout << "Matriz[" << i << "][" << j <<"]: " << m[i][j];
11       cout << endl;
12     }
13   }
```

Índice de contenidos

Estructuras

- Una estructura es una colección de uno o más datos que no tiene porque ser del mismo tipo, agrupados bajo un mismo nombre.
- Cada dato recibe el nombre de campo.
- Se trata de un tipo de dato derivado creado a partir de tipos ya definidos.
- Ejemplo de una estructura para representar los datos de una persona:

Nombre del campo	Tipo de Dato
Nombre	Cadena
Edad	Entero
Altura	Flotante
Peso	Flotante
Sexo	Carácter

Definición de estructuras

- Al definir una estructura, estamos definiendo un tipo de dato nuevo.
- Debemos definirlo antes de utilizarlo.
- La sintaxis es:

```
struct <nombre estructura>{
  tipo campo1;
  tipo campo2;
  ...
  tipo campoN;
};
```

- En el ejemplo anterior:

```
struct persona{
  string nombre;
  int edad;
  float altura;
  float peso;
  char sexo;
};
```

Definición de tipos

- El nuevo tipo de dato sería: **struct persona**.
- Los miembros o campos: **nombre, edad, altura, peso y sexo**.
- Hay que definirlo antes del *main* o en un fichero de cabecera (lo veremos en el próximo tema), si queremos utilizar el tipo más de una vez.
  - En varias funciones.
  - En varios ficheros.

Declaración de variables

Se pueden declarar variables del nuevo tipo:

- Listándolas después de la definición (serían variables globales):

```
1 struct persona{
2   string nombre;
3   int edad;
4   float altura;
5   float peso;
6   char sexo;
7 } persona1 persona2;
```

- Declarándolas como variables como cualquier otro tipo:

```
1 struct persona persona1 persona2;
```

Inicialización y asignación

- Al declarar una variable de tipo **struct** la podemos inicializar.
  - Le asignamos los valores encerrados entre {}.
  - Los valores irán en el mismo orden en que se han declarado los campos.

```
1 struct persona persona1 = {Antonio, 30, 191, 90, 'H'};
```

- Se pueden asignar dos variables de tipo **struct**, es decir, no es necesario hacerlo campo a campo.

```
1 struct persona persona1 = {Antonio, 30, 191, 90, 'H'};
2 struct persona persona2;
3 persona2 = persona1;
```

Comparación

- Dos variables de tipo **struct** no se pueden comparar con el operador ==.
- Dos estructuras serán iguales si lo son campo a campo.

```
1  struct persona p1 = {Antonio, 30, 191, 90, 'H'};
2  struct persona p2;
3  p2 = p1;
4  //Modo Incorrecto
5  if (p1 == p2){
6    cout << "Son iguales" << endl;
7  }
8  //Modo Correcto
9  if((p1.nombre==p2.nombre) && (p1.edad == p2.edad) ...){
10   cout << "Son iguales" << endl;
11 }
```

Acceso a las estructuras

- Acceso a los campos de una estructura por nombre y utilizando el operador punto (.).
  variable.campo.
- El operador punto conecta el nombre de la variable con el nombre del campo. Por ejemplo:
  - *persona1.nombre*: se accede al campo nombre de la variable persona1.
  - *persona2.altura*: se accede al campo altura de la variable persona2.
- Los campos de una estructura se comportan como variables normales (almacenar y recuperar información).

Acceso a las estructuras

```
#include <iostream>
using namespace std;
struct persona{
   string nombre;
   int edad;
   char sexo;
};
int main(){
   struct persona p;
   cout << "Introduce el nombre: ";
   cin >> p.nombre;
   cout << "Introduce la edad: ";
   cin >> p.edad;
   cout << "Introduce el sexo: ";
   cin >> sexo;
   cout << "Persona: " << p.nombre << ", " << p.edad << ", " << p.sexo <<
       endl;
   return 0;
}
```

Tamaño de una estructura

- El tamaño de una estructura, al igual que el de cualquier tipo, se puede saber utilizando el operador **sizeof**.
- El tamaño de una estructura es la suma de los tamaños de sus campos.

```
1  #include<iostream>
2  using namespace std;
3  struct punto{
4    int x;
5    int y;
6  };
7  int main(){
8    struct punto p1 = {1,2};
9    cout << "Size: " << sizeof(p1) << endl;
10   cout << "Size: " << sizeof(struct punto) << endl;
11   cout << "Size: " << sizeof(p1.x)+sizeof(p1.y) << endl;
12   return 0;
13 }
```

Estructuras jerárquicas

Los campos de una estructura pueden ser a su vez, otras estructuras ya definidas previamente.

```
1  struct punto{
2    float x;
3    float y;
4  };
5
6  struct triangulo{
7    struct punto p1;
8    struct punto p2;
9    struct punto p3;
10 };
```

Acceso

```
1  struct punto{
2    float x;
3    float y;
4  };
5
6  struct triangulo{
7    struct punto p1;
8    struct punto p2;
9    struct punto p3;
10 };
11 ...
12 struct punto p1 = {0,0}, p2 = {1,5}, p3 = {-1,3}
13 struct triangulo t = {p1, p2, p3};
14 //Accedemos a la coordenada x del p1 del triangulo
15 cout << t.p1.x << endl;
16 ...
```

Arrays de estructuras

- Igual que declaramos un array de enteros, reales, ..., podemos declarar arrays cuyos elementos sean una estructura previamente definida.

```
struct punto v[10];
```

- Se tratan igual que cualquier otro array.
- Inicialización:

```
//Reserva memoria para dos elementos. Inicializa dos:
struct punto v[2] = {{1,1},{2,2}};
//Reserva memoria para diez elementos. Inicializa dos:
struct punto v[10] = {{1,1},{2,2}};
//Reserva memoria para dos elementos. Inicializa dos:
struct punto v[]={{1,1},{2,2}};
```

- Acceso: se accede al índice del array y luego al campo con el punto.

```
v[1].x
```

- Los campos de una estructura también pueden ser arrays.

Typedef

- Instrucción que permite la creación de sinónimo o alias para tipos de datos ya definidos.
- Formato: **typedef <tipo a redefinir > <nuevo nombre>;**
- La idea es asignar nombres más cortos o más acordes con lo que se quiere representar.

```
1 typedef struct punto stPunto;
2 typedef struct persona stPersona;
```

Ejemplo 1

```
1  struct punto{
2    int x;
3    int y;
4  };
5  typedef struct punto stPunto;
6
7  struct triangulo{
8    stPunto p1;
9    stPunto p2;
10   stPunto p3;
11 };
12 typedef struct triangulo stTriangulo;
13
14 main(){
15   stPunto p;
16   stTriangulo t;
17 }
```

Ejemplo 2

```
1  typedef struct punto{
2    int x;
3    int y;
4  }stPunto;
5
6  typedef struct triangulo{
7    stPunto p1;
8    stPunto p2;
9    stPunto p3;
10 }stTriangulo;
11
12 main(){
13   stPunto p;
14   stTriangulo t;
15 }
```

Referencias

Recursos electrónicos:

- cppreference. (2020). Referencia C++: https://en.cppreference.com/w/
- Goalkicker.com (2018). C++: Note for Professionals.
- Grimes, R. (2017). Beginning C++ Programming.
- Joyanes, L. (2000). Programación en C++: Algoritmos, Estructuras de datos y Objetos.
- Juneja, B.L., Seth, A. (2009). Programming with C++.

Referencias

Libros:

- Prieto, A., Lloris, A., Torres, J.C. (2008). Introducción a la Informática (4 ed).
- Joyanes, L. (2008). Fundamentos de programación : algoritmos, estructura de datos y objetos.
- Martí, N., Ortega, Y., Verdejo, J.A. (2003). Estructuras de datos y métodos algorítmicos: Ejercicios resueltos.
- Tapia, S., García-Beltrán, A., Martínez, R., Jaen, J.A., del Álamo, J. (2012). Fundamentos de programación en C.

¿Preguntas?